

# 3

## 3 Numerical Data

### Objectives

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

- Select proper types for numerical data.
- Write arithmetic expressions in Java.
- Evaluate arithmetic expressions using 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.
- Convert input string values to numerical data.
- 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 also introduce more standard classes to reinforce the object-oriented style of programming. In addition to the standard classes, we will introduce two author-written classes that provide convenient input and output routines. The use of these classes illustrate some of the important aspects of object-oriented programming, but their use in the actual programs is strictly optional. We will learn how to implement a simplified version of these classes in the next 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, a well planned development steps will smooth the development effort.

## 3.1 | Variables

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

$$x + y$$

and

$$x - y$$

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

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

```
int    x, y;
```

variable

When this declaration is made, memory locations to store data values for  $x$  and  $y$  are allocated. These memory locations are called *variables*, and  $x$  and  $y$  are the names we

associate with the memory locations. Any valid identifier can be used as a variable name. After the declaration is made, we can assign only integers to *x* and *y*. We cannot, for example, assign real numbers to them.

### Helpful Reminder



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

Although we must say “*x* and *y* are variable names” to be precise, we will use abbreviated forms “*x* and *y* are variables” or “*x* and *y* are integer variables” whenever appropriate.

The general syntax for declaring variables is

variable  
declaration  
syntax

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

where **<variables>** is a sequence of identifiers separated by commas. Every variable you 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, you 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.

**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 181.

<sup>††</sup> The character E indicates a number is expressed in scientific notation. This notation is explained on page 107.

## 4 Chapter 3 Numerical Data

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 trade-off for higher precision is memory space—to store a number with higher precision, you need more space. A variable of type **short** requires 2 bytes and a variable of type **int** requires 4 bytes, for example. If your program does not use many integers, then whether you declare them as **short** or **int** is really not that critical. The difference in memory usage is very small and not a deciding factor in the program design. The 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.

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

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

### Take my Advice



In the same way that you can initialize variables at the time you declare them, you can declare and create an object at the same time. 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 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;
```

The syntax for the assignment statement is

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

*assignment  
statement syntax*

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 you need to remember about variables is

### Helpful Reminder



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

The diagram in Figure 3.2 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 44. Figure 3.1 compares the two. What we have been calling object names are really variables. The only difference between a variable for numbers and a variable for objects is the contents in the memory locations. For numbers, a variable contains the numerical value itself, and for objects, a variable contains an address where the object is stored. We use an arrow in the diagram to indicate that the content is an address, not the value itself.

### Helpful Reminder



*Object names are synonymous to 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. You can view the situation where two variables refer to the same object as the object having two distinct names.

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, for example. The difference in space usage for the account objects would occur if we have to keep track of checks written against the accounts. If one account has 15 checks

## 6 Chapter 3 Numerical Data

### 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 3.1** A difference between object declaration and numerical data declaration.

written and the second account has 25 checks written, then we need more memory space for the second account than 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 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*.

reference vs.  
primitive data  
types

## Quick CHECK

1. Why are the following declarations all invalid?

```
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

$$23 + 45$$

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

## 8 Chapter 3 Numerical Data

(A) `int firstNumber, secondNumber;`  
`firstNumber = 234;`  
`secondNumber = 87;`

## State of Memory

after (A) is executed

firstNumber

secondNumber