

# Numerical Data

# 3

## O b j e c t i v e s

*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 numerical data by using **System.in** and output numerical 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



# W

hen 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. The `Random` class includes methods to generate different types of random numbers. In Chapter 2, we performed String input and output by using the standard input (`Scanner`) and output (`System.out`). We will describe the input and output routines for numerical data in this chapter.

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 Java 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.

## Things to Remember



*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

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

variable  
declaration  
syntax

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

**Table 3.1** Java numerical data types and their precisions

Data Type	Content	Default Value <sup>†</sup>	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 <sup>‡</sup>	3.40282347E+38
double	Real	0.0	-1.79769313486231570E+308	1.79769313486231570E+308

<sup>†</sup> No default value is assigned to a local variable. A local variable is explained on page 191 in Section 4.8.

<sup>‡</sup> The character E indicates a number is expressed in scientific notation. This notation is explained on page 100.

design. The 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;
```

## Hints, & Tips, Pitfalls



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;
```

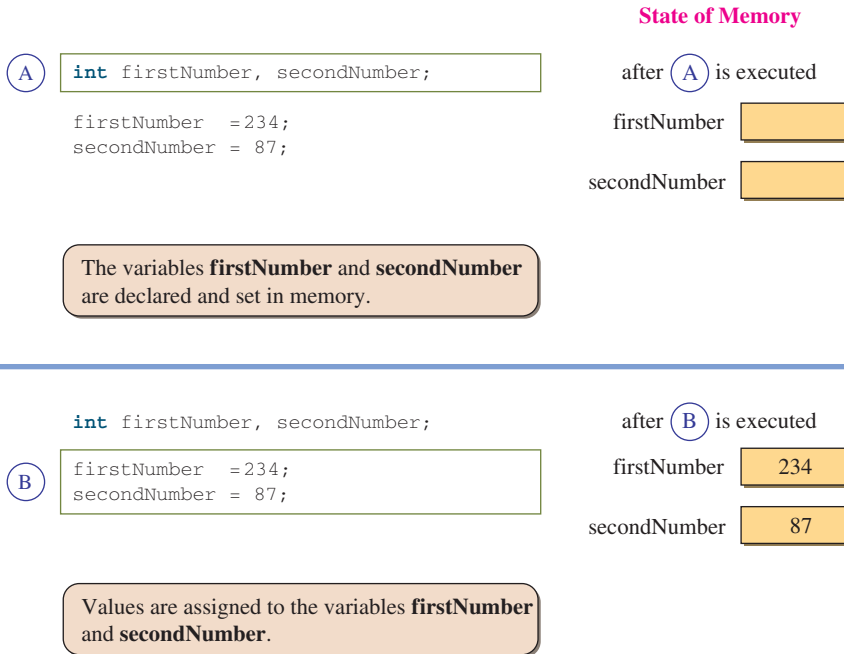
We will present a detailed discussion of arithmetic expressions in Section 3.2. One key point we need to remember about variables is the following:

## Things to Remember



*Before using a variable, we must first 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 36. Figure 3.2 compares the two.



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

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.

## Things to Remember



*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.

## Numerical Data

```
int number;
```

```
number = 237;
number = 35;
```

number



## Object

```
Customer customer;
```

```
customer = new Customer();
customer = new Customer();
```

customer

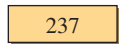


```
int number;
```

```
number = 237;
```

```
number = 35;
```

number



```
Customer customer;
```

```
customer = new Customer();
```

```
customer = new Customer();
```

customer



```
int number;
```

```
number = 237;
```

```
number = 35;
```

number

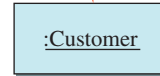


```
Customer customer;
```

```
customer = new Customer();
```

```
customer = new Customer();
```

