3 Numerical Data

Objectives

After you have read and studied this chapter, you should be able to

- Select proper types for numerical data.
- Write arithmetic expressions in Java.
- Evaluate arithmetic expressions, following the precedence rules.
- Describe how the memory allocation works for objects and primitive data values.
- Write mathematical expressions, using methods in the **Math** class.
- Use the **GregorianCalendar** class in manipulating date information such as year, month, and day.
- Use the **DecimalFormat** class to format numerical data.
- Convert input string values to numerical data.
- Input data by using **System.in** and output data by using **System.out.**
- Apply the incremental development technique in writing programs.
- (*Optional*) Describe how the integers and real numbers are represented in memory.

Introduction

When we review the Ch2Monogram sample program, we can visualize three tasks: input, computation, and output. We view computer programs as getting input, performing computation on the input data, and outputting the results of the computations. The type of computation we performed in Chapter 2 is string processing. In this chapter, we will study another type of computation, the one that deals with numerical data. Consider, for example, a metric converter program that accepts measurements in U.S. units (input), converts the measurements (computation), and displays their metric equivalents (output). The three tasks are not limited to numerical or string values, though. An input could be a mouse movement. A drawing program may accept mouse dragging (input), remember the points of mouse positions (computation), and draw lines connecting the points (output). Selecting a menu item is yet another form of input. For beginners, however, it is easiest to start writing programs that accept numerical or string values as input and display the result of computation as output.

We will introduce more standard classes to reinforce the object-oriented style of programming. The Math class includes methods we can use to express mathematical formulas. The DecimalFormat class includes a method to format numerical data so we can display the data in a desired precision. The GregorianCalendar class includes methods to manipulate the date. In Chapter 2, we performed input and output using JOptionPane. We will describe another way of doing input and output using System.in and System.out.

Finally, we will continue to employ the incremental development technique introduced in Chapter 2 in developing the sample application, a loan calculator program. As the sample program gets more complex, well-planned development steps will smooth the development effort.

3.1 Variables

Suppose we want to compute the sum and difference of two numbers. Let's call the two numbers x and y. In mathematics, we say

```
x + y
```

and

```
x - y
```

To compute the sum and the difference of x and y in a program, we must first declare what kind of data will be assigned to them. After we assign values to them, we can compute their sum and difference.

Let's say x and y are integers. To declare that the type of data assigned to them is an integer, we write

```
int    x, y;
```

variable

When this declaration is made, memory locations to store data values for x and y are allocated. These memory locations are called ***variables***, and x and y are the names we associate with the memory locations. Any valid identifier can be used as a variable name. After the declaration is made, we can assign only integers to x and y. We cannot, for example, assign real numbers to them.

Helpful Reminder

A variable has three properties: a memory location to store the value, the type of data stored in the memory location, and the name used to refer to the memory location.

Although we must say "x and y are variable names" to be precise, we will use the abbreviated form "x and y are variables" or "x and y are integer variables" whenever appropriate.

The general syntax for declaring variables is

variable declaration syntax

```
<data type>     <variables> ;
```

where <variables> is a sequence of identifiers separated by commas. Every variable we use in a program must be declared. We may have as many declarations as we wish. For example, we can declare x and y separately as

```
int     x;
int     y;
```

However, we cannot declare the same variable more than once; therefore, the second declaration below is invalid because y is declared twice:

```
int     x, y, z;
int     y;
```

six numerical data types

There are ***six numerical data types*** in Java: byte, short, int, long, float, and double. The data types byte, short, int, and long are for integers; and the data types float and double are for real numbers. The data type names byte, short, and others are all reserved words. The difference among these six numerical data types is in the range of values they can represent, as shown in Table 3.1.

higher precision

A data type with a larger range of values is said to have a ***higher precision***. For example, the data type double has a higher precision than the data type float. The tradeoff for higher precision is memory space—to store a number with higher precision, you need more space. A variable of type short requires 2 bytes and a variable of type int requires 4 bytes, for example. If your program does not use many integers, then whether you declare them as short or int is really not that critical. The difference in memory usage is very small and not a deciding factor in the program design. The

Table 3.1 Java numerical data types and their precisions

Data Type	Content	Default Value†	Minimum Value	Maximum Value
byte	Integer	0	−128	127
short	Integer	0	−32768	32767
int	Integer	0	−2147483648	2147483647
long	Integer	0	−9223372036854775808	9223372036854775807
float	Real	0.0	−3.40282347E+38‡	3.40282347E+38
double	Real	0.0	−1.79769313486231570E+308	1.79769313486231570E+308

† No default value is assigned to a local variable. A local variable is explained on page 189 in Section 4.8.
‡ The character E indicates a number is expressed in scientific notation. This notation is explained on page 96.

storage difference becomes significant only when your program uses thousands of integers. Therefore, we will almost always use the data type int for integers. We use long when we need to process very large integers that are outside the range of values int can represent. For real numbers, it is more common to use double. Although it requires more memory space than float, we prefer double because of its higher precision in representing real numbers. We will describe how the numbers are stored in memory in Section 3.10.

You Might Want to Know

Application programs we develop in this book are intended for computers with a large amount of memory (such as desktops or laptops), so the storage space is not normally a major concern because we have more than enough. However, when we develop applications for embedded or specialized devices with a very limited amount of memory, such as PDAs, cellular phones, mobile robots for Mars exploration, and others, reducing the memory usage becomes a major concern.

Here is an example of declaring variables of different data types:

```
int      i, j, k;
float    numberOne, numberTwo;
long     bigInteger;
double   bigNumber;
```

At the time a variable is declared, it also can be initialized. For example, we may initialize the integer variables count and height to 10 and 34 as in

```
int count = 10, height = 34;
```

Take my Advice

As we mentioned in Chapter 2, you can declare and create an object just as you can initialize variables at the time you declare them. For example, the declaration

```
Date today = new Date();
```

is equivalent to

```
Date today;
today = new Date();
```

assignment statement

We assign a value to a variable by using an ***assignment statement***. To assign the value 234 to the variable named firstNumber, for example, we write

```
firstNumber = 234;
```

Be careful not to confuse mathematical equality and assignment. For example, the following are not valid Java code:

```
4 + 5 = x;
x + y = y + x;
```

assignment statement syntax

The syntax for the assignment statement is

```
<variable> = <expression> ;
```

where <expression> is an arithmetic expression, and the value of <expression> is assigned to the <variable>. The following are sample assignment statements:

```
sum       = firstNumber + secondNumber;
solution  = x * x - 2 * x + 1;
average   = (x + y + z) / 3.0;
```

We will present a detailed discussion of arithmetic expressions in Section 3.2. One key point we need to remember about variables is the following:

Helpful Reminder

Before using a variable, first we must declare and assign a value to it.

The diagram in Figure 3.1 illustrates the effect of variable declaration and assignment. Notice the similarity with this and memory allocation for object declaration and creation, illustrated in Figure 2.4 on page 38. Figure 3.2 compares

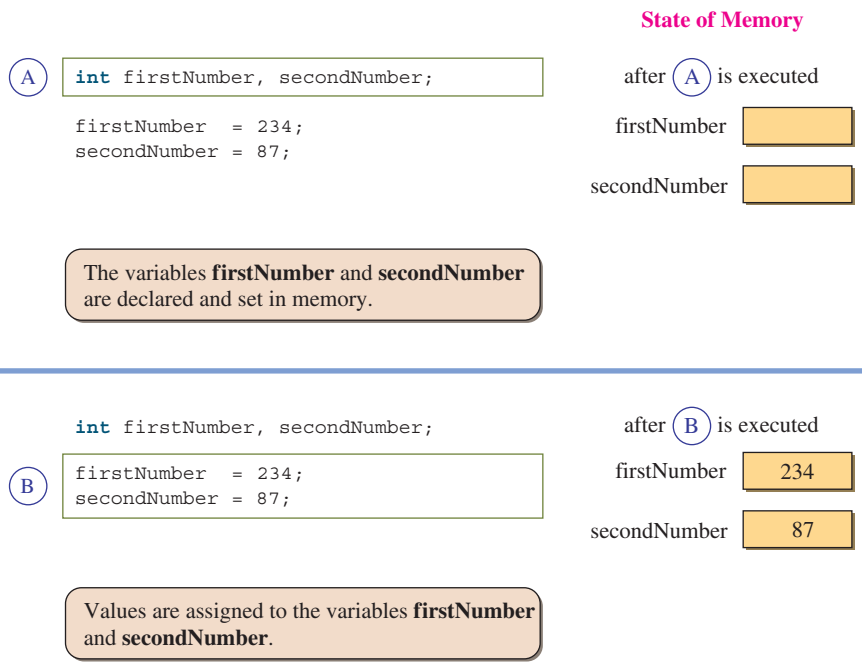


Figure 3.1 A diagram showing how two memory locations (variables) with names **firstNumber** and **secondNumber** are declared, and values are assigned to them.

the two. What we have been calling object names are really variables. The only difference between a variable for numbers and a variable for objects is the contents in the memory locations. For numbers, a variable contains the numerical value itself; and for objects, a variable contains an address where the object is stored. We use an arrow in the diagram to indicate that the content is an address, not the value itself.

Helpful Reminder

Object names are synonymous with variables whose contents are references to objects (i.e., memory addresses).

Figure 3.3 contrasts the effect of assigning the content of one variable to another variable for numerical data values and for objects. Because the content of a variable for objects is an address, assigning the content of a variable to another makes two variables that refer to the same object. Assignment does not create a new object. Without executing the new command, no new object is created. We can view the situation in which two variables refer to the same object as the object having two distinct names.

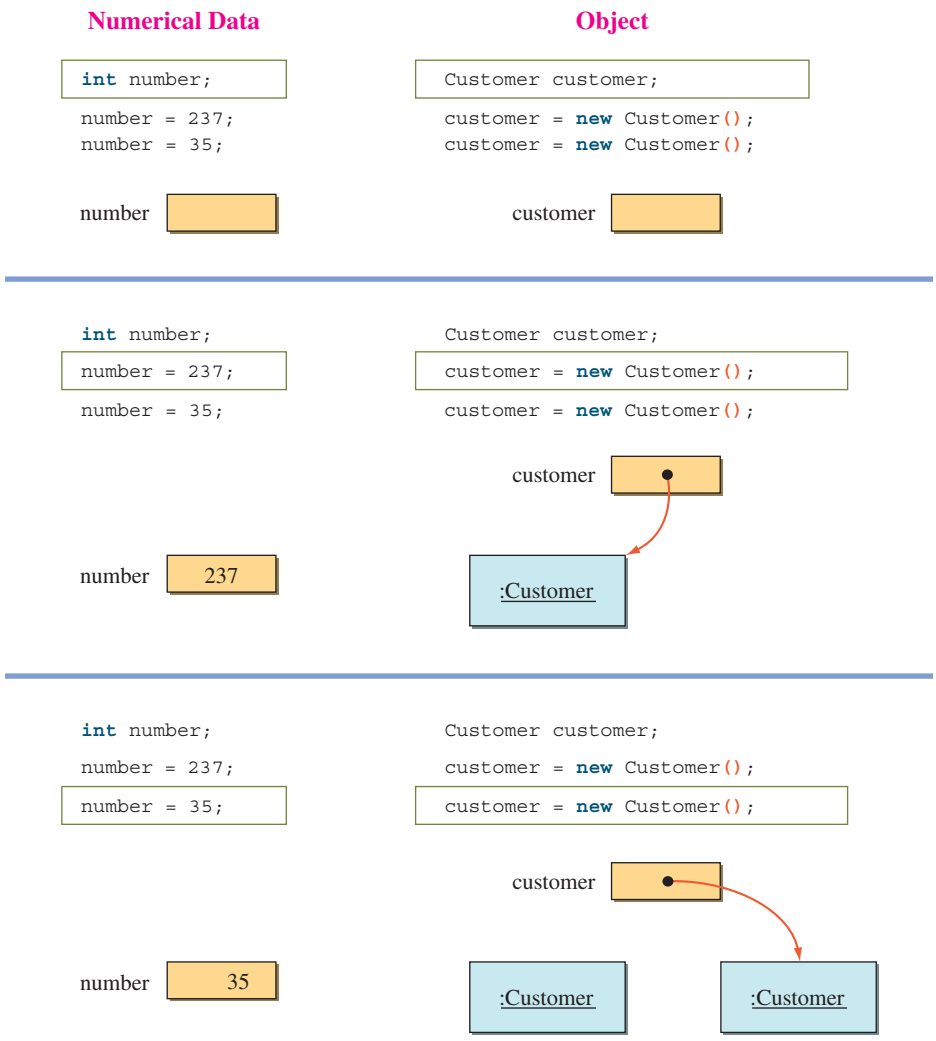


Figure 3.2 A difference between object declaration and numerical data declaration.

For numbers, the amount of memory space required is fixed. The values for data type int require 4 bytes, for example, and this won't change. However, with objects, the amount of memory space required is not constant. One instance of the Account class may require 120 bytes, while another instance of the same class may require 140 bytes. The difference in space usage for the account objects would occur if we had to keep track of checks written against the accounts. If one account has 15 checks written and the second account has 25 checks written, then we need more memory space for the second account than for the first account.

We use the new command to actually create an object. Remember that declaring an object only allocates the variable whose content will be an address. On the

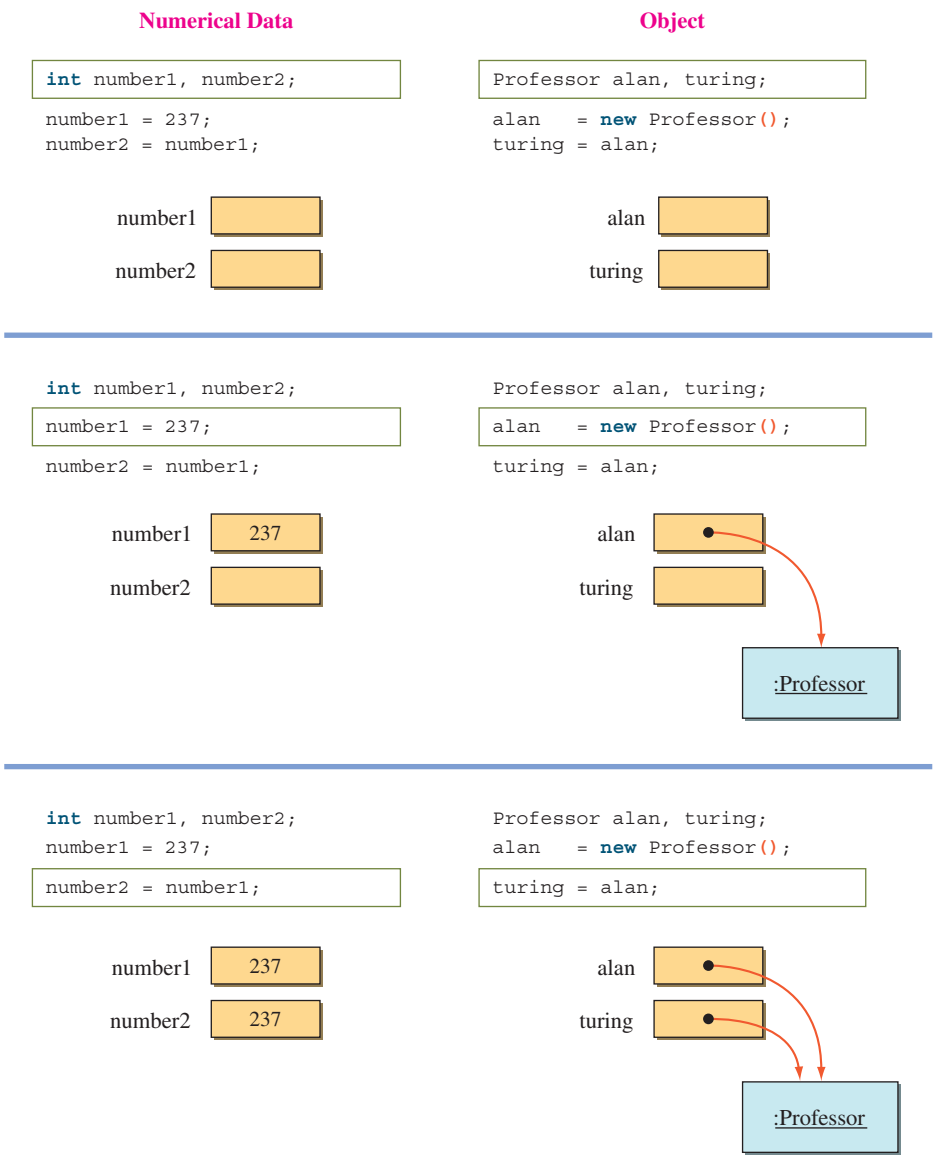


Figure 3.3 An effect of assigning the content of one variable to another.

reference versus primitive data types

other hand, we don't "create" an integer because the space to store the value is already allocated at the time the integer variable is declared. Because the contents are addresses that refer to memory locations where the objects are actually stored, objects are called ***reference data types***. In contrast, numerical data types are called ***primitive data types***.

Take my Advice

In addition to the six numerical data types, there are two nonnumerical primitive data types. The data type **boolean** is used to represent two logical values **true** and **false.** For example, the statements

```
boolean raining;
raining = true;
```

assign the value **true** to a boolean variable raining. We will explain and start using boolean variables beginning in Chapter 5. The second nonnumerical primitive data type is **char** (for character). It is used to represent a single character (letter, digit, punctuation marks, and others). The following example assigns the uppercase letter **A** to a **char** variable **letter:**

```
char letter;
letter = 'A';
```

A **char** constant is designated by single quotes. We will study the **char** data type in Chapter 9 on string processing.

Quick CHECK

1. Why are the following declarations all invalid (color highlighting is disabled)?

```
int       a, b, a;
float     x, int;
float     w, int x;
bigNumber double;
```

2. Assuming the following declarations are executed in sequence, why are the second and third declarations invalid?

```
int       a, b;
int       a;
float     b;
```

3. Name six data types for numerical values.

4. Which of the following are valid assignment statements (assuming the variables are properly declared)?

```
x      =  12;
12     =  x;
y + y  =  x;
y      =  x + 12;
```

5. Draw the state-of-memory diagram for the following code.

```
Account latteAcct, espressoAcct;

latteAcct    = new Account();
espressoAcct = new Account();
latteAcct    = espressoAcct;
```

3.2 Arithmetic Expressions

An expression involving numerical values such as

```
23 + 45
```

is called an *arithmetic expression,* because it consists of arithmetic operators and operands. An ***arithmetic operator***, such as + in the example, designates numerical computation. Table 3.2 summarizes the arithmetic operators available in Java.

arithmetic operator

Notice how the division operator works in Java. When both numbers are integers, the result is an integer quotient. That is, any fractional part is truncated. Division between two integers is called ***integer division***. When either or both numbers are float or double, the result is a real number. Here are some division examples:

integer division

Division Operation	Result
`23 / 5`	`4`
`23 / 5.0`	`4.6`
`25.0 / 5.0`	`5.0`

The modulo operator returns the remainder of a division. Although real numbers can be used with the modulo operator, the most common use of the modulo operator involves only integers. Here are some examples:

Modulo Operation	Result
`23 % 5`	`3`
`23 % 25`	`23`
`16 % 2`	`0`

Table 3.2 Arithmetic operators

Operation	Java Operator	Example	Value (x = 10, y = 7, z = 2.5)
Addition	+	`x + y`	17
Subtraction	-	`x - y`	3
Multiplication	*	`x * y`	70
Division	/	`x / y`	1
		`x / z`	4.0
Modulo division (remainder)	%	`x % y`	3

The expression 23 % 5 results in 3 because 23 divided by 5 is 4 with remainder 3. Notice that x % y = 0 when y divides x perfectly; for example, 16 % 2 = 0. Also notice that x % y = x when y is larger than x; for example, 23 % 25 = 23.

operand

An ***operand*** in arithmetic expressions can be a constant, a variable, a method call, or another arithmetic expression, possibly surrounded by parentheses. Let's look at examples. In the expression

```
x + 4
```

binary operator

we have one addition operator and two operands—a variable x and a constant 4. The addition operator is called a ***binary operator*** because it operates on two operands. All other arithmetic operators except the minus are also binary. The minus and plus operators can be both binary and unary. A unary operator operates on one operand as in

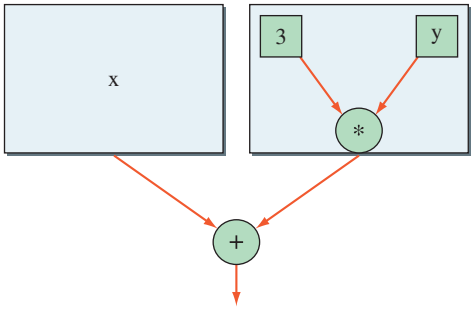
```
-x
```

In the expression

```
x + 3 * y
```

subexpression

the addition operator acts on operands x and 3 * y. The right operand for the addition operator is itself an expression. Often a nested expression is called a ***subexpression***. The subexpression 3 * y has operands 3 and y. The following diagram illustrates this relationship:



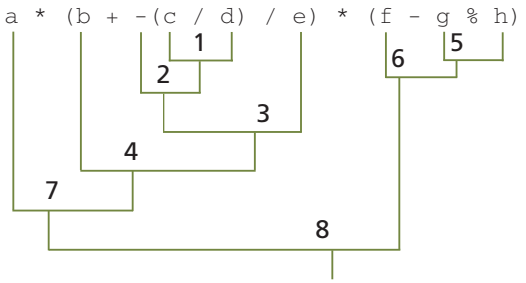
precedence rules

When two or more operators are present in an expression, we determine the order of evaluation by following the ***precedence rules***. For example, multiplication has a higher precedence than addition. Therefore, in the expression x + 3 * y, the multiplication operation is evaluated first, and the addition operation is evaluated next. Table 3.3 summarizes the precedence rules for arithmetic operators.

Table 3.3 **Precedence rules for arithmetic operators and parentheses**

Order	Group	Operator	Rule
High	Subexpression	( )	Subexpressions are evaluated first. If parentheses are nested, the innermost subexpression is evaluated first. If two or more pairs of parentheses are on the same level, then they are evaluated from left to right.
	Unary operator	-, +	Unary minuses and pluses are evaluated second.
	Multiplicative operator	*, /, %	Multiplicative operators are evaluated third. If two or more multiplicative operators are in an expression, then they are evaluated from left to right.
Low	Additive operator	+, -	Additive operators are evaluated last. If two or more additive operators are in an expression, then they are evaluated from left to right.

The following example illustrates the precedence rules applied to a complex arithmetic expression:



When an arithmetic expression consists of variables and constants of the same data type, then the result of the expression will be that data type also. For example, if the data type of a and b is int, then the result of the expression

```
a * b + 23
```

is also an int. When the data types of variables and constants in an arithmetic expression are different data types, then a casting conversion will take place. A *casting conversion,* or ***typecasting***, is a process that converts a value of one data type to another data type. Two types of casting conversions in Java are ***implicit*** and ***explicit***.

implicit and explicit type-casting

Table 3.4 Rules for arithmetic promotion

Operator Type	Promotion Rule
Unary	1. If the operand is of type `byte` or `short`, then it is converted to `int`. 2. Otherwise, the operand remains the same type.
Binary	1. If either operand is of type `double`, then the other operand is converted to `double`. 2. Otherwise, if either operand is of type `float`, then the other operand is converted to `float`. 3. Otherwise, if either operand is of type `long`, then the other operand is converted to `long`. 4. Otherwise, both operands are converted to `int`.

numeric promotion

An implicit conversion called ***numeric promotion*** is applied to the operands of an arithmetic operator. The promotion is based on the rules stated in Table 3.4. This conversion is called *promotion* because the operand is converted from a lower to a higher precision.

typecast operator

Instead of relying on implicit conversion, we can use explicit conversion to convert an operand from one data type to another. Explicit conversion is applied to an operand by using a ***typecast operator***. For example, to convert the int variable x in the expression

```
x / 3
```

to float so the result will not be truncated, we apply the typecast operator (float) as

```
(float) x / 3
```

The syntax is

typecasting syntax

```
( <data type> ) <expression>
```

The typecast operator is a unary operator and has a precedence higher than that of any binary operator. You must use parentheses to typecast a subexpression; for example, the expression

```
a + (double) (x + y * z)
```

will result in the subexpression x + y * z typecast to double.

Assuming the variable x is an int, then the assignment statement

```
x = 2 * (14343 / 2344);
```

will assign the integer result of the expression to the variable x. However, if the data type of x is other than int, then an implicit conversion will occur so that the

assignment conversion

data type of the expression becomes the same as the data type of the variable. An ***assignment conversion*** is another implicit conversion that occurs when the variable and the value of an expression in an assignment statement are not of the same data type. An assignment conversion occurs only if the data type of the variable has a higher precision than the data type of the expression's value. For example,

```
double number;
number = 25;
```

is valid, but

```
int number;
number = 234.56;
```

is not.

In writing programs, we often have to increment or decrement the value of a variable by a certain amount. For example, to increase the value of sum by 5, we write

```
sum = sum + 5;
```

shorthand assignment operator

We can rewrite this statement witout repeating the same variable on the left- and right-hand sides of the assignment symbol by using the ***shorthand assignment operator***:

```
sum += number;
```

Table 3.5 lists five shorthand assignment operators available in Java.
These shorthand assignment operators have precedence lower than that of any other arithmetic operators; so, for example, the statement

```
sum *= a + b;
```

is equivalent to

```
sum = sum * (a + b);
```

Table 3.5 Shorthand assignment operators

Operator	Usage	Meaning
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

Take my Advice

If we wish to assign a value to multiple variables, we can cascade the assignment operations as

```
x = y = 1;
```

which is equivalent to saying

```
y = 1;
x = 1;
```

The assignment symbol = is actually an operator, and its precedence order is lower than that of any other operators. Assignment operators are evaluated right to left.

Quick CHECK

1. Evaluate the following expressions.

a. `3 + 5 / 7`
b. `3 * 3 + 3 % 2`
c. `3 + 2 / 5 + -2 * 4`
d. `2 * (1 + -(3/4) / 2) * (2 - 6 % 3)`

2. What is the data type of the result of the following expressions?

a. `(3 + 5) / 7`
b. `(3 + 5) / (float) 7`
c. `(float) ( (3 + 5) / 7 )`

3. Which of the following expressions is equivalent to $-b(c + 34)/(2a)$?

a. `-b * (c + 34) / 2 * a`
b. `-b * (c + 34) / (2 * a)`
c. `-b * c + 34 / (2 * a)`

4. Rewrite the following statements without using the shorthand operators.

a. `x += y;`
b. `x *= v + w;`
c. `x /= y;`

3.3 Constants

constant

While a program is running, different values may be assigned to a variable at different times (thus the name *variable,* since the values it contains can *vary*), but in some cases we do not want this to happen. In other words, we want to "lock" the assigned value so that no changes can take place. If we want a value to remain fixed, then we use a ***constant***. A constant is declared in a manner similar to a variable but

with the additional reserved word final. A constant must be assigned a value at the time of its declaration. Here's an example of declaring four constants:

```
final double  PI = 3.14159;
final short   FARADAY_CONSTANT = 23060; // unit is cal/volt
final double  CM_PER_INCH = 2.54;
final int     MONTHS_IN_YEAR = 12;
```

We follow the standard Java convention to name a constant, using only capital letters and underscores. Judicious use of constants makes programs more readable. You will be seeing many uses of constants later in the book, beginning with the sample program in this chapter.

named constant

literal constant

The constant PI is called a ***named constant*** or *symbolic constant*. We refer to symbolic constants with identifiers such as PI and FARADAY_CONSTANT. The second type of constant is called a ***literal constant***, and we refer to it by using an actual value. For example, the following statements contain three literal constants:

```
final double   PI   = 3.14159 ;
double area;
area = 2 * PI * 345.79 ;
```

Literal constants

When we use the literal constant 2, the data type of the constant is set to int by default. Then how can we specify a literal constant of type long?[1] We append the constant with an l (a lowercase letter L) or L as in

```
2L * PI * 345.79
```

How about the literal constant 345.79? Since the literal constant contains a decimal point, its data type can be only float or double. But which one? The answer is double. If a literal constant contains a decimal point, then it is of type double by default. To designate a literal constant of type float, we must append the letter f or F. For example,

```
2 * PI * 345.79F
```

To represent a double literal constant, we may optionally append a d or D. So the following two constants are equivalent:

```
2 * PI * 345.79    is equivalent to    2 * PI * 345.79D
```

We also can express float and double literal constants in scientific notation as

$$\text{Number} \times 10^{\text{exponent}}$$

[1] In most cases, it is not significant to distinguish the two because of automatic type conversion; see Section 3.2.

exponential notation in Java

which in Java is expressed as

```
<number> E <exponent>
```

Take my Advice

Since a numerical constant such as 345.79 represents a **double** value, these statements

```
float number;
number = 345.79;
```

for example, would result in a compilation error. The data types do not match, and the variable (**float**) has lower precision than that of the constant (**double**). To correct this error, we have to write the assignment statement as

```
number = 345.79f;
```

or

```
number = (float) 345.79;
```

This is one of the common errors that people make in writing Java programs, especially those with prior programming experience.

where <number> is a literal constant that may or may not contain a decimal point and <exponent> is a signed or an unsigned integer. Lowercase e may be substituted for the exponent symbol E. The whole expression may be suffixed by f, F, d, or D. The <number> itself cannot be suffixed with symbols f, F, d, or D. Here are some examples:

```
12.40e+209
23E33
29.0098e-102
234e+5D
4.45e2
```

Here are some additional examples of constant declarations:

```
final double SPEED_OF_LIGHT = 3.0E+10D; // unit is cm/s
final short  MAX_WGT_ALLOWED = 400;
```

3.4 Getting Numerical Input Values

In Chapter 2, we introduced the use of the showInputDialog method to get the string input. So it is natural for us to consider writing the following statements to get a numerical input value:

Bad Version

```
int age;
age = JOptionPane.showInputDialog(null, "Enter Age:");
```

type mismatch

This code will not work because of ***type mismatch***. Notice that an assignment such as

```
String num = 14;
```

or

```
int num = "14";
```


is invalid because of type mismatch. We cannot assign a value of incompatible type to a variable of another type. Incompatible types mean the values of one type are represented differently in computer memory from the values of the other type. The literal constant "14", for example, is a String object and represented in computer memory differently from the integer constant 14. This difference in representation makes the assignments such as those just shown invalid.