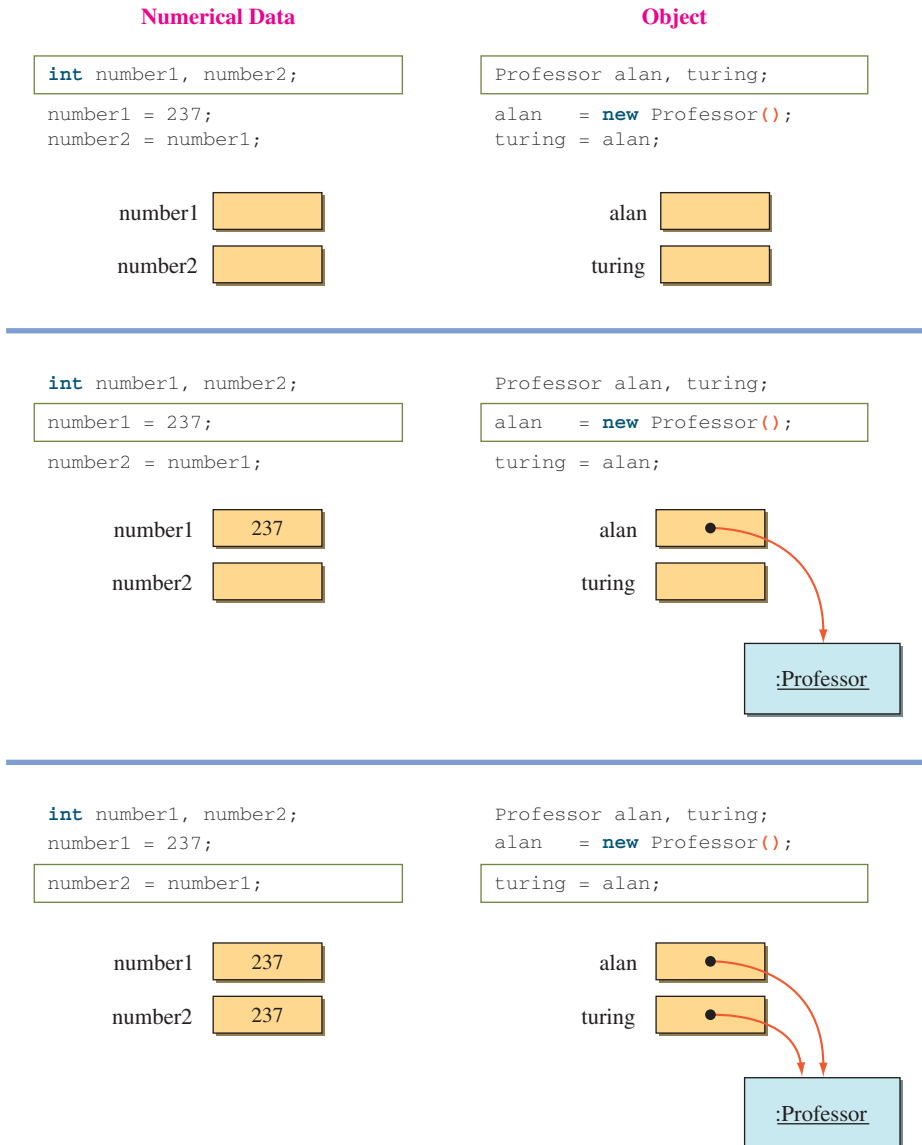
customer



**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*.



## Hints, & Tips, Pitfalls



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

$$23 + 45$$

arithmetic  
operator

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.

integer division

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:

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

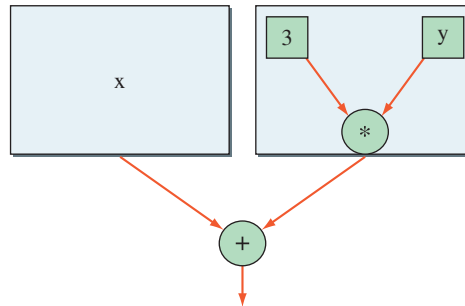
$$-x$$

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:



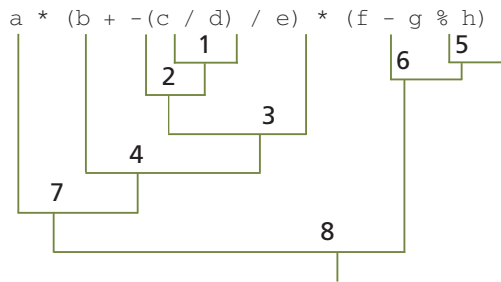
precedence rules

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   Low	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.
	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	<ol style="list-style-type: none"> <li>1. If the operand is of type <code>byte</code> or <code>short</code>, then it is converted to <code>int</code>.</li> <li>2. Otherwise, the operand remains the same type.</li> </ol>
Binary	<ol style="list-style-type: none"> <li>1. If either operand is of type <code>double</code>, then the other operand is converted to <code>double</code>.</li> <li>2. Otherwise, if either operand is of type <code>float</code>, then the other operand is converted to <code>float</code>.</li> <li>3. Otherwise, if either operand is of type <code>long</code>, then the other operand is converted to <code>long</code>.</li> <li>4. Otherwise, both operands are converted to <code>int</code>.</li> </ol>

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.

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

typecast operator

```
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; ← INVALID
```

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 without repeating the same variable on the left- and right-hand sides of the assignment symbol by using the *shorthand assignment operator*:

```
sum += 5;
```

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
<code>+=</code>	<code>a += b;</code>	<code>a = a + b;</code>
<code>-=</code>	<code>a -= b;</code>	<code>a = a - b;</code>
<code>*=</code>	<code>a *= b;</code>	<code>a = a * b;</code>
<code>/=</code>	<code>a /= b;</code>	<code>a = a / b;</code>
<code>%=</code>	<code>a %= b;</code>	<code>a = a % b;</code>

## Hints, & Tips, & Pitfalls



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 ✓

- Evaluate the following expressions.
  - $3 + 5 / 7$
  - $3 * 3 + 3 \% 2$
  - $3 + 2 / 5 + -2 * 4$
  - $2 * (1 + -(3/4) / 2) * (2 - 6 \% 3)$
- What is the data type of the result of the following expressions?
  - $(3 + 5) / 7$
  - $(3 + 5) / (\text{float}) 7$
  - $(\text{float}) ((3 + 5) / 7)$
- Which of the following expressions is equivalent to  $-b(c + 34)/(2a)$ ?
  - $-b * (c + 34) / 2 * a$
  - $-b * (c + 34) / (2 * a)$
  - $-b * c + 34 / (2 * a)$
- Rewrite the following statements without using the shorthand operators.
  - $x += y;$
  - $x *= v + w;$
  - $x /= y;$

### 3.3 | Constants

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

constant

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;
```

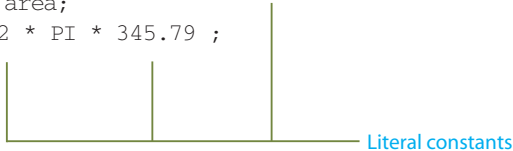
We follow the standard Java convention to name a constant, using only capital letters and underscores. Judicious use of constants makes programs more readable. You will be seeing many uses of constants later in the book, beginning with the sample program in this chapter.

named  
constant

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:

literal constant

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



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`?<sup>1</sup> 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

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

<sup>1</sup> 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>
```