The showInputDialog method returns a String object, and therefore, we cannot assign it directly to a variable of numerical data type. To input a numerical value, we have to first input a String object and then convert it to a numerical representation. We call such operation a ***type conversion***.

type conversion

To perform the necessary type conversion in Java, we use different utility classes called ***wrapper classes***. The name *wrapper* derives from the fact that these classes surround, or wrap, primitive data with useful methods. Type conversion is one of several methods provided by these wrapper classes. To convert string data to integer data, we use the parseInt class method of the Integer class. For example, to convert a string "14" to an int value 14, we write

wrapper classes

Integer

```
int num = Integer.parseInt("14");
```

To input an integer value, say, age, we can write the code as

```
String str
   = JOptionPane.showInputDialog(null, "Enter age:");

int age = Integer.parseInt(str);
```

If the user enters a string that cannot be converted to an int, for example, 12.34 or abc123, an error will result. Table 3.6 lists common wrapper classes and their corresponding conversion method.

Let's write a short program that inputs the radius of a circle and computes the circle's area and circumference. Here's the program:

```
/*
    Chapter 3 Sample Program: Compute Area and Circumference

    File: Ch3Circle.java
*/
```

```
import javax.swing.*;

class Ch3Circle {

    public static void main( String[] args ) {

        final  double PI = 3.14159;

        String radiusStr;
        double radius, area, circumference;

        radiusStr = JOptionPane.showInputDialog(null, "Enter radius:");

        radius = Double.parseDouble(radiusStr);

        //compute area and circumference
        area          = PI * radius * radius;
        circumference = 2.0 * PI * radius;

        JOptionPane.showMessageDialog(null,
                                      "Given Radius: " + radius + "\n"
                                    + "Area: " + area+ "\n"
                                    "Circumference: " + circumference);
    }
}
```

We are allowed to concatenate **String** and numerical data.

the + symbol means addition or concatenation

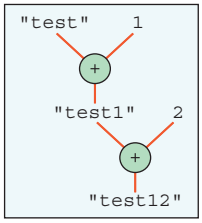
Notice the long expression we pass as the second argument for the showMessageDialog method. In Chapter 2, we use the plus symbol to concatenate strings. We can use the same symbol to concatenate strings and numerical data. Numerical data are automatically converted to string representation and then concatenated.

Table 3.6 Common wrapper classes and their conversion method

Class	Method	Example
Integer	parseInt	Integer.parseInt("25") → 25 Integer.parseInt("25.3") → error
Long	parseLong	Long.parseLong("25") → 25L Long.parseLong("25.3") → error
Float	parseFloat	Float.parseFloat("25.3") → 25.3F Float.parseFloat("ab3") → error
Double	parseDouble	Double.parseDouble("25") → 25.0 Double.parseDouble("ab3") → error

operator overloading

We learned in this chapter that the plus symbol is used for arithmetic addition also. When a symbol is used to represent more than one operation, this is called ***operator overloading***. When the Java compiler encounters an overloaded operator, how does it know which operation the symbol represents? The Java compiler determines the meaning of a symbol by its context. If the left and the right operands of the plus symbol are numerical values, then the compiler will treat the symbol as addition. Otherwise, it will treat the symbol as concatenation. The plus symbol operator is evaluated from left to right, and the result of concatenation is text, so the code

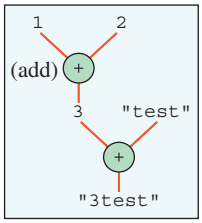



```
int x = 1;
int y = 2;
String output = "test" + x + y ;
```

will result in output being set to

```
test12
```

while the statement




```
String output = x + y + "test" ;
```

will result in output being set to

```
3test
```

To get the result of test3, we have to write the statement as

```
String output = "test" + (x + y);
```

so the arithmetic addition is performed first.

Let's get back to the program. When we run the program and enter 2.35, the dialog shown in Figure 3.4 appears on the screen. Notice the precision of decimal places displayed for the results, especially the one for the circumference. We can

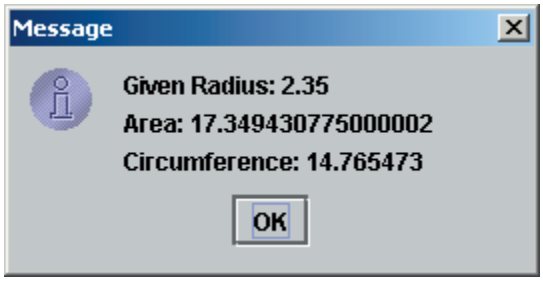


Figure 3.4 The dialog that appears when the input value 2.35 is entered into the **Ch3Circle** program.

DecimalFormat

specify the number of decimal places to display by using the DecimalFormat class from the java.text package. The usage is similar to the one for the SimpleDateFormat class.

Although the full use of the DecimalFormat class can be fairly complicated, it is very straightforward if all we want is to limit the number of decimal places to be displayed. To limit the decimal places to three, we create a DecimalFormat object as

```
DecimalFormat df = new DecimalFormat("0.000");
```

and use it to format the number as

```
double num = 234.5698709;

JOptionPane.showMessageDialog("Num: " + df.format(num));
```

When we add an instance of the DecimalFormat class named df and change the output statement of the Ch3Circle class to

```
JOptionPane.showMessageDialog(null,
                              "GivenRadius: " + radius + "\n"
                              +"Area: "+df.format(area)+"\n"
                              +"Circumference: "
                              + df.format(circumference));
```

the dialog shown in Figure 3.5 appears on the screen. The modified class is named Ch3Circle2.

1. Write a code fragment to input the user's height in inches and assign the value to an int variable named height.
2. Write a code fragment to input the user's weight in pounds and display the weight in kilograms. 1 lb = 453.592 g.

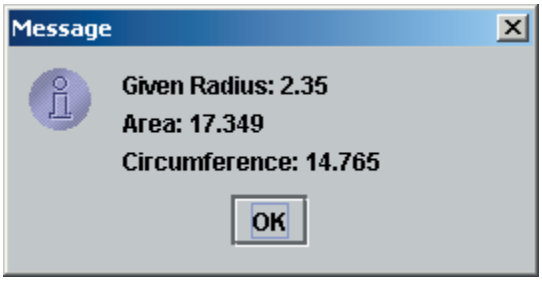


Figure 3.5 The result of formatting the output values by using a **DecimalFormat** object.

3.5 Standard Output

The showMessageDialog method of the JOptionPane class can be used to output multiple lines of text by separating the lines with the special control characters \n. However, the use of showMessageDialog becomes cumbersome and inconvenient when we have to output many lines of text. The showMessageDialog method is intended for displaying short one-line messages, not for a general-purpose output mechanism.

Among different approaches, we will introduce the simplest technique to display multiple lines of text in this section. When we execute the earlier sample programs, we see a window with black background—something similar to one shown in Figure 3.6 appears on the screen. This window is called the ***standard output window***, and we can output multiple lines of text (we can output any numerical values by converting them to text) to this window via System.out. The actual appearance of this standard output window will differ depending on which Java development tool we use. Despite the difference in the actual appearance, its functionality of displaying multiple lines of text is the same among different Java tools.

standard output window

System.out

Helpful Reminder

System.out *refers to a precreated* ***PrintStream*** *object we use to output multiple lines of text to the standard output window. The actual appearance of the standard output window depends on which Java tool we use.*

The System class includes a number of useful class data values. One of them is an instance of the PrintStream class named out. Since this is a class data value, we refer to it through the class name as System.out, and this PrintStream object is tied

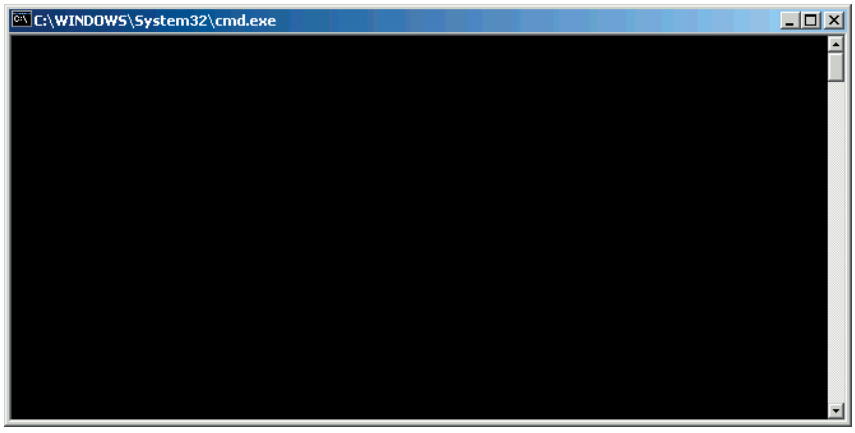


Figure 3.6 The standard output window for displaying multiple lines of text. We can output text to this window via the **System.out** object.

standard output

to the standard output window. Every text we send to System.out will appear on the standard output window. We call the technique to output data by using System.out the ***standard output***.

We use the print method to output a value. For example, executing the code

```
System.out.print("Hello, Dr. Caffeine.");
```

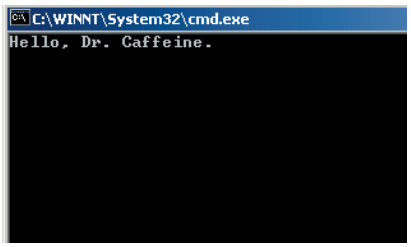
will result in the standard output window shown in Figure 3.7.

The print method will continue printing from the end of the currently displayed output. Executing the following statements after the preceding print message will result in the standard output window shown in Figure 3.8.

```
int x, y;
x = 123;
y = x + x;
System.out.print(" x = ");
System.out.print( x );
System.out.print(" x + x = ");
System.out.print( y );
System.out.print(" THE END");
```

Notice that in the statement

```
System.out.print( x );
```



Note
Depending on the tool you use, you may see additional text such as
Press any key to continue...
or something similar to it. We will ignore any text that may be displayed automatically by the system.

Figure 3.7 Result of executing **System.out.print("Hello, Dr. Caffeine.").**

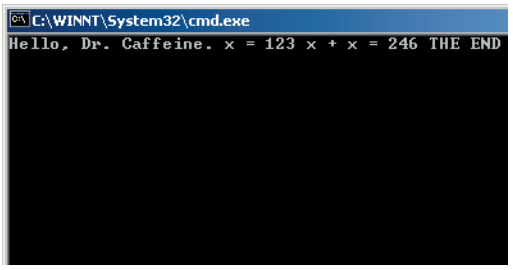


Figure 3.8 Result of sending five **print** messages to **System.out** of Figure 3.7.

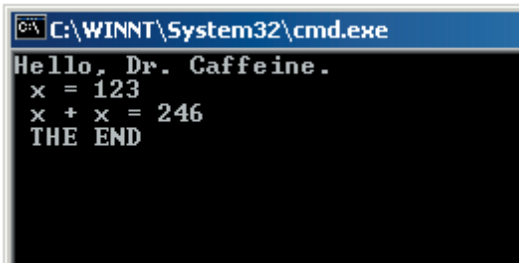


Figure 3.9 Result of mixing **print** with four **println** messages to **System.out.**

we are sending a numerical value as the parameter. But we stated earlier that we use the standard output window for displaying multiple lines of text, that is, string data. Actually, the statement

```
System.out.print( x );
```

is equivalent to

```
System.out.print( Integer.toString(x) );
```

because the print method will do the necessary type conversion if we pass numerical data. We can pass any primitive data value as the parameter to the print method.

We can print an argument and skip to the next line so that subsequent output will start on the next line by using println instead of print. The standard output window of Figure 3.9 will result if we use println instead of print in the preceding example, that is,

```
int x, y;
x = 123;
y = x + x;
System.out.println("Hello, Dr. Caffeine.");
System.out.print(" x = ");
System.out.println( x );
System.out.print(" x + x = ");
System.out.println( y );
System.out.println(" THE END");
```

Let's rewrite the Ch3Circle2 class by using the standard output for displaying the results. Here's the modified program:

```
/*
    Chapter 3 Sample Program: Compute Area and Circumference
                              with formatting using standard
                              output
```

```
    File: Ch2Circle3.java
*/

import javax.swing.*;
import java.text.*;

class Ch3Circle3 {

    public static void main( String[] args )  {

        final double PI = 3.14159;

        String radiusStr;
        double radius, area, circumference;

        DecimalFormat df = new DecimalFormat("0.000");

        //Get input
        radiusStr = JOptionPane.showInputDialog(null, "Enter radius:");
        radius    = Double.parseDouble(radiusStr);

        //Compute area and circumference
        area          = PI * radius * radius;
        circumference = 2.0 * PI * radius;

        //Display the results
        System.out.println("");
        System.out.println("Given Radius: " + radius);
        System.out.println("Area: " + df.format(area));
        System.out.println("Circumference: " + df.format(circumference));
    }
}
```

The output statements can be written by using only one println method as

```
System.out.println("\nGiven Radius: " + radius + "\n"
                   + "Area: " + df.format(area)+ "\n"
                   + "Circumference: "
                   + df.format(circumference));
```

It was necessary to pass all information at once when you are using the showMessageDialog method, but for System.out, it is typical to use one println method for each line of output.

1. Using the standard output, write a Java statement to display a message dialog with the text I Love Java.
2. Using the standard output, write statements to display the following shopping list:

```
Shopping List:
        Apple
        Banana
        Lowfat Milk
```

3.6 Standard Input

As an alternative of using JOptionPane for a simple input routine, we will introduce an approach that works nicely with System.out. Some people may prefer this approach because of the tighter integration with System.out.

System.in

standard input

console input

Analogous to System.out for output, we have System.in for input. We call the technique to input data using System.in ***standard input.*** We also use the term ***console input*** to refer to standard input. Using System.in for input is slightly more complicated than using System.out for output. System.in is an instance of the InputStream class that provides only a facility to input 1 byte at a time with its read method. However, multiple bytes are required to represent a primitive data type or a string (for example, 4 bytes is used to store an int value). So to input primitive data values or strings easily, we need an input facility to process multiple bytes as a single unit.

Scanner

The most common way of using System.in for input is to associate a Scanner object to the System.in object. A Scanner object will allow us to read primitive data type values and strings. The Scanner class is located in the java.util package. First we create a new Scanner object by passing System.in as an argument:

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

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

Once we have a Scanner object, then we can call its nextInt method to input an int. Here's how we input a person's age:

```
int age;
age = scanner.nextInt();
```

If we enter a value that cannot be converted to an int, for example, abc13, then an error will result. When we call the showInputDialog method of JOptionPane, an input dialog appears on the screen which serves as an visual cue that the program is

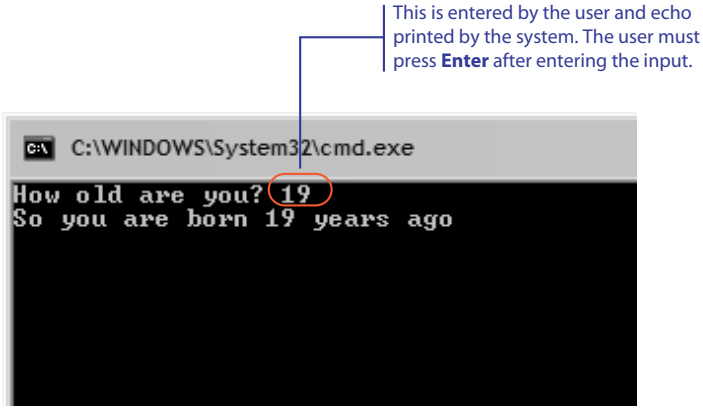


Figure 3.10 Sample interaction using **System.in** and **System.out.**

waiting for the user to input data. There's no such indicator when we use the nextInt method, so we need to prompt the user via System.out. For example,

```
int age;
System.out.print("Enter age: ");
age = scanner.nextInt();
```

echo printing

The characters entered by the user are displayed in the standard output window as they are typed in, so the user can see what's been entered. Printing out the values just entered is called ***echo printing.*** The input line is not processed until the Enter (or Return) key is pressed, so we can erase the characters by pressing the Backspace key while entering the line. Figure 3.10 shows the standard output window when the following code is executed and the user enters the input 19:

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

int age;

System.out.print("How old are you? ");
age = scanner.nextInt();

System.out.println("So you are born "
                        + age + " years ago");
```

You can enter more than one input value on a single line. The following code reads two integers:

```
Scanner scanner = Scanner.create(System.in);

int num1, num2;

System.out.print("Enter two integers: ");
num1 = scanner.nextInt(); //get the first int
```

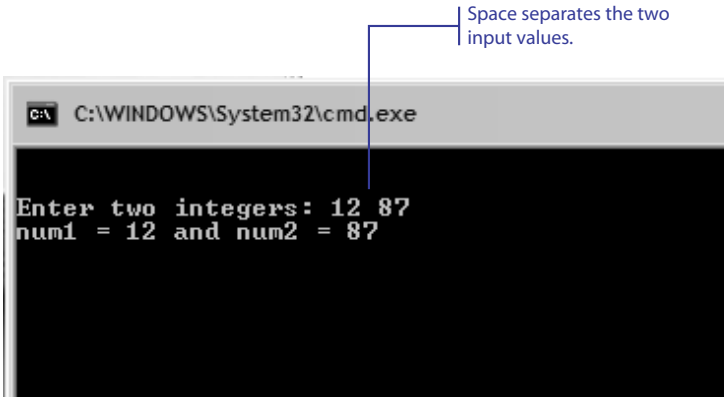


Figure 3.11 Entering two input values on a single line.

```
num2 = scanner.nextInt(); //get the second int

System.out.println("num1 = " + num1 +
                   " and num2 = " + num2);
```

Figure 3.11 shows a sample run. The user must press the Enter key after the second input. Notice that one blank space separates the two input values. Any number of white space characters, such as spaces, tabs, and new lines, can be used to separate the input values.

To input other numerical primitive data types, we use the corresponding methods such as readDouble and readLong. The following code reads a double, an int, and another double:

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

int iNum1;
double dNum1, dNum2;

System.out.print("Enter double, int, double: ");

dNum1 = scanner.nextDouble();
iNum1 = scanner.nextInt();
dNum1 = scanner.nextDouble();
```

Table 3.7 lists six input methods to read numerical data types.

input buffer

When we enter data using System.in, they are placed in an ***input buffer***. And the next available datum in the input buffer is processed when one of the input methods is called. This means that the actual processing of the input data does not necessarily correspond to the display timing of the prompts. Let's look at an example. Consider the following code:

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

int num1, num2, num3;
```

Table 3.7 Methods to input six numerical data types

Method	Example
`nextByte( )`	`byte b = scanner.nextByte( );`
`nextDouble( )`	`double d = scanner.nextDouble( );`
`nextFloat( )`	`float f = scanner.nextFloat( );`
`nextInt( )`	`int i = scanner.nextInt( );`
`nextLong( )`	`long l = scanner.nextLong( );`
`nextShort( )`	`short s = scanner.nextShort( );`

```
System.out.print("Enter Number 1:");
num1 = scanner.nextInt();

System.out.print("Enter Number 2:");
num2 = scanner.nextInt();

System.out.print("Enter Number 3:");
num3 = scanner.nextInt();

System.out.println("Values entered are " +
                   num1 + " " + num2 + " " + num3);
```

For the majority of users, we expect they will input three integers one at a time, as requested by the prompts:

```
Enter Number 1: 14
Enter Number 2: 35
Enter Number 3: 23
Values entered are 14 35 23
```

Values entered by the user are shown in blue.

However, users do not really have to enter the values one at a time. It is possible to enter all three values on a single line without waiting for prompts, for example. Here's one such interaction:

```
Enter Number 1: 14 35 23
Enter Number 2:
Enter Number 3:
Values entered are 14 35 23
```

Values entered by the user are shown in blue.

It works because the three input values are placed in the input buffer, and when the second and third `nextInt` methods are called, the values are in the input buffer, so there's no problem. The critical point is that the appropriate data are available in the input buffer when one of the input methods is called. For example, when

the nextDouble method is called, there will be no problem as long as valid double data are in the input buffer.

The user can enter multiple input values on a single line because the white space is, by default, set as a delimiter between the input values. For instance, when the first nextInt is called, the system will read the string "14" because the space separates it from the second input value. The input string is then converted to an int format. If we want to restrict the users to enter one datum at a time, then we can set the line separator as the delimiter. If the line separator is set as the delimiter, then the user must press the Enter key to separate input values. Because the user must press the Enter key after each input value, she can input only one value on a single input line.

We override the default delimiter by calling the useDelimiter method and pass the appropriate argument. Here's how we set the line separator as the input delimiter:

```
Scanner scanner = Scanner.create(System.in);
String lineSeparator
               = System.getProperty("line.separator");
scanner.useDelimiter(lineSeparator);
```

Since each computer platform could use a different sequence of control characters as the line separator, we use System.getProperty to retrieve the sequence that is specific to the platform which our program is running. For the Windows platform, we can call the useDelimiter method as

```
scanner.useDelimiter("\r\n");
```

But such a statement may not work on other platforms. To make the code general enough to work on all platforms, we use System.getProperty.

Once we override the default delimiter, white space is no longer a delimiter; so if the user enters three values at once, an error will result. Suppose the user enters

```
14 25 45
```

and presses the Enter key. For the given sample code that input three int values, the system accepts the whole string "14 25 45" as the next input value and attempts to convert it into an int, which will result in an error.

Input String

Reading a string input is slightly more complicated than reading numerical data. To input a single word, we use the next method as follows:

```
Scanner scanner = new Scanner(System.in);
String name;
System.out.print("Enter your first name: ");
name = scanner.next( );
```

Because the white space is the default delimiter, we must override it and set the line separator as the delimiter if we want to read a whole line as single input. For example, the following code will accept the whole line as a single string input:

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

String lineSeparator
              = System.getProperty("line.separator");

scanner.useDelimiter(lineSeparator);

String quote;

System.out.print("Enter your favorite quote: ");

quote = scanner.next( );

System.out.println("You entered: " + quote);
```

```
Enter your favorite quote: I think, therefore I am.
You entered: I think, therefore I am.
```

Values entered by the user are shown in blue.

If, however, we do not override the default delimiter and we enter the same input, we will get the following result (only the first word is read by the system):

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

String quote;

System.out.print("Enter your favorite quote: ");

quote = scanner.next( );

System.out.println("You entered: " + quote);
```

```
Enter your favorite quote: I think, therefore I am.
You entered: I
```

Values entered by the user are shown in blue.

If you have more than one input with a mixture of string and primitive data, then the recommended appproach is to set the line separator as the delimiter and input one value per input line.

Helpful Reminder

To input more than one string and primitive data, set the line separator as the delimiter and input one value per input line.

Let's finish this section with a sample program. We will rewrite the Ch3Circle3 class by using the standard input. Here's the program:

```
/*
    Chapter 3 Sample Program: Compute Area and Circumference
                              with formatting using standard
                              input and output

    File: Ch2Circle4.java
*/
import java.util.*;
import java.text.*;

class Ch3Circle4 {

    public static void main( String[] args ) {

        final double PI = 3.14159;

        double radius, area, circumference;
        Scanner scanner;

        DecimalFormat df = new DecimalFormat("0.000");

        scanner = new Scanner(System.in);

        //Get input
        System.out.print("Enter radius: ");

        radius = scanner.nextDouble();

        //Compute area and circumference
        area          = PI * radius * radius;
        circumference = 2.0 * PI * radius;

        //Display the results
        System.out.println("");
        System.out.println("Given Radius: " + radius);
        System.out.println("Area: " + df.format(area));
        System.out.println("Circumference: " + df.format(circumference));
    }
}
```

1. Using the standard input, write a code fragment to input the user's height in inches and assign the value to an int variable named height.
2. Write a code fragment that sets the line separator as the input delimiter.
3. Using the standard input and output, write a code fragment to input the user's weight in pounds and display the weight in kilograms. 1 lb = 453.592 g.

3.7 The Math Class

Using only the arithmetic operators to express numerical computations is very limiting. Many computations require the use of mathematical functions. For example, to express the mathematical formula

$$\frac{1}{2}\sin\left(x - \frac{\pi}{\sqrt{y}}\right)$$

we need the trigonometric sine and square root functions. The Math class in the java.lang package contains class methods for commonly used mathematical functions. Table 3.8 is a partial list of class methods available in the Math class. The class also has two class constants PI and E for π and the natural number e, respectively. Using the Math class constant and methods, we can express the preceding formula as

```
(1.0 /2.0) * Math.sin( x - Math.PI / Math.sqrt(y) )
```

Table 3.8 Math class methods for commonly used mathematical functions

Class Method	Argument Type	Result Type	Description	Example
abs(a)	int	int	Returns the absolute int value of a.	abs(10) → 10 abs(−5) → 5
	long	long	Returns the absolute long value of a.	
	float	float	Returns the absolute float value of a.	
	double	double	Returns the absolute double value of a.	
acos(a)†	double	double	Returns the arccosine of a.	acos(−1) → 3.14159
asin(a)†	double	double	Returns the arcsine of a.	asin(1) → 1.57079
atan(a)†	double	double	Returns the arctangent of a.	atan(1) → 0.785398
ceil(a)	double	double	Returns the smallest whole number greater than or equal to a.	ceil(5.6) → 6.0 ceil(5.0) → 5.0 ceil(−5.6) → −5.0
cos(a)†	double	double	Returns the trigonometric cosine of a.	cos(π/2) → 0.0
exp(a)	double	double	Returns the natural number *e* (2.718 . . .) raised to the power of a.	exp(2) → 7.389056099

Table 3.8 `Math` **class methods for commonly used mathematical functions** *(Continued)*

Class Method	Argument Type	Result Type	Description	Example
`floor( a )`	`double`	`double`	Returns the largest whole number less than or equal to `a`.	floor(5.6) → 5.0 floor(5.0) → 5.0 floor(−5.6) → −6.0
`log( a )`	`double`	`double`	Returns the natural logarithm (base *e*) of `a`.	log(2.7183) → 1.0
`max( a, b )`	`int`	`int`	Returns the larger of `a` and `b`.	max(10, 20) → 20
	`long`	`long`	Same as above.	
	`float`	`float`	Same as above.	
`min(a, b)`	`int`	`int`	Returns the smaller of `a` and `b`.	min(10, 20) → 10
	`long`	`long`	Same as above.	
	`float`	`float`	Same as above.	
`pow(a, b)`	`double`	`double`	Returns the number `a` raised to the power of `b`.	pow(2.0, 3.0) → 8.0
`random( )`	`<none>`	`double`	Generates a random number greater than or equal to 0.0 and less than 1.0.	Examples given in Chapter 5
`round( a )`	`float`	`int`	Returns the `int` value of `a` rounded to the nearest whole number.	round(5.6) → 6 round(5.4) → 5 round(−5.6) → −6
	`double`	`long`	Returns the `float` value of `a` rounded to the nearest whole number.	
`sin( a )`†	`double`	`double`	Returns the trigonometric sine of `a`.	sin(π/2) → 1.0
`sqrt( a )`	`double`	`double`	Returns the square root of `a`.	sqrt(9.0) → 3.0
`tan( a )`†	`double`	`double`	Returns the trigonometric tangent of `a`.	tan(π/4) → 1.0
`toDegrees`	`double`	`double`	Converts the given angle in radians to degrees.	toDegrees(π/4) → 45.0
`toRadians`	`double`	`double`	Reverse of `toDegrees`.	toRadians(90.0) → 1.5707963

†All trigonometric functions are computed in radians.

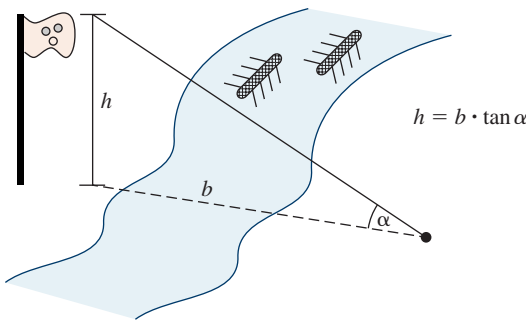
Notice how the class methods and class constants are referred to in the expression. The syntax is