Hints,  
& Tips,  
& Pitfalls



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 | Displaying Numerical Values

In Chapter 2, we learned how to output string values to the console window by using System.out. We can easily output numerical values to the console window

as well. We will use the same `print` and `println` methods to output numerical values. Here's a simple example that outputs the values of a constant and a variable:

```
int num = 15;

System.out.print(num); //print a variable
System.out.print(" "); //print a blank space
System.out.print(10); //print a constant
```

Executing the code will result in the following console window:

Console  
Window

```
15 10
```

We can use the `println` method to skip a line after printing out the value. Executing

```
int num = 15;
System.out.println(num);
System.out.println(10);
```

will result in

Console  
Window

```
15
10
```

By using the concatenation operation, it is possible to output multiple values with a single `print` or `println` method. For example, the statement

```
System.out.print(30 + " " + 40);
```

is equivalent to

```
System.out.print(30);
System.out.print(" ");
System.out.print(40);
```

Notice that the expression

```
30 + " " + 40
```

mixes numerical values and a string. We learned in Chapter 2 that the plus symbol is used to concatenate strings, for example,

```
"Benjamin" + " " + "Franklin"
```

And, in this chapter, we learned the same plus symbol is used to add numerical values, for example,

$$4 + 36$$

The plus symbol, therefore, could mean two different things: string concatenation or numerical addition. When a symbol is used to represent more than one operation, this is called *operator overloading*.

operator  
overloading

What happens when the plus symbol appears in a mixed expression? When the Java compiler encounters an overloaded operator, the compiler determines the meaning of a symbol by its context. If the left operand and the right operand of the plus symbol are both 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 a string, 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

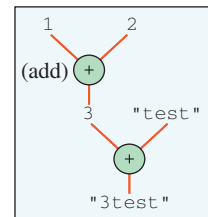
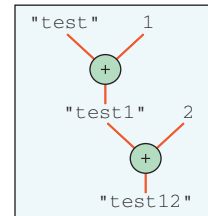
```
3test
```

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.

Now let's look at a small sample program that illustrates a typical use of string concatenation in displaying computation results. In this sample program, we compute the circumference and the area of a circle with a given radius. The value for the radius is assigned in the program (we will discuss how to input this value in Section 3.5). Here's the program:



```
/*
```

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

```
File: Ch3Circle.java
```

```
*/
```

```

class Ch3Circle {
    public static void main(String[] args) {

        final double PI = 3.14159;

        double radius, area, circumference;

        radius = 2.35;

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

        System.out.println("Given Radius: " + radius);
        System.out.println("Area: " + area);
        System.out.println("Circumference: " + circumference);
    }
}

```

When we run this program, we get the following output:

Console  
Window

```

Given Radius: 2.35
Area: 17.349430775000002
Circumference: 14.765473

```

Notice the precision of decimal places displayed for the results, especially the one for the circumference. Although we desire such a high level of precision provided by `double` values during the computation, we may not when displaying the result. We can restrict the number of decimal places to display by using the `DecimalFormat` class from the `java.text` package.

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, for example, we create a `DecimalFormat` object as

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

and use it to format the number as

```
double num = 234.5698709;
System.out.println("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

```

System.out.println("Given Radius: " + df.format(radius));
System.out.println("Area: " + df.format(area));
System.out.println("Circumference: "
                    + df.format(circumference));

```

we produce the following result:

Console  
Window

```
Given Radius:  2.350
Area: 17.349
Circumference: 14.765
```

The modified class is named `Ch3Circle2` (not shown here).

Instead of using one `println` method per line of output, it is possible to output multiple lines with a single `println` or `print` method by embedding a new-line control character in the output. We briefly mentioned a control character in Section 2.4.4. A control character is for controlling the output, and we use the backslash symbol to denote a control character. The new-line control character is denoted as `\n` and has the effect of pressing the Enter key in the output. For example, the statements

new-line  
control  
character

```
System.out.println("Given Radius: " + radius);
System.out.println("Area: " + area);
System.out.println("Circumference: " + circumference);
```

can be written by using only one `println` statement as

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

There is no limit to the number of new-line control characters you can embed, so we can easily skip two lines, for example, by putting two new-line control characters as follows:

```
System.out.println("Number 1: " + num1 + "\n\n" +
                  "Number 2: " + num2);
```

tab  
control  
character

Another useful control character is a *tab*, which is denoted as `\t`. We can use the tab *control character* to output the labels, and this results in two columns as follows:

```
System.out.println("Given Radius: " + "\t" + radius + "\n" +
                  "Area: " + "\t\t" + area + "\n" +
                  "Circumference: " + "\t" + circumference);
```

Notice there are two tabs before we output the area. You need to experiment with the actual number of tabs to get the right output (the actual number of spaces used for each tab could be different depending on your Java IDE). The resulting output will be

Console  
Window

```
Given Radius:  2.35
Area:          17.349430775000002
Circumference: 14.765473
```

You can also adjust the output format by appending blank spaces in the label. For example, you can rewrite the sample statement as

```
System.out.println("Given Radius: " + "\t" + radius + "\n" +
                  "Area:           " + "\t" + area + "\n" +
                  "Circumference: " + "\t" + circumference);
```

And, as always, the use of symbolic constants will clean up the code:

```
...
final String TAB = "\t";
final String NEWLINE = "\n";
...
System.out.println(
    "Given Radius: " + TAB + radius + NEWLINE +
    "Area:         " + TAB + area  + NEWLINE +
    "Circumference: " + TAB + circumference);
```

The new program that illustrates the use of both `DecimalFormat` and control characters is named `Ch3Circle3`. Here's the program:

```
/*
Chapter 3 Sample Program: Compute Area and Circumference
File: Ch3Circle3.java
*/
import java.text.*;
class Ch3Circle3 {
    public static void main(String[] args) {
        final double PI = 3.14159;
        final String TAB = "\t";
        final String NEWLINE = "\n";

        double radius, area, circumference;

        DecimalFormat df = new DecimalFormat("0.000");

        radius = 2.35;

        //compute the area and circumference
        area      = PI * radius * radius;
        circumference = 2.0 * PI * radius;
        //Display the results
        System.out.println(
            "Given Radius: " + TAB + df.format(radius) + NEWLINE +
```

```

"Area:          " + TAB + df.format(area)  + NEWLINE +
"Circumference: " + TAB + df.format(circumference));
    }
}

```



1. What is the purpose of the control characters?
2. Which control character is used for a new line?
3. Using one print statement, output the following:

```

Hello, world!
    My favorite Ben Franklin quote:

        An investment in knowledge
        always pays the best interest.

```

### 3.5 Getting Numerical Input

We learned how to input string values by using the `Scanner` class in Chapter 2. We study how to input numerical values with the `Scanner` class in this section. To input strings, we use the `next` method of the `Scanner` class. For the numerical input values, we use an equivalent method that corresponds to the data type of the value we try to input. For instance, to input an `int` value, we use the `nextInt` method. Here's an example of inputting a person's age:

```

Scanner scanner = new Scanner(System.in);
int age;
System.out.print("Enter your age: ");
age = scanner.nextInt();

```

In addition to the `int` data type, we have five input methods that correspond to the other numerical data types. The six input methods for the primitive numerical data types are listed in Table 3.6.

**Table 3.6** Methods to input six numerical data types

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

The following example inputs a person's height in inches (int) and GPA (float):

```
Scanner scanner = new Scanner(System.in);
int height;
float gpa;
System.out.print("Enter your height in inches: ");
height = scanner.nextInt();
System.out.print("Enter your gpa: ");
gpa = scanner.nextFloat();
```

Remember that the default delimiter between the input values is a white space (such as the blank space or a tab); it is possible to input more than one value on a single line. The following code inputs two integers:

```
Scanner scanner = new Scanner(System.in);
int num1, num2;
System.out.print("Enter two integers: ");
num1 = scanner.nextInt();
num2 = scanner.nextInt();
System.out.print("num1 = " + num1 + " num2 = " + num2);
```

And here's a sample interaction:

Space separates the  
two input values.

```
Enter two integers: 12 8
num1 = 12 and num2 = 87
```

Since the new-line character (when we press the Enter key, this new-line character is entered into the system) is also a white space, we can enter the two integers by pressing the Enter key after each number. Here's a sample:

```
Enter two integers: 12
87
num1 = 12 and num2 = 87
```

#### input buffer

When we enter data using `System.in`, they are placed in *input buffer*. And the next available data in the input buffer are processed when one of the input methods is called. This means that the actual processing of 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;
```



```

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.print("Values entered are " +
                num1 + " " + num2 + " " + num3);

```

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

```

Enter Number 1: 10 
Enter Number 2: 20 
Enter Number 3: 30 
Values entered are 10 20 30

```

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. This will result in an awkward display in the console window. Here's an example:

```

Enter Number 1: 10, 20, 30 
Enter Number 2: Enter Number 3: Values entered are 10 20 30

```

Although the display is awkward, the input values are assigned to the respective variables correctly. The three input values are placed in the input buffer, and when the second and third `nextInt` methods are called, the corresponding values are already in the input buffer, so they get assigned to the variables correctly.

In Section 3.2, we explained the assignment conversion that allows us to assign a value to a higher-precision variable (e.g., assigning an `int` value to a `double` variable). This type of implicit conversion also occurs with the `Scanner` class. For example, the `nextDouble` method works without a problem as long as the user enters a value that is assignable to a `double` variable. Here's an example:

```

Scanner scanner = new Scanner(System.in);

double num;

System.out.print("Enter a double: ");
num = scanner.nextDouble( );

System.out.print("You entered " + num);

```

```

Enter a double: 35 
You entered 35.0

```

The `nextDouble` method accepts the value 35 and then converts it to a double data type. This method returns a double value, so even if the user enters an integer, you cannot assign the input to an `int` variable. The following code is therefore invalid:

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

int num;

System.out.print("Enter an integer: ");
num = scanner.nextDouble( ); _____ Type mismatch

System.out.print("You entered " + num);
```

Now let's study how we can mix the input of strings and numerical values. We begin with an example. Consider inputting a racehorse's name and age. Here are a proposed code and a sample of expected interaction:

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

String horseName;
int age;

System.out.print("Enter the horse name: ");
horseName = scanner.next( );

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

System.out.print(horseName + " is " + age + "years old. ");
```

```
Enter the horse name: Barbaro 
Enter the age: 3 
Barbaro is 3 years old.
```

Everything seems to be working okay. What will happen if the name of a horse has more than one word, such as Sea Biscuit? The code will not work because only the first word is assigned to the `String` variable `horseName`. Remember that the default delimiter is the white space, so the blank space after the first word is treated as the end of the first input. Here's the result when you enter Sea Biscuit:

```
Enter the horse name: Sea Biscuit 
Enter the age: java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:819)
at java.util.Scanner.next(Scanner.java:1431)
at java.util.Scanner.nextInt(Scanner.java:2040)
...
```

Only the first four lines of error messages are shown here.