```
<class name> . <method name> ( <arguments> )
```

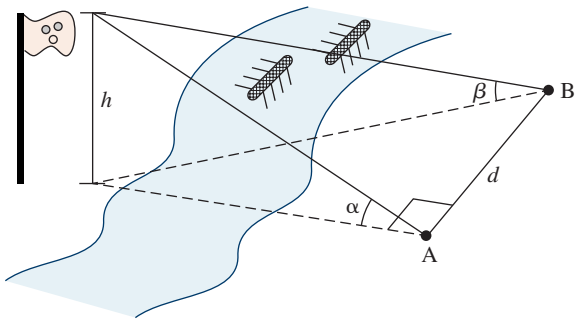
or

```
<class name> . <class constant>
```

Let's conclude this section with a sample program. Today is the final meet of the women's rowing team against the arch rival university before the upcoming Division I NCAA championship. The cheerleaders of the rival team hoisted their school flag on the other shore of the river to boost their moral. Not to be outdone, we want to hoist our school flag, too. To bring the Goddess of Victory to our side, we want our pole to be taller than theirs. Since they won't let us, we can't find the height of their pole by actually measuring it. We can, however, determine the height without actually measuring it if we know the distance b to their flagpole. We can use the tangent of angle to determine the pole's height h as follows:



Unfortunately, there's no means for us to go across the river to find out the distance b. After a moment of deep meditation, it hits us that there's no need to go across the river. We can determine the pole's height by measuring angles from two points on this side of the riverbank, as shown below:



And the equation to compute the height h is

$$h = \frac{d \sin\alpha \sin\beta}{\sqrt{\sin(\alpha + \beta)\sin(\alpha - \beta)}}$$

Once we have this equation, all that's left is to put together a Java program. Here's the program:

```
/*
    Chapter 3 Sample Program: Estimate the Pole Height

    File: Ch3PoleHeight.java

*/

import java.text.*;
import java.util.*;

class Ch3PoleHeight {

    public static void main( String[] args ) {

        double height;          //height of the pole
        double distance;        //distance between points A and B
        double alpha;           //angle measured at point A
        double beta;            //angle measured at point B
        double alphaRad;        //angle alpha in radians
        double betaRad;         //angle beta in radians

        Scanner scanner = new Scanner(System.in);
        scanner.useDelimiter(System.getProperty("line.separator"));

        //Get three input values
        System.out.print("Angle alpha (in degrees):");
        alpha = scanner.nextDouble();

        System.out.print("Angle beta (in degree):");
        beta  = scanner.nextDouble();

        System.out.print("Distance between points A and B (ft):");
        distance   = scanner.nextDouble();

        //compute the height of the tower
        alphaRad = Math.toRadians(alpha);
        betaRad  = Math.toRadians(beta);

        height = ( distance * Math.sin(alphaRad) * Math.sin(betaRad) )
                        /
                    Math.sqrt( Math.sin(alphaRad + betaRad) *
                               Math.sin(alphaRad - betaRad) );
```

```
        DecimalFormat df = new DecimalFormat("0.000");

        System.out.println("Estimating the height of the pole"
            + "\n\n"
            + "Angle at point A (deg): "         + df.format(alpha)    + "\n"
            + "Angle at point B (deg): "         + df.format(beta)     + "\n"
            + "Distance between A and B (ft): " + df.format(distance)+ "\n"
            + "Estimated height (ft): "     + df.format(height));
    }
}
```

1. What's wrong with the following?

a. `y = (1/2) * Math.sqrt( X );`
b. `y = sqrt(38.0);`
c. `y = Math.exp(2, 3);`
d. `y = math.sqrt( b*b - 4*a*c) / ( 2 * a );`

2. If another programmer writes the following statements, do you suspect any misunderstanding on the part of this programmer? What will be the value of `y`?

a. `y = Math.sin( 360 ) ;`
b. `y = Math.cos( 45 );`

3.8 Random Number Generation

In many computer applications, especially in simulation and games, we need to generate random numbers. For example, to simulate a roll of dice, we can generate an integer between 1 and 6. In this section, we explain how to generate random numbers using the random method of the Math class. (Alternatively, you can use the Random class. We refer you to the Java API documentation for information on this class.)

pseudorandom number generator

The method random is called a ***pseudorandom number generator*** and returns a number (type double) that is greater than or equal to 0.0 but less than 1.0, that is, $0.0 \leq X < 1.0$. The generated number is called a *pseudorandom number* because the number is not truly random. When we call this method repeatedly, eventually the numbers generated will repeat themselves. Therefore, theoretically the generated numbers are not random; but for all practical purposes, they are random enough.

The random numbers we want to generate for most applications are integers. For example, to simulate the draw of a card, we need to generate an integer between 1 and 4 for the suit and an integer between 1 and 13 for the number. Since the number returned from the random method ranges from 0.0 up to but not including 1.0, we need to perform some form of conversion so the converted number will fall in our desired range. Let's assume the range of integer values we want is [min, max]. If X is a

number returned by random, then we can convert it into a number Y such that Y is in the range [min, max] that is, min $\leq Y \leq$ max by applying the following fourmula:

$$Y = \lfloor X \times (\max - \min + 1) \rfloor + \min$$

For many applications, the value for min is 1, so the formula is simplified to

$$Y = \lfloor X \times \max \rfloor + 1$$

Expressing the general formula in Java will result in the following statement:

```
//assume correct values are assigned to 'max' and 'min'
int randomNumber
     = (int) (Math.floor(Math.random() * (max-min+1))
               + min);
```

Notice that we have to typecast the result of Math.floor to int because the data type of the result is double.

Let's write a short program that selects a winner among the party goers of the annual spring fraternity dance. The party goers will receive numbers $M + 1$, $M + 2$, $M + 3$, and so on, as they enter the house. The starting value M is selected by the chairperson of the party committee. The last number assigned is $M + N$ if there are N party goers. At the end of the party, we run the program that will randomly select the winning number from the range of $M + 1$ and $M + N$. Here's the program:

```
/*
    Chapter 3 Sample Program: Select the Winning Number

    File: Ch3SelectWinner.java

*/

import java.util.*;

class Ch3SelectWinner {

    public static void main( String[] args ) {

        int startingNumber;  //the starting number
        int count;           //the number of party goers
        int winningNumber;   //the winner
        int min, max;        //the range of random numbers to generate

        Scanner scan = new Scanner(System.in);

        //Get two input values
        System.out.print("Enter the starting number M:     ");
        startingNumber = scan.nextInt();

        System.out.print("Enter the number of party goers: ");
        count  = scan.nextInt();
```

```
        //select the winner
        min = startingNumber + 1;
        max = startingNumber + count;

        winningNumber  = (int) (Math.floor(Math.random() * (max - min + 1))
                                                + min);

        System.out.println("\nThe Winning Number is " + winningNumber);

    }
}
```

3.9 The GregorianCalendar Class

In Chapter 2, we introduced the java.util.Date class to represent a specific instant in time. Notice that we are using here the more concise expression "the java.util.Date class" to refer to a class from a specific package instead of the longer expression "the Date class from the java.util package." This shorter version is our preferred way of notation when we need or want to identify the package to which the class belongs.

Helpful Reminder

*When we need to identify the specific package to which a class belongs, we will commonly use the concise expression with the full path name, such as **java.util.Date**, instead of writing "the **Date** class from the **java.util** package."*

Gregorian-Calendar

In addition to this class, we have a very useful class named java.util.GregorianCalendar in manipulating calendar information such as year, month, and day. We can create a new GregorianCalendar object that represents today as

```
GregorianCalendar today = new GregorianCalendar( );
```

or a specific day, say, July 4, 1776, by passing year, month, and day as the parameters as

The value of 6 means July.

```
GregorianCalendar independenceDay =
            new GregorianCalendar(1776, 6, 4);
```

No, the value of 6 as the second parameter is not an error. The first month of a year, January, is represented by 0, the second month by 1, and so forth. To avoid confusion, we can use constants defined for months in the superclass Calendar (GregorianCalendar is a subclass of Calendar). Instead of remembering that the

value 6 represents July, we can use the defined constant Calendar.JULY as

```
GregorianCalendar independenceDay =
        new GregorianCalendar(1776, Calendar.JULY, 4);
```

Table 3.9 explains the use of some of the more common constants defined in the Calendar class.

When the date and time are November 11, 2002, 6:13 p.m. and we run the Ch3TestCalendar program, we will see the result shown in Figure 3.12.

Table 3.9 Constants defined in the `Calendar` class for retrieved different pieces of calendar/time information

Constant	Description
YEAR	The year portion of the calendar date
MONTH	The month portion of the calendar date
DATE	The day of the month
DAY_OF_MONTH	Same as DATE
DAY_OF_YEAR	The day number within the year
DAY_OF_MONTH	The day number within the month
DAY_OF_WEEK	The day of the week (Sun — 1, Mon — 2, etc.)
WEEK_OF_YEAR	The week number within the year
WEEK_OF_MONTH	The week number within the month
AM_PM	The indicator for AM or PM (AM — 0 and PM — 1)
HOUR	The hour in 12-hour notation
HOUR_OF_DAY	The hour in 24-hour notation
MINUTE	The minute within the hour

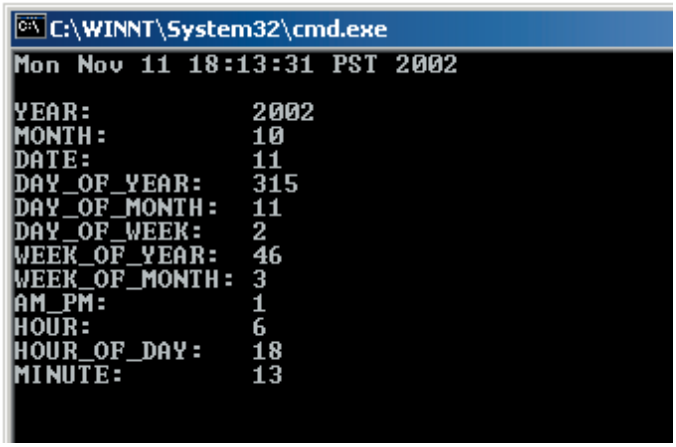


Figure 3.12 Result of running the **Ch3TestCalendar** program at November 11, 2002, 6:13 p.m.

```
/*
    Chapter 3 Sample Program: Display Calendar Info

    File: Ch3TestCalendar.java
*/

import java.util.*;

class Ch3TestCalendar {

    public static void main( String[] args ) {

        GregorianCalendar cal = new GregorianCalendar();

        System.out.println(cal.getTime());
        System.out.println("");

        System.out.println("YEAR:           " + cal.get(Calendar.YEAR));
        System.out.println("MONTH:          " + cal.get(Calendar.MONTH));
        System.out.println("DATE:           " + cal.get(Calendar.DATE));

        System.out.println("DAY_OF_YEAR:    "
                                    + cal.get(Calendar.DAY_OF_YEAR));
        System.out.println("DAY_OF_MONTH:   "
                                    + cal.get(Calendar.DAY_OF_MONTH));
        System.out.println("DAY_OF_WEEK:    "
                                    + cal.get(Calendar.DAY_OF_WEEK));

        System.out.println("WEEK_OF_YEAR:   "
                                    + cal.get(Calendar.WEEK_OF_YEAR));
        System.out.println("WEEK_OF_MONTH: "
                                     + cal.get(Calendar.WEEK_OF_MONTH));

        System.out.println("AM_PM:          " + cal.get(Calendar.AM_PM));
        System.out.println("HOUR:           " + cal.get(Calendar.HOUR));
        System.out.println("HOUR_OF_DAY:    "
                                    + cal.get(Calendar.HOUR_OF_DAY));
        System.out.println("MINUTE:         " + cal.get(Calendar.MINUTE));
    }
}
```

getTime

Notice that the first line in the output shows the full date and time information. The full date and time information can be accessed by calling the calendar object's getTime method. This method returns the same information as a Date object.

Notice also that we get only the numerical values when we retrieve the day of the week or month information. We can spell out the information by using the SimpleDateFormat class. Since the constructor of the SimpleDateFormat class accepts only the Date object, first we need to convert a GregorianCalendar object to

an equivalent Date object by calling its getTime method. For example, here's how we can display the day of the week on which our Declaration of Independence was adopted in Philadelphia:

```
/*
    Chapter 3 Sample Program: Day of the week the Declaration of
                              Independence was adopted

    File: Ch3IndependenceDay.java

*/

import java.util.*;
import java.text.*;
import javax.swing.*;

class Ch3IndependenceDay {

    public static void main( String[] args ) {

        GregorianCalendar independenceDay
           = new GregorianCalendar(1776, Calendar.JULY, 4);

        SimpleDateFormat sdf = new SimpleDateFormat("EEEE");

        JOptionPane.showMessageDialog(null, "It was adopted on "
                        + sdf.format(independenceDay.getTime()));
    }
}
```

Let's finish the section with a sample program that extends the Ch3IndependenceDay program. We will allow the user to enter the year, month, and day; and we will reply with the day of the week of the given date (our birthday, grandparent's wedding day, and so on). We will use the standard input and output for this program. Here's the program:

```
/*
    Chapter 3 Sample Program: Find the Day of Week of a Given Date

    File: Ch3FindDayOfWeek.java
*/

import java.util.*;
import java.text.*;
```

```
class Ch3FindDayOfWeek {

    public static void main( String[] args ) {

        String  inputStr;
        int     year, month, day;

        GregorianCalendar cal;
        SimpleDateFormat  sdf;

        Scanner scanner = new Scanner(System.in);
        scanner.useDelimiter(System.getProperty("line.separator"));