The most reasonable solution here is to change the delimiter to the line separator, as described in Section 2.4.4. Here's how:

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

//the rest is the same
```

```
Enter the horse name: Sea Biscuit  ENTER
Enter the age: 3  ENTER
Sea Biscuit is 3 years old.
```

For most situations, using the line separator as the delimiter and inputting one value per input line are the best approach. We can, however, use any string for the delimiter. So, for example, we can delimit the input values with a character such as the pound sign (#), provided, of course, that the pound sign does not occur in the actual input values.

## Things to Remember



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

Instead of using the data type specific methods such as `nextInt`, `nextDouble`, and others of the `Scanner` class, we can input a numerical value in a string format and convert it to an appropriate data type by ourselves. For example, we can use the class method `parseInt` of the `Integer` class to convert a string to an `int`. Here's a statement that converts "14" to an `int` value 14:

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

So, the statement

```
int num = Integer.parseInt(scanner.next());
```

is equivalent to

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

Passing a string that cannot be converted to an `int` (e.g., "12b") will result in an error. The conversion method is not particularly useful or necessary with the scanner, but it can be when the input source is different from the scanner. Other common conversion methods are `parseDouble`, `parseFloat`, and `parseLong` of the `Double`, `Float`, and `Long` classes, respectively.

We close this section by presenting a sample program that extends the Ch3Circle3 class by accepting the radius of a circle as an input. Here's the program:

```

/*
Chapter 3 Sample Program: Compute Area and Circumference with formatting
and standard I/O

File: Ch3Circle4.java
*/
import java.text.*;
import java.util.*;

class Ch3Circle4 {

    public static void main(String[] args) {

        final double PI = 3.14159;
        final String TAB = "\t";
        final String NEWLINE = "\n";

        double radius, area, circumference;

        Scanner scanner = new Scanner(System.in);

        DecimalFormat df = new DecimalFormat("0.000");

        System.out.println("Enter radius: ");
        radius = scanner.nextDouble( );

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

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

```



1. Write a code to input the height of a user in feet (int) and inches (int).
2. Write a code to input the full name of a person and his or her age. The full name of a person includes the first name and the last name.
3. Write a code that creates a Scanner object and sets its delimiter to the pound sign.

### 3.6 | 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 3.7 is a partial list of class methods available in the Math class. The class also has two class constants PI and E for  $\pi$  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 3.7** Math class methods for commonly used mathematical functions

Class Method	Argument Type	Result Type	Description	Example
abs( a )	int	int	Returns the absolute 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 ) <sup>†</sup>	double	double	Returns the arccosine of a.	acos(-1) → 3.14159
asin( a ) <sup>†</sup>	double	double	Returns the arcsine of a.	asin(1) → 1.57079
atan( a ) <sup>†</sup>	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 ) <sup>†</sup>	double	double	Returns the trigonometric cosine of a.	cos( $\pi/2$ ) → 0.0
exp( a )	double	double	Returns the natural number $e$ (2.718 ...) raised to the power of a.	exp(2) → 7.389056099

**Table 3.7** Math class methods for commonly used mathematical functions (Continued)

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

<sup>†</sup>All trigonometric functions are computed in radians.

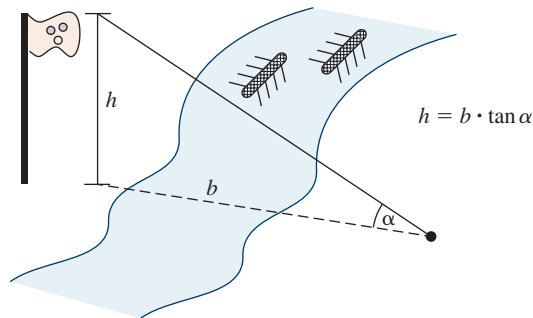
Notice how the class methods and class constants are referred to in the expression. The syntax is

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

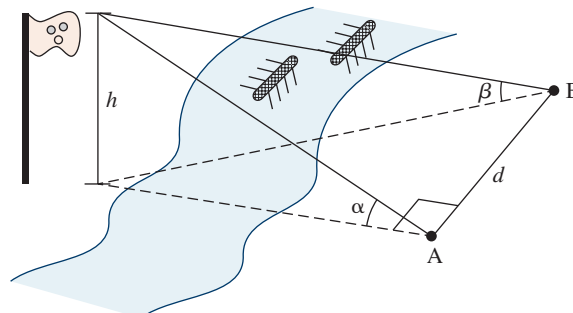
or

```
<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("\n\n 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 );`

### 3.7 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` class from the `java.util` package. (Alternatively, we can use the `random` method of the `Math` class to generate random numbers, but it is more difficult to use than the `Random` class).

For most applications, the random numbers we want to generate are integers. To generate random integers, we use the `nextInt` method of the `Random` class. Here's an example:

```

import java.util.*;
...
Random random = new Random( );
int num = random.nextInt( );

```

The `nextInt` method returns an `int` value, that is any value between  $-2147483648$  and  $2147483647$  (see Table 3.1). To restrict range of possible values, we can use the second form of the `nextInt` method in which we pass an argument that specifies the upper bound of the range. The lower bound is set to 0. To generate a random integer between 0 and 10, for example, we write as follows:

```
int num = random.nextInt(11);
```

Notice that the argument we pass is 11, not 10. The argument we pass to the method specifies the total number of possible values, starting from 0. So passing the value of 11 specifies that we are getting one of the 11 possible values, ranging from 0 to 10.

Often we want the lower bound of the possible range to be other than 0. There is no method in the `Random` class that allows us to specify both the lower bound. It is possible, however, to generate any number between *min* and *max* where *min* is greater than 0? Suppose, for example, we want a number between 1 and 6. We can first generate a number between 0 and 5 and then add 1 to the result as

```
int num = random.nextInt(6) + 1;
```

This generates an integer  
between 0 and 5, inclusive.

Let's derive a formula for the general case. To generate a random integer in the range of  $[\text{min}, \text{max}]$  where  $0 \leq \text{min} < \text{max}$ , we write

```
int num = random.nextInt(max-min+1) + min;
```

This generates an integer between  
0 and (max-min), inclusive.

The expression `random.nextInt(max-min+1)` returns an integer between 0 and  $(\text{max}-\text{min})$ . By adding *min* to this value, the final result will then be a value between  $0 + \text{min} = \text{min}$  and  $(\text{max} - \text{min}) + \text{min} = \text{max}$ , as desired.



The `nextInt` (and other methods) in the `Random` class is called a *pseudorandom number generator* because the number is not truly random. When we call the 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.

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 determined by the fraternity president. 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 using the Random class  
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  
  
        Random random = new Random(); //random number generator  
  
        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 = random.nextInt(max-min+1) + min;  
  
        System.out.println("\nThe Winning Number is " + winningNumber);  
  
    }  
}
```

## Quick CHECK



1. What are the possible minimum and the maximum numbers assigned to `num` by the following statement?

```
int num = random.nextInt(15);
```

2. Write a statement that assigns a random number between 1 and 4, inclusive, to an integer variable `num`.
3. Write a statement that assigns a random number between `min` and `max`, inclusive, where  $0 \leq \text{min} < \text{max}$ , to an integer variable `num`.

### 3.8 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.

#### Things to Remember



*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.”*

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

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

The value of 6  
means July.

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

**Table 3.8** Constants defined in the `Calendar` class for retrieved different pieces of calendar/time information

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

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 3.8 explains the use of some of the more common constants defined in the `Calendar` class.

When the date and time are February 13, 2008, 13:30 p.m. and we run the `Ch3TestCalendar` program, we will see the result shown in Figure 3.4.

```
Wed Feb 13:30:51 PST 2008
YEAR:          2008
MONTH:         1
DATE:          13
DAY_OF_YEAR:   44
DAY_OF_MONTH:  13
DAY_OF_WEEK:   4
WEEK_OF_YEAR:  7
WEEK_OF_MONTH: 3
AM_PM:         1
HOUR:          1
HOUR_OF_DAY:  13
MINUTE:        30
```

**Figure 3.4** Result of running the `Ch3TestCalendar` program at February 13, 2008, 13:30 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));
    }
}

```

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.*;

class Ch3IndependenceDay {

    public static void main(String[] args) {

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

        SimpleDateFormat sdf = new SimpleDateFormat("EEEE");

        System.out.println("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). 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) {

        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 `Ch3IndependenceDay` 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.ushistory.org/declaration/>



## 3.9 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. For input and output, we continue to use the **Scanner** class and **System.out (PrintStream)**. For computing the monthly and total payments, there are no standard classes that will provide such 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

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

## 3.9 Sample Development—continued

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

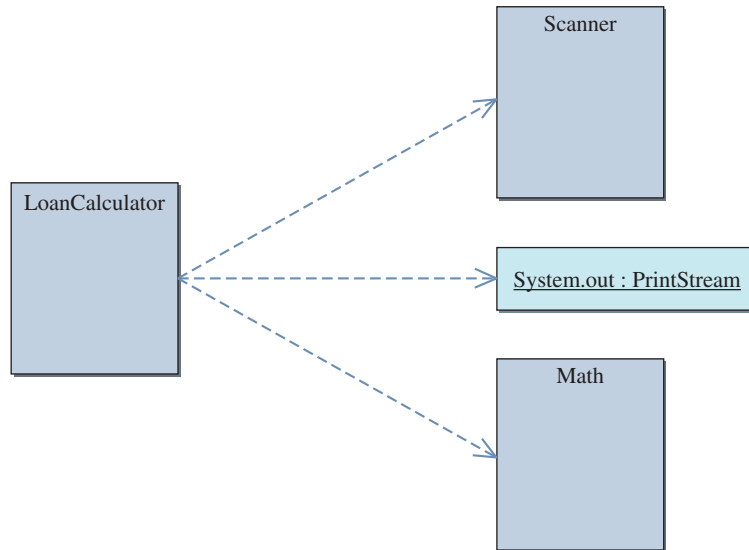
The program diagram based on the classes listed in the design document is shown in Figure 3.5. Keep in mind that this is only a preliminary design. The preliminary document is really a working document that we will modify and expand as we progress through the development steps.

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:

develop-  
ment 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



**Figure 3.5** The object diagram for the program **LoanCalculator**.

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. 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

3.9 Sample Development—*continued*

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 echo-print the input values to **System.out**.