        System.out.print("Year (yyyy): ");
        year      = scanner.nextInt();

        System.out.print("Month (1-12): ");
        month     = scanner.nextInt();

        System.out.print("Day (1-31): ");
        day       = scanner.nextInt();

        cal = new GregorianCalendar(year, month-1, day);
        sdf = new SimpleDateFormat("EEEE");

        System.out.println("");
        System.out.println("Day of Week: " + sdf.format(cal.getTime()));
    }
}
```

Notice that we are allowing the user to enter the month as an integer between 1 and 12, so we need to subtract 1 from the entered data in creating a new GregorianCalendar object.

The Gregorian calendar system was adopted by England and its colonies, including the colonial United States, in 1752. So the technique shown here works only after this adoption. For a fascinating story about calendars, visit **http://webexhibits.org/calendars/year-countries.html**

Running **Ch3IndpendenceDay** will tell you that our venerable document was signed on Thursday. History textbooks will say something like "the document was formally adopted July 4, 1776, on a bright, but cool Philadelphia day" but never the day of the week. Well, now you know. See how useful Java is? By the way, the document was adopted by the Second Continental Congress on July 4, but the actual signing did not take place until August 2 (it was Friday—what a great reason for a TGIF party) after the approval of all 13 colonies. For more stories behind the Declaration of Independence, visit

http://www.archives.gov/exhibit_hall/charters_of_freedom/declaration/declaration.html

or

http://www.ushistory.org/declaration/

3.10 Sample Development

Loan Calculator

In this section, we develop a simple loan calculator program. We develop this program by using an incremental development technique, which develops the program in small incremental steps. We start out with a bare-bones program and gradually build up the program by adding more and more code to it. At each incremental step, we design, code, and test the program before moving on to the next step. This methodical development of a program allows us to focus our attention on a single task at each step, and this reduces the chance of introducing errors into the program.

Problem Statement

The next time you buy a new TV or a stereo, watch out for those "0% down, 0% interest until next July" deals. Read the fine print, and you'll notice that if you don't make the full payment by the end of a certain date, hefty interest will start accruing. You may be better off to get an ordinary loan from the beginning with a cheaper interest rate. What matters most is the total payment (loan amount plus total interest) you'll have to make. To compare different loan deals, let's develop a loan calculator. Here's the problem statement:

> *Write a loan calculator program that computes both monthly and total payments for a given loan amount, annual interest rate, and loan period.*

Overall Plan

Our first task is to map out the overall plan for development. We will identify classes necessary for the program and the steps we will follow to implement the program. We begin with the outline of program logic. For a simple program such as this one, it is kind of obvious; but to practice the incremental development, let's put down the outline of

program flow explicitly. We can express the program flow as having three tasks:

program tasks

1. Get three input values: **loanAmount, interestRate,** and **loanPeriod.**
2. Compute the monthly and total payments.
3. Output the results.

Having identified the three major tasks of the program, we now identify the classes we can use to implement the three tasks. First, we need an object to handle the input of three values. Second, we need an object to display the monthly and total payments. For output, we can use either **JOptionPane** or **System.out.** Since we plan to output multiple lines of text, we will use **System.out.** For input, we can use either **JOptionPane** or **System.in.** If we use **System.in** with **System.out,** then we cannot avoid mixing the echo printing from the input routines with the computation results. We would like to have a clean slate for displaying nicely formatted input values and computation results, so we choose **JOptionPane** for input. Finally, we need to consider how we are going to compute the monthly and total payments. There are no objects in standard packages that will do the computation, so we have to write our own code.

The formula for computing the monthly payment can be found in any mathematics book that covers geometric sequences. It is

$$\text{Monthly payment} = \frac{L \times R}{1 - [1/(1 + R)]^N}$$

where L is the loan amount, R is the monthly interest rate, and N is the number of payments. The monthly rate R is expressed in a fractional value, for example, 0.01 for 1 percent monthly rate. Once the monthly payment is derived, the total payment can be determined by multiplying the monthly payment by the number of months the payment is made. Since the formula includes exponentiation, we will have to use the **pow** method of the **Math** class.

Let's summarize what we have decided so far in a design document:

program classes

Design Document: `LoanCalculator`

Class	Purpose
`LoanCalculator`	The main class of the program.
`JOptionPane`	The `showInputDialog` of the `JOptionPane` class is used to get three input values: loan amount, annual interest rate, and loan period.
`PrintStream (System.out)`	`System.out` is used to display the input values and two computed results: monthly payment and total payment.
`Math`	The `pow` method is used to evaluate exponentiation in the formula for computing the monthly payment. This class is from `java.lang`. *Note:* You don't have to import `java.lang`. The classes in `java.lang` are available to a program without importing.

3.10 Sample Development—*continued*

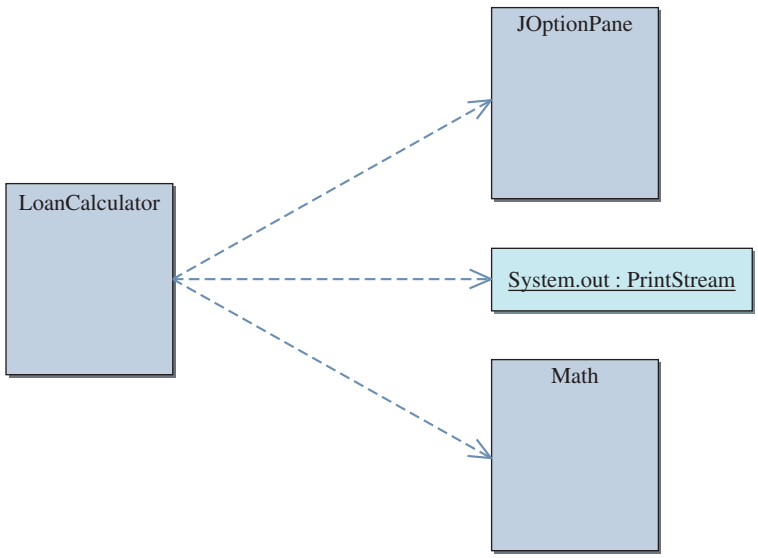


Figure 3.13 The object diagram for the program **LoanCalculator.**

The program diagram based on the classes listed in the design document is shown in Figure 3.13. Keep in mind that this is only a preliminary design. The preliminary document is really a working document that we will modify and expand as we progress through the development steps.

Before we can actually start our development, we must sketch the steps we will follow to implement the program. There is more than one possible sequence of steps to implement a program, and the number of possible sequences will increase as the program becomes more complex. For this program, we will implement the program in four steps:

development steps

1. Start with code to accept three input values.
2. Add code to output the results.
3. Add code to compute the monthly and total payments.
4. Update or modify code and tie up any loose ends.

Notice how the first three steps are ordered. Other orders are possible to develop this program. So why did we choose this particular order? The main reason is our desire to defer the most difficult task until the end. It's possible, but if we implement the computation part in the second incremental step, then we need to code some temporary output routines to verify that the computation is done correctly. However, if we

implement the real output routines before implementing the computation routines, then there is no need for us to worry about temporary output routines. As for step 1 and step 2, their relative order does not matter much. We simply chose to implement the input routine before the output routine because input comes before output in the program.

Step 1 Development: Input Three Data Values

step 1 design

The next task is to determine how we will accept the input values. We will use the **JOptionPane** class we learned in this chapter. We will call the **showInputDialog** method three times to accept three input values: loan amount, annual interest rate, and loan period. The problem statement does not specify the exact format of input, so we will decide that now. Based on how people normally refer to loans, the input values will be accepted in the following format:

Input	Format	Data Type
Loan amount	In dollars and cents (for example, 15000.00)	`double`
Annual interest rate	In percent (for example, 12.5)	`double`
Loan period	In years (for example, 30)	`int`

Be aware that we need to convert the annual interest rate to the monthly interest rate and the input value loan period to the number of monthly payments, to use the given formula. In this case, the conversion is very simple, but even if the conversion routines were more complicated, we must do the conversion. It is not acceptable to ask users to enter an input value that is unnatural to them. For example, people do not think of interest rates in fractional values such as 0.07. They think of interest in terms of percentages such as 7 percent. Computer programs work for humans, not the other way round. Programs we develop should not support an interface that is difficult and awkward for humans to use.

When the user inputs an invalid value, for example, an input string value that cannot be converted to a numerical value or that converts to a negative number, the program should respond accordingly, such as by printing an error message. We do not possess enough skills to implement such a robust program yet, so we will make the following assumptions: (1) The input values are nonnegative numbers, and (2) the loan period is a whole number.

One important objective of this step is to verify that the input values are read in correctly by the program. To verify this, we will use **System.out** to print out the values accepted by **JOptionPane.** Remember that characters entered by the user via **System.in** are echo printed in the standard output window. What we are doing here is another form of echo printing in which our program displays the values entered by the user. Since we are going to use **System.out** in the final program, we will use it for echo printing in this step.

3.10 Sample Development—*continued*

step 1 code

Here's our step 1 program:

```
/*
    Chapter 3 Sample Development: Loan Calculator  (Step 1)

    File: Step1/Ch3LoanCalculator.java

    Step 1: Input Data Values
*/

import javax.swing.*;

class Ch3LoanCalculator {

    public static void main (String[] args) {
        double  loanAmount,
                annualInterestRate;

        int     loanPeriod;

        String  inputStr;

        //get input values
        inputStr           = JOptionPane.showInputDialog(null,
                                 "Loan Amount (Dollars+Cents):");
        loanAmount         = Double.parseDouble(inputStr);

        inputStr           = JOptionPane.showInputDialog(null,
                               "Annual Interest Rate (e.g., 9.5):");
        annualInterestRate = Double.parseDouble(inputStr);

        inputStr           = JOptionPane.showInputDialog(null,
                                  "Loan Period - # of years:");
        loanPeriod         = Integer.parseInt(inputStr);

        //echo print the input values
        System.out.println("Loan Amount:          $" + loanAmount);
        System.out.println("Annual Interest Rate:  "
                                          + annualInterestRate + "%");
        System.out.println("Loan Period (years):  " + loanPeriod);
    }
}
```

step 1 test

To verify the input routine is working correctly, we run the program multiple times and enter different sets of data. We make sure the values are displayed in the standard output window as entered.

Step 2 Development: Output Values

step 2 design

The second step is to add code to display the output values. We will use the standard output window for displaying output values. We need to display the result in a layout that is meaningful and easy to read. Just displaying numbers such as the following is totally unacceptable.

```
132.151.15858.1
```

We must label the output values so the user can tell what the numbers represent. In addition, we must display the input values with the computed result so it will not be meaningless. Which of the two shown in Figure 3.14 do you think is more meaningful? The output format of this program will be

```
For
Loan Amount:              $ <amount>
Annual Interest Rate:       <annual interest rate> %
Loan Period (years):        <year>

Monthly payment is $ <monthly payment>
  TOTAL payment is $ <total payment>
```

with **<amount>, <annual interest rate>,** and others replaced by the actual figures.

Since the computations for the monthly and total payments are not yet implemented, we will use the following dummy assignment statements:

```
monthlyPayment = 135.15;
totalPayment   = 15858.10;
```

We will replace these statements with the real ones in the next step.

Only the computed values (and their labels) are shown.

```
Monthly payment:          $ 143.47
Total payment:            $ 17216.50
```

Both the input and computed values (and their labels) are shown.

```
For
Loan Amount:              $ 10000.00
Annual Interest Rate:       12.0%
Loan Period (years):        10

Monthly payment is        $ 143.47
  TOTAL payment is        $ 17216.50
```

Figure 3.14 Two different display formats, one with input values displayed and the other with only the computed values displayed.

3.10 Sample Development—*continued*

step 2 code

Here's our step 2 program with the newly added portion surrounded by a rectangle and white background:

```
/*
    Chapter 3 Sample Development: Loan Calculator  (Step 2)

    File: Step2/Ch3LoanCalculator.java

    Step 2: Display the Result
*/

import javax.swing.*;

class Ch3LoanCalculator {

    public static void main (String[] args){
        double  loanAmount,
                annualInterestRate;

        double  monthlyPayment,
                totalPayment;

        int     loanPeriod;

        String  inputStr;

        //get input values
        inputStr           = JOptionPane.showInputDialog(null,
                                "Loan Amount (Dollars+Cents):");
        loanAmount         = Double.parseDouble(inputStr);

        inputStr           = JOptionPane.showInputDialog(null,
                                "Annual Interest Rate (e.g., 9.5):");
        annualInterestRate = Double.parseDouble(inputStr);

        inputStr           = JOptionPane.showInputDialog(null,
                                "Loan Period - # of years:");
        loanPeriod         = Integer.parseInt(inputStr);

        //compute the monthly and total payments
        monthlyPayment = 132.15;
        totalPayment   =  15858.10;

        //display the result
        System.out.println("Loan Amount:          $" + loanAmount);
        System.out.println("Annual Interest Rate:  "
                                          + annualInterestRate + "%");
        System.out.println("Loan Period (years):   " + loanPeriod);
```

```
        System.out.println("\n"); //skip two lines
        System.out.println("Monthly payment is $ "    + monthlyPayment);
        System.out.println("  TOTAL payment is $ "    + totalPayment);
    }
}
```

step 2 test

To verify the output routine is working correctly, we run the program and verify the layout. Most likely, we have to run the program several times to fine-tune the arguments for the **printLine** methods until we get the layout that looks clean and nice on the screen.

Step 3 Development: Compute Loan Amount

step 3 design

We are now ready to complete the program by implementing the formula derived in the design phase. The formula requires the monthly interest rate and the number of monthly payments. The input values to the program, however, are the annual interest rate and the loan period in years. So we need to convert the annual interest rate to a monthly interest rate and the loan period to the number of monthly payments. The two input values are converted as

```
monthlyInterestRate = annualInterestRate / 100.0 / MONTHS_IN_YEAR;

numberOfPayments    = loanPeriod * MONTHS_IN_YEAR;
```

where **MONTHS_IN_YEAR** is a symbolic constant with value **12.** Notice that we need to divide the input annual interest rate by 100 first because the formula for loan computation requires that the interest rate be a fractional value, for example, 0.01, but the input annual interest rate is entered as a percentage point, for example, 12.0. Please read Exercise 23 on pages 146–147 for information on how the monthly interest rate is derived from a given annual interest rate.

The formula for computing the monthly and total payments can be expressed as