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 java.util.*;

class Ch3LoanCalculator {

    public static void main(String[] args) {

        double loanAmount,
            annualInterestRate;

        int loanPeriod;

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

        //get input values
        System.out.print("Loan Amount (Dollars+Cents): ");
        loanAmount = scanner.nextDouble();

        System.out.print("Annual Interest Rate (e.g., 9.5): ");
        annualInterestRate = scanner.nextDouble();
    }
}

```

```

System.out.print("Loan Period - # of years: ");
loanPeriod = scanner.nextInt();

//echo print the input values
System.out.println("");
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 design

**Step 2 Development: Output Values**

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 3.6 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.

## 3.9 Sample Development—continued

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

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.

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 Results
*/
import java.util.*;

class Ch3LoanCalculator {
    public static void main(String[] args) {
        double loanAmount,
               annualInterestRate;
        double monthlyPayment,
               totalPayment;
        int loanPeriod;
```

```

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

//get input values
System.out.print("Loan Amount (Dollars+Cents): ");
loanAmount = scanner.nextDouble();

System.out.print("Annual Interest Rate (e.g., 9.5): ");
annualInterestRate = scanner.nextDouble();

System.out.print("Loan Period - # of years: ");
loanPeriod = scanner.nextInt();

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

//display the result
System.out.println("");
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 **println** methods until we get the layout that looks clean and nice on the screen.

**step 3 design****Step 3 Development: Compute Loan Amount**

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;

```

## 3.9 Sample Development—continued

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 26 on page 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: Display the Results
*/
import java.util.*;

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;

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

//get input values
System.out.print("Loan Amount (Dollars+Cents): ");
loanAmount = scanner.nextDouble();

System.out.print("Annual Interest Rate (e.g., 9.5): ");
annualInterestRate = scanner.nextDouble();

System.out.print("Loan Period - # of years: ");
loanPeriod = scanner.nextInt();

//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("");
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.

3.9 Sample Development—*continued*

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.

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 java.util.*;
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;
```

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

```
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
```

```
System.out.print("Loan Amount (Dollars+Cents): ");
loanAmount = scanner.nextDouble( );
```

```
System.out.print("Annual Interest Rate (e.g., 9.5): ");
annualInterestRate = scanner.nextDouble( );
```

```
System.out.print("Loan Period - # of years: ");
loanPeriod = scanner.nextInt( );
```

```
//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("");
System.out.println("Loan Amount:          $" + loanAmount);
```

3.9 Sample Development—*continued*

```

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.

### Things to Remember



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

### 3.10 Numerical Representation (Optional)

twos complement

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.

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

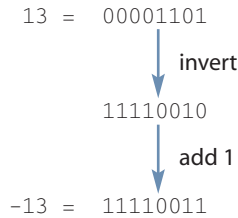
sign bit

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 two's complement representation. Notice that zero has only one representation.

8-Bit Binary Number (Two's 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

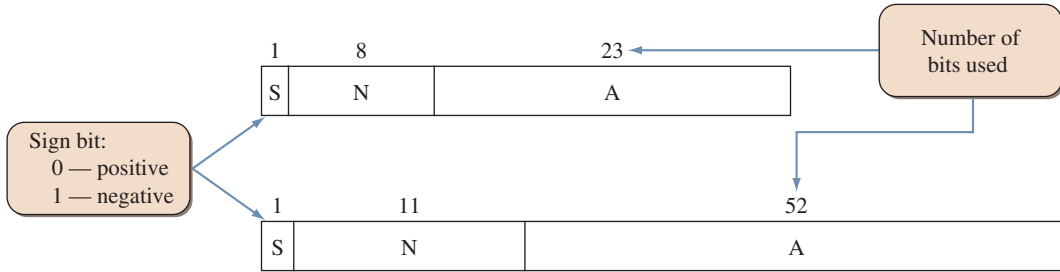
$$A \times 10^N$$

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  $1.67339 \times 10^{-24}$ , which is equal to 0.000000000000000000000000167339.

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.

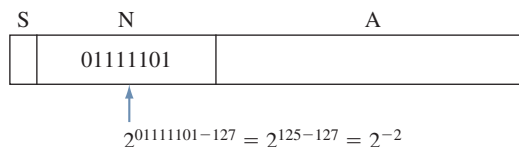


excess format

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 arithmetic expressions.
- Symbolic constants hold values just as variables do, but we cannot change their values.
- The standard classes introduced in this chapter are
  - Math
  - GregorianCalendar
  - DecimalFormat
- 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 Scanner object to System.in to input primitive data type.
- 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	assignment conversion
primitive data types	constants
reference data types	standard output
arithmetic expression	standard input
arithmetic operators	echo printing
precedence rules	twos complement ( <i>optional</i> )
typecasting	floating point ( <i>optional</i> )
implicit and explicit casting	



## Chapter 3 Exercises

### Review 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.

- $(x + 1.5) / (250.0 * (i/j))$
- $x + 1.5 / 250.0 * i / j$
- $-x * -y * (i + j) / k$
- $(i / 5) * y$
- `Math.min(i, Math.min(j,k))`
- `Math.exp(3, 2)`
- $y \% x$
- `Math.pow(3, 2)`
- $(int)y \% k$
- $i / 5 * y$

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.

- `w = Math.pow(3, Math.pow(i, j));`
- `v = x / i;`
- `w = Math.ceil(y) % k;`
- `n = (int) x / y * i / 2;`
- `x = Math.sqrt(i*i - 4*j*k);`
- `m = n + i * j;`
- `n = k / (j * i) * x + y;`
- `i = i + 1;`
- `w = float(x + i);`
- `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?

- `i = x;`
- `x = u + y;`
- `x = 23.4 + j * y;`
- `v = (int) x;`
- `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  $\pi R^2$  of a circle
5. Determine the output of the following program without running it.

```
class TestOutput {
    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 ) );
```

### Level 1 Programming Exercises ★

7. Write a program to convert centimeters (input) to feet and inches (output).  
1 in = 2.54 cm.
8. Write a program that inputs temperature in degrees Celsius and prints out the temperature in degrees Fahrenheit. The formula to convert degrees Celsius to equivalent degrees Fahrenheit is

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

9. Write a program 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;
```

(Note: We are not distinguishing between genders.)

10. Write a program that does the reverse of Exercise 9, that is, input degrees Fahrenheit and prints out the temperature in degrees Celsius. The formula to convert degrees Fahrenheit to equivalent degrees Celsius is

$$\text{Celsius} = \frac{5}{9} \times (\text{Fahrenheit} - 32)$$

11. Write a program that inputs the year a person is born and outputs the age of the person in the following format:

You were born in 1990 and will be (are) 18 this year.

12. 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.

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

$$P(1 + R/100)^N$$

dollars. Write an application that accepts  $P$ ,  $R$ , and  $N$  and computes the amount of money earned after  $N$  years.

14. The volume of a sphere is computed by the equation

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

where  $V$  is the volume and  $r$  is the radius of the sphere. Write a program that computes the volume of a sphere with a given radius  $r$ .

### Level 2 Programming Exercises ★★

15. The velocity of a satellite circling around the Earth is computed by the formula

$$v = \sqrt{\frac{GM_E}{r}}$$

where  $M_E = 5.98 \times 10^{24}$  kg is the mass of the Earth,  $r$  the distance from the center of the Earth to the satellite in meters, and  $G = 6.67 \times 10^{-11}$  m<sup>3</sup>/kg · s<sup>2</sup> the universal gravitational constant. The unit of the velocity  $v$  is m/s. Write a program that inputs the radius  $r$  and outputs the satellite’s velocity. Confirm that a satellite that is closer to the Earth travels faster. Define symbolic constants for  $M_E$  and  $G$ . The distance to the Hubble Space Telescope from the center of the Earth, for example, is approximately  $6.98 \times 10^6$  m.

16. 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 a program that inputs the user's Earth weight and outputs her or his weight on Mercury, Venus, Jupiter, and Saturn. Use the values in this table.

<b>Planet</b>	<b>Multiply the Earth Weight by</b>
Mercury	0.4
Venus	0.9
Jupiter	2.5
Saturn	1.1

17. 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. The values for  $d$  are listed in the table.

**$d =$  Approximate Number of Earth Days for This Planet to Travel around the Sun**

<b>Planet</b>	<b><math>d</math></b>
Mercury	88
Venus	225
Jupiter	4,380
Saturn	10,767

18. Write a program 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  $x$ .

19. Write a program 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.
20. 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.
21. 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.

<u>Month No.</u>	<u>Number of Pairs</u>
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.

22. 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  $6.67\text{E-}8 \text{ dyn} \cdot \text{cm}^2/\text{g}^2$ . 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  $1.984\text{E}25 \text{ dyn}$ . The mass of the earth is  $5.983\text{E}27 \text{ g}$ , the mass of the moon is  $7.347\text{E}25 \text{ g}$ , and the distance between the two is  $3.844\text{E}10 \text{ cm}$ .

23. 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  $R$ . Ms. Latte's constant is  $2.5\text{E}2 \text{ mocha lines}^2/\text{min}^2$ . (*Note*: This is just for fun. Develop your own law, using various functions from the Math class.)

### Level 3 Programming Exercises ★★★

24. Write a program 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

Draw the program diagram.

25. 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.

26. In Section 3.9, 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 3.9 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

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

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.

27. Ask the user to enter his or her birthday in the MM/DD/YYYY format and output the number of days between the birthday and today. This gives the person's age in days.

### 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.

28. 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. 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).

29. 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 in the least expensive manner. For example, the order of 52 bags will be shipped in four boxes: two large, one medium, and one small. The rule for packing is to fill the large and medium boxes completely; that is, the box is fully packed. Only the small boxes can have empty spaces. For example, to ship 52 bags, you could have used 3 large boxes, but that would leave the third box not fully packed. Develop a program that computes the total cost of an order. 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

```

- 30.** Repeat Exercise 29, 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, 2008 is entered as 11/01/2008. 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

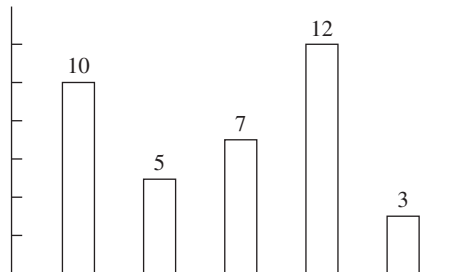
Your total cost is: $ 291.20

Date of Order:           November 1, 2008
Expected Date of Arrival: November 15, 2008

```

- 31.** Using a Turtle object from the `galapagos` package, draw three rectangles. 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 the McGraw-Hill book website.

32. Develop a program that draws a bar chart using a Turtle object. 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.