```
monthlyPayment = (loanAmount * monthlyInterestRate)
                 /
                (1 - Math.pow( 1 /(1 + monthlyInterestRate),
                               numberOfPayments) );

totalPayment = monthlyPayment * numberOfPayments;
```

step 3 code

Let's put in the necessary code for the computations and complete the program. Here's our program:

```
/*
    Chapter 3 Sample Development: Loan Calculator  (Step 3)

    File: Step3/Ch3LoanCalculator.java

    Step 3: Compute the monthly and total payments
*/
```

3.10 Sample Development—*continued*

```
import javax.swing.*;

class Ch3LoanCalculator {

    public static void main (String[] args) {

        final int MONTHS_IN_YEAR = 12;

        double  loanAmount,
                annualInterestRate;

        double  monthlyPayment,
                totalPayment;

        double  monthlyInterestRate;

        int     loanPeriod;

        int     numberOfPayments;

        String  inputStr;

        //get input values
        inputStr           = JOptionPane.showInputDialog(null,
                                 "Loan Amount (Dollars+Cents):");
        loanAmount         = Double.parseDouble(inputStr);

        inputStr           = JOptionPane.showInputDialog(null,
                                 "Annual Interest Rate (e.g., 9.5):");
        annualInterestRate = Double.parseDouble(inputStr);

        inputStr           = JOptionPane.showInputDialog(null,
                                 "Loan Period - # of years:");
        loanPeriod         = Integer.parseInt(inputStr);

        //compute the monthly and total payments
        monthlyInterestRate = annualInterestRate / MONTHS_IN_YEAR / 100;
        numberOfPayments    = loanPeriod * MONTHS_IN_YEAR;

        monthlyPayment = (loanAmount * monthlyInterestRate) /
                         (1 - Math.pow(1/(1 + monthlyInterestRate),
                                       numberOfPayments ) );

        totalPayment  =  monthlyPayment * numberOfPayments;

        //display the result
        System.out.println("Loan Amount:          $" + loanAmount);
        System.out.println("Annual Interest Rate:  "
                                          + annualInterestRate + "%");
```

```
        System.out.println("Loan Period (years):    " + loanPeriod);

        System.out.println("\n"); //skip two lines
        System.out.println("Monthly payment is     $ " + monthlyPayment);
        System.out.println("  TOTAL payment is     $ " + totalPayment);
    }
}
```

step 3 test

After the program is coded, we need to run the program through a number of tests. Since we made the assumption that the input values must be valid, we will test the program only for valid input values. If we don't make that assumption, then we need to test that the program will respond correctly when invalid values are entered. We will perform such testing beginning in Chapter 5. To check that this program produces correct results, we can run the program with the following input values. The right two columns show the correct results. Try other input values as well.

Input			Output (shown up to three decimal places only)	
Loan Amount	**Annual Interest Rate**	**Loan Period (Years)**	**Monthly Payment**	**Total Payment**
10000	10	10	132.151	15858.088
15000	7	15	134.824	24268.363
10000	12	10	143.471	17216.514
0	10	5	0.000	0.000
30	8.5	50	0.216	129.373

Step 4 Development: Finishing Up

step 4 design

We finalize the program in the last step by making any necessary modifications or additions. We will make two additions to the program. The first is necessary while the second is optional but desirable. The first addition is the inclusion of a program description. One of the necessary features of any nontrivial program is the description of what the program does for the user. We will print out a description at the beginning of the program to **System.out.** The second addition is the formatting of the output values. We will format the monthly and total payments to two decimal places, using a **DecimalFormat** object.

3.10 Sample Development—*continued*

step 4 code

Here is our final program:

```
/*
    Chapter 3 Sample Development: Loan Calculator  (Step 4)
    File: Step4/Ch3LoanCalculator.java
    Step 4: Finalize the program
*/
import javax.swing.*;
import java.text.*;
class Ch3LoanCalculator {
    public static void main (String[] args) {
        final int MONTHS_IN_YEAR = 12;
        double  loanAmount,
                annualInterestRate;
        double  monthlyPayment,
                totalPayment;
        double  monthlyInterestRate;
        int     loanPeriod;
        int     numberOfPayments;
        String  inputStr;
        DecimalFormat df = new DecimalFormat("0.00");
    //describe the program
    System.out.println("This program computes the monthly and total");
    System.out.println("payments for a given loan amount, annual ");
    System.out.println("interest rate, and loan period.");
    System.out.println("Loan amount in dollars and cents,
                                                    e.g., 12345.50");
    System.out.println("Annual interest rate in percentage e.g., 12.75");
    System.out.println("Loan period in number of years, e.g., 15");
    System.out.println("\n"); //skip two lines
    //get input values
    inputStr           = JOptionPane.showInputDialog(null,
                               "Loan Amount (Dollars+Cents):");
    loanAmount         = Double.parseDouble(inputStr);
```

```
        inputStr           = JOptionPane.showInputDialog(null,
                                "Annual Interest Rate (e.g., 9.5):");
        annualInterestRate = Double.parseDouble(inputStr);

        inputStr           = JOptionPane.showInputDialog(null,
                                "Loan Period - # of years:");
        loanPeriod         = Integer.parseInt(inputStr);

        //compute the monthly and total payments
        monthlyInterestRate = annualInterestRate / MONTHS_IN_YEAR / 100;
        numberOfPayments    = loanPeriod * MONTHS_IN_YEAR;

        monthlyPayment = (loanAmount * monthlyInterestRate) /
                            (1 - Math.pow(1/(1 + monthlyInterestRate),
                                          numberOfPayments ) );

        totalPayment   =  monthlyPayment * numberOfPayments;

        //display the result
        System.out.println("Loan Amount:          $" + loanAmount);
        System.out.println("Annual Interest Rate:  "
                                        + annualInterestRate + "%");
        System.out.println("Loan Period (years):   " + loanPeriod);

        System.out.println("\n"); //skip two lines

        System.out.println("Monthly payment is    $ "
                                        + df.format(monthlyPayment));
        System.out.println("  TOTAL payment is    $ "
                                        + df.format(totalPayment));
    }
}
```

step 4 test

We repeat the test runs from step 3 and confirm the modified program still runs correctly. Since we have not made any substantial additions or modifications, we fully expect the program to work correctly. However, it is very easy to introduce errors in coding, so even if we think the changes are trivial, we should never skip the testing after even a slight modification.

Helpful Reminder

Always test after making any additions or modifications to a program, no matter how trivial you think the changes are.

3.11 Numerical Representation (*Optional*)

In this section we explain how integers and real numbers are stored in memory. Although computer manufacturers have used various formats for storing numerical values, today's standard is to use the ***twos complement*** format for storing integers and the *floating-point* format for real numbers. We describe these formats in this section.

twos complement

An integer can occupy 1, 2, 4, or 8 bytes depending on which data type (i.e., byte, short, int, or long) is declared. To make the examples easy to follow, we will use 1 byte (= 8 bits) to explain twos complement form. The same principle applies to 2, 4, and 8 bytes. (They just utilize more bits.)

The following table shows the first five and the last four of the 256 positive binary numbers using 8 bits. The right column lists their decimal equivalents.

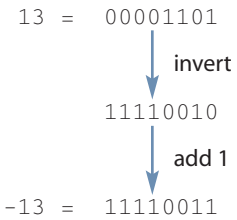
8-Bit Binary Number	Decimal Equivalent
00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
...	
11111100	252
11111101	253
11111110	254
11111111	255

sign bit

Using 8 bits, we can represent positive integers from 0 to 255. Now let's see the possible range of negative and positive numbers that we can represent, using 8 bits. We can designate the leftmost bit as a ***sign bit:*** 0 means positive and 1 means negative. Using this scheme, we can represent integers from −127 to +127 as shown in the following table:

8-Bit Binary Number (with a Sign Bit)	Decimal Equivalent
0 0000000	+0
0 0000001	+1
0 0000010	+2
...	
0 1111111	+127
1 0000000	−0
1 0000001	−1
...	
1 1111110	−126
1 1111111	−127

Notice that zero has two distinct representations (+0 = 00000000 and −0 = 10000000), which adds complexity in hardware design. Twos complement format avoids this problem of duplicate representations for zero. In twos complement format, all positive numbers have zero in their leftmost bit. The representation of a negative number is derived by first inverting all the bits (changing 1s to 0s and 0s to 1s) in the representation of the positive number and then adding 1. The following diagram illustrates the process:



The following table shows the decimal equivalents of 8-bit binary numbers by using twos complement representation. Notice that zero has only one representation.

8-Bit Binary Number (Twos Complement)	Decimal Equivalent
00000000	+0
00000001	+1
00000010	+2
...	
01111111	+127
10000000	−128
10000001	−127
...	
11111110	−2
11111111	−1

floating-point

Now let's see how real numbers are stored in memory in ***floating-point*** format. We present only the basic ideas of storing real numbers in computer memory here. We omit the precise details of the Institute of Electronics and Electrical Engineers (IEEE) Standard 754 that Java uses to store real numbers.

Real numbers are represented in the computer by using scientific notation. In base-10 scientific notation, a real number is expressed as

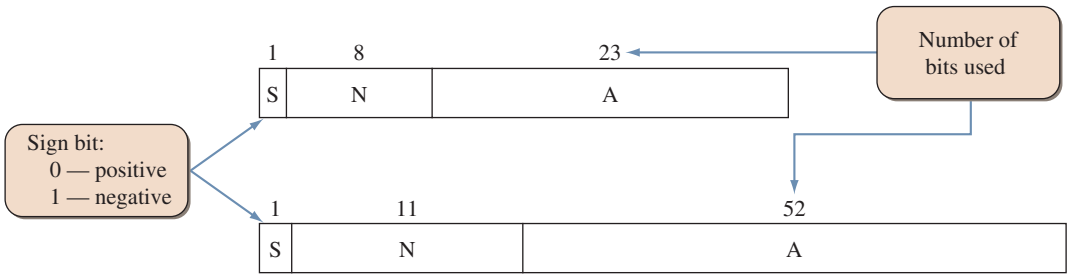
$$A \times 10^N$$

where A is a real number and N is an integral exponent. For example, the mass of a hydrogen atom (in grams) is expressed in decimal scientific notation as 1.67339×10^{-24}, which is equal to 0.00000000000000000000000167339.

We use base-2 scientific notation to store real numbers in computer memory. Base-2 scientific notation represents a real number as follows:

$$A \times 2^N$$

The float and double data types use 32 and 64 bits, respectively, with the number A and exponent N stored as follows:

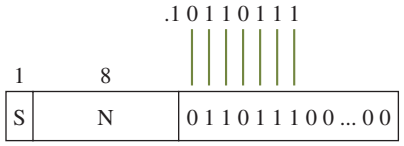


normalized fraction

The value A is a ***normalized fraction***, where the fraction begins with a binary point, followed by a 1 bit and the rest of the fraction. (*Note:* A decimal number has a decimal point; a binary number has a binary point.) The following numbers are sample normalized and unnormalized binary fractions:

Normalized	Unnormalized
1.1010100	1.100111
1.100011	.0000000001
1.101110011	.0001010110

Since a normalized number always start with a 1, this bit does not actually have to be stored. The following diagram illustrates how the A value is stored.



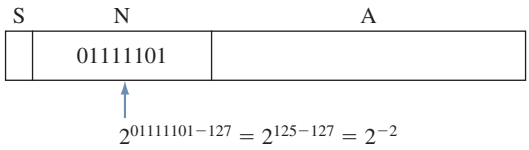
excess format

The sign bit S indicates the sign of a number, so A is stored in memory as an unsigned number. The integral exponent N can be negative or positive. Instead of using twos complement for storing N, we use a format called ***excess format***. The 8-bit exponent uses the excess-127 format, and the 11-bit exponent uses the excess-1023 format. We will explain the excess-127 format here. The excess-1023 works similarly. With the excess-127 format, the actual exponent is computed as

$$N - 127$$

Therefore, the number 127 represents an exponent of zero. Numbers less than 127 represent negative exponents, and numbers greater than 127 represent positive

exponents. The following diagram illustrates that the number 125 in the exponent field represents $2^{125-127} = 2^{-2}$.



Summary

- A variable is a memory location in which to store a value.
- A variable has a name and a data type.
- A variable must be declared before we can assign a value to it.
- There are six numerical data types in Java: byte, short, int, long, float, and double.
- Object names are synonymous with variables whose contents are memory addresses.
- Numerical data types are called primitive data types, and objects are called reference data types.
- Precedence rules determine the order of evaluating arithemetic expressions.
- Symbolic constants hold values just as variables do, but we cannot change their values.
- The standard classes introduced in this chapter are

Math	InputStream
GregorianCalendar	InputStreamReader
DecimalFormat	BufferedReader
PrintStream	IOException
Integer, Double, etc.	

- We use methods from the wrapper classes Integer, Double, and others to convert an input string to a numerical value.
- System.out is used to output multiple lines of text to the standard output window.
- System.in is used to input a stream of bytes. We associate a BufferedReader object to System.in (with an intermediate InputStreamReader) to input one line of text at at time.
- The Math class contains many class methods for mathematical functions.
- The GregorianCalendar class is used in the manipulation of calendar information.
- The DecimalFormat class is used to format numerical data.
- (*Optional*) Twos complement format is used for storing integers, and floating-pointing format is used for storing real numbers.

Key Concepts

variables
primitive data types
reference data types
arithmetic expression
arithmetic operators
precedence rules
typecasting
implicit and explicit casting
assignment conversion
constants
string-to-number conversion
standard output
standard input
echo printing
twos complement (*optional*)
floating point (*optional*)

Exercises

1. Suppose we have the following declarations:

```
int i = 3, j = 4, k = 5;
float x = 34.5f, y = 12.25f;
```

Determine the value for each of the following expressions, or explain why it is not a valid expression.

a. `(x + 1.5) / (250.0 * (i/j))`
b. `x + 1.5 / 250.0 * i / j`
c. `-x * -y * (i + j) / k`
d. `(i / 5) * y`
e. `Math.min(i, Math.min(j,k))`
f. `Math.exp(3, 2)`
g. `y % x`
h. `Math.pow(3, 2)`
i. `(int)y % k`
j. `i / 5 * y`

2. Suppose we have the following declarations:

```
int m, n, i = 3, j = 4, k = 5;
float v, w, x = 34.5f, y = 12.25f;
```

Determine the value assigned to the variable in each of the following assignment statements, or explain why it is not a valid assignment.

a. `w = Math.pow(3,Math.pow(i,j));`
b. `v = x / i;`
c. `w = Math.ceil(y) % k;`
d. `n = (int) x / y * i / 2;`
e. `x = Math.sqrt(i*i - 4*j*k);`
f. `m = n + i * j;`

g. `n = k /(j * i) * x + y;`
h. `i = i + 1;`
i. `w = float(x + i);`
j. `x = x / i / y / j;`

3. Suppose we have the following declarations:

```
int    i, j;
float  x, y;
double u, v;
```

Which of the following assignments are valid?

a. `i = x;`
b. `x = u + y;`
c. `x = 23.4 + j * y;`
d. `v = (int) x;`
e. `y = j / i * x;`

4. Write Java expressions to compute each of the following.

a. The square root of $B^2 + 4AC$ (A and C are distinct variables)
b. The square root of $X + 4Y^3$
c. The cube root of the product of X and Y
d. The area πR^2 of a circle

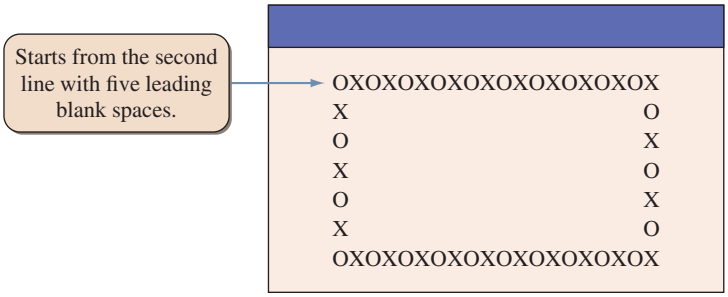
5. Determine the output of the following program without running it.

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

      System.out.println("One");
      System.out.print("Two");
      System.out.print("\n");

      System.out.print("Three");
      System.out.println("Four");
      System.out.print("\n");

      System.out.print("Five");
      System.out.println("Six");
   }
}
```

6. Determine the output of the following code.

```
int x, y;
x = 1;
y = 2;
System.out.println("The output is " + x + y );
System.out.println("The output is " + (x + y) );
```

7. Write an application that displays the following pattern in the standard output window.



Note: The output window is not drawn to scale.

8. Write an application to convert centimeters (input) to feet and inches (output). Use JOptionPane for input and output. 1 in = 2.54 cm.

9. Write an application that inputs temperature in degrees Celsius and prints out the temperature in degrees Fahrenheit. Use System.in for input and System.out for output. The formula to convert degrees Celsius to equivalent degrees Fahrenheit is

$$\text{Fahrenheit} = 1.8 \times \text{Celsius} + 32$$

10. Write an application that accepts a person's weight and displays the number of calories the person needs in one day. A person needs 19 calories per pound of body weight, so the formula expressed in Java is

```
calories = bodyWeight * 19;
```

Use JOptionPane for input and output. (*Note:* We are not distinguishing between genders.)

11. A quantity known as the *body mass index* (BMI) is used to calculate the risk of weight-related health problems. BMI is computed by the formula

$$\text{BMI} = \frac{w}{(h/100.0)^2}$$

where w is weight in kilograms and h is height in centimeters. A BMI of about 20 to 25 is considered "normal." Write an application that accepts weight and height (both integers) and outputs the BMI. Use System.in for input and System.out for output.

12. Your weight is actually the amount of gravitational attraction exerted on you by the Earth. Since the Moon's gravity is only one-sixth of the Earth's gravity, on the Moon you would weigh only one-sixth of what you weigh on Earth. Write an application that inputs the user's Earth weight and outputs

her or his weight on Mercury, Venus, Jupiter, and Saturn. Use your preferred choice for input and output. Use the values in this table.

Planet	Multiply the Earth Weight by
Mercury	0.4
Venus	0.9
Jupiter	2.5
Saturn	1.1

13. When you say you are 18 years old, you are really saying that the Earth has circled the Sun 18 times. Since other planets take fewer or more days than Earth to travel around the Sun, your age would be different on other planets. You can compute how old you are on other planets by the formula

$$y = \frac{x \times 365}{d}$$

where `x` is the age on Earth, `y` is the age on planet `Y`, and `d` is the number of Earth days the planet `Y` takes to travel around the Sun. Write an application that inputs the user's Earth age and print outs his or her age on Mercury, Venus, Jupiter, and Saturn. Use your preferred choice for input and output. The values for `d` are listed in the table.

Planet	d = Approximate Number of Earth Days for This Planet to Travel around the Sun
Mercury	88
Venus	225
Jupiter	4,380
Saturn	10,767

14. Write an application to solve quadratic equations of the form

$$Ax^2 + Bx + C = 0$$

where the coefficients A, B, and C are real numbers. The two real number solutions are derived by the formula

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

For this exercise, you may assume that $A \neq 0$ and the relationship

$$B^2 \geq 4AC$$

holds, so there will be real number solutions for `x`. Use the standard input and output.

15. Write an application that determines the number of days in a given semester. Input to the program is the year, month, and day information of the first and the last days of a semester. *Hint:* Create GregorianCalendar objects for the start and end dates of a semester and manipulate their DAY_OF_YEAR data.

16. Modify the Ch3FindDayOfWeek program by accepting the date information as a single string instead of accepting the year, month, and day information separately. The input string must be in the MM/dd/yyyy format. For example, July 4, 1776, is entered as 07/04/1776. There will be exactly two digits for the month and day and four digits for the year.

17. Write an application that accepts the unit weight of a bag of coffee in pounds and the number of bags sold and displays the total price of the sale, computed as

```
totalPrice        = unitWeight * numberOfUnits * 5.99;
totalPriceWithTax = totalPrice + totalPrice * 0.0725;
```

where 5.99 is the cost per pound and 0.0725 is the sales tax. Display the result in the following manner:

```
Number of bags sold:  32
     Weight per bag:  5 lb
    Price per pound:  $5.99
          Sales tax:  7.25%

        Total price: $ 1027.884
```

Use JOptionPane for input and System.out for output. Draw the program diagram.

18. If you invest *P* dollars at *R* percent interest rate compounded annually, in *N* years, your investment will grow to

$$\frac{P[1-(R/100)^{N+1}]}{1-R/100}$$

dollars. Write an application that accepts P, R, and N and computes the amount of money earned after N years. Use JOptionPane for input and output.

19. Leonardo Fibonacci of Pisa was one of the greatest mathematicians of the Middle Ages. He is perhaps most famous for the Fibonacci sequence, which can be applied to many diverse problems. One amusing application of the Fibonacci sequence is in finding the growth rate of rabbits. Suppose a pair of rabbits matures in 2 months and is capable of reproducing another pair every month after maturity. If every new pair has the same capability, how many pairs will there be after 1 year? (We assume here that no pairs die.) The table

below shows the sequence for the first 7 months. Notice that at the end of the second month, the first pair matures and bears its first offspring in the third month, making the total two pairs.

Month No.	Number of Pairs
1	1
2	1
3	2
4	3
5	5
6	8
7	13

The *N*th Fibonacci number in the sequence can be evaluated with the formula

$$F_N = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^N - \left(\frac{1-\sqrt{5}}{2}\right)^N\right]$$

Write an application that accepts N and displays F_N. Note that the result of computation using the Math class is double. You need to display it as an integer. Use JOptionPane for input and output.

20. According to Newton's universal law of gravitation, the force *F* between two bodies with masses M_1 and M_2 is computed as

$$F = k\left(\frac{M_1 M_2}{d^2}\right)$$

where *d* is the distance between the two bodies and *k* is a positive real number called the *gravitational constant*. The gravitational constant *k* is approximately equal to 6.67E-8 dyn · cm^2/g^2. Write an application that (1) accepts the mass for two bodies in grams and the distance between the two bodies in centimeters and (2) computes the force F. Use the standard input and output, and format the output appropriately. For your information, the force between the Earth and the Moon is 1.984E25 dyn. The mass of the earth is 5.983E27 g, the mass of the moon is 7.347E25 g, and the distance between the two is 3.844E10 cm.

21. Dr. Caffeine's Law of Program Readability states that the degree of program readability *R* (whose unit is *mocha*) is determined as

$$R = k \cdot \frac{CT^2}{V^3}$$

where *k* is Ms. Latte's constant, *C* is the number of lines in the program that contain comments, *T* is the time spent (in minutes) by the programmer developing the program, and *V* is the number of lines in the program that contain nondescriptive variable names. Write an application to compute the program readability R. Use your preferred choice for input and output.

Ms. Latte's constant is `2.5E2` mocha lines^2/min^2. (*Note:* This is just for fun. Develop your own law, using various functions from the `Math` class.)

22. If the population of a country grows according to the formula

$$y = ce^{kx}$$

where *y* is the population after *x* years from the reference year, then we can determine the population of a country for a given year from two census figures. For example, given that a country with a population of 1,000,000 in 1970 grows to 2,000,000 by 1990, we can predict the country's population in the year 2000. Here's how we do the computation. Letting *x* be the number of years after 1970, we obtain the constant *c* as 1,000,000 because

$$1{,}000{,}000 = ce^{k0} = c$$

Then we determine the value of *k* as

$$y = 1{,}000{,}000e^{kx}$$

$$\frac{2{,}000{,}000}{1{,}000{,}000} = e^{20k}$$

$$k = \frac{1}{20}\ln\frac{2{,}000{,}000}{1{,}000{,}000} \approx 0.03466$$

Finally we can predict the population in the year 2000 by substituting `0.03466` for *k* and 30 for *x* (2000 − 1970 = 30). Thus, we predict

$$y = 1{,}000{,}000e^{0.03466(30)} \approx 2{,}828{,}651$$

as the population of the country for the year 2000. Write an application that accepts five input values—year A, population in year A, year B, population in year B, and year C—and predict the population for year C. Use any technique for input and output.

23. In Section 3.10, we use the formula

$$\text{MR} = \frac{\text{AR}}{12}$$

to derive the monthly interest rate from a given annual interest rate, where MR is the monthly interest rate and AR is the annual interest rate (expressed in a fractional value such as 0.083). This annual interest rate AR is called the *stated annual interest rate* to distinguish it from the *effective annual interest rate,* which is the true cost of a loan. If the stated annual interest rate is 9 percent, for example, then the effective annual interest rate is actually 9.38 percent. Naturally, the rate that the financial institutions advertise more prominently is the stated interest rate. The loan calculator program in Section 3.10 treats the annual interest rate that the user enters as the stated

annual interest rate. If the input is the effective annual interest rate, then we compute the monthly rate as

$$MR = (1 + EAR)^{1/12} - 1$$

where EAR is the effective annual interest rate. The difference between the stated and effective annual interest rates is negligible only when the loan amount is small or the loan period is short. Modify the loan calculator program so that the interest rate that the user enters is treated as the effective annual interest rate. Run the original and modified loan calculator programs, and compare the differences in the monthly and total payments. Use loan amounts of 1, 10, and 50 million dollars with loan periods of 10, 20, and 30 years and annual interest rates of 0.07, 0.10, and 0.18 percent, respectively. Try other combinations also.

Visit several websites that provide a loan calculator for computing a monthly mortgage payment (one such site is the financial page at **www.cnn.com**). Compare your results to the values computed by the websites you visited. Determine whether the websites treat the input annual interest rate as stated or effective.

Development Exercises

For the following exercises, use the incremental development methodology to implement the program. For each exercise, identify the program tasks, create a design document with class descriptions, and draw the program diagram. Map out the development steps at the start. State any assumptions you must make about the input. Present any design alternatives and justify your selection. Be sure to perform adequate testing at the end of each development step.

24. Develop an application that reads a purchase price and an amount tendered and then displays the change in dollars, quarters, dimes, nickels, and pennies. Two input values are entered in cents, for example, 3480 for $34.80 and 70 for $0.70. Use any appropriate technique for input and output. Display the output in the following format:

```
 Purchase Price: $ 34.80
Amount Tendered: $ 40.00

 Your change is: $ 5.20

                 5 one-dollar bill(s)
                 0 quarter(s)
                 2 dime(s)
                 0 nickel(s)
                 0 penn(y/ies)

Thank you for your business. Come back soon.
```

Notice the input values are to be entered in cents (int data type), but the echo printed values must be displayed with decimal points (float data type).

25. MyJava Coffee Outlet runs a catalog business. It sells only one type of coffee beans, harvested exclusively in the remote area of Irian Jaya. The company sells the coffee in 2-lb bags only, and the price of a single 2-lb bag is $5.50. When a customer places an order, the company ships the order in boxes. The boxes come in three sizes: the large box holds 20 bags of 2 lb, the medium 10 bags, and the small 5 bags. The cost of a large box is $1.80; a medium box, $1.00; and a small box, $0.60. The order is shipped using the least number of boxes. For example, the order of 52 bags will be shipped in two boxes, one large and one small. Develop an application that computes the total cost of an order. Use any appropriate technique for input and output. Display the output in the following format:

```
Number of Bags Ordered: 52 - $ 286.00

Boxes Used:
            2 Large  - $3.60
            1 Medium - $1.00
            1 Small  - $0.60

Your total cost is: $ 291.20
```

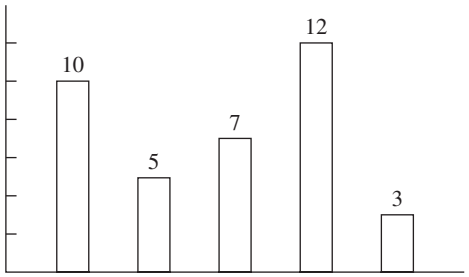
26. Repeat Exercise 25, but this time, accept the date when the order is placed and display the expected date of arrival. The expected date of arrival is two weeks (14 days) from the date of order. The order date is entered as a single string in the MM/dd/yyyy format. For example, November 1, 2004 is entered as 11/01/2004. There will be exactly two digits each for the month and day and four digits for the year. Display the output in the following format:

```
Number of Bags Ordered: 52 - $ 286.00

Boxes Used:
            2 Large  - $3.60
            1 Medium - $1.00
            1 Small  - $0.60

Your total cost is: $ 291.20

Date of Order:             November 1, 2004
Expected Date of Arrival:  November 15, 2004
```

27. Using a Turtle object from the galapagos package (see Exercise 23 on page 147), draw three rectangles. Use JOptionPane to accept the width and the

length of the smallest rectangle from the user. The middle and the largest rectangles are 40 and 80 percent larger, respectively, than the smallest rectangle. The galapagos package and its documentation are available at **www.drcaffeine.com.**

28. Develop a program that draws a bar chart using a Turtle object (see Exercise 23 on page 146). Input five int values, and draw the vertical bars that represent the entered values in the following manner:



Your Turtle must draw everything shown in the diagram, including the axes and numbers. Use any suitable approach for input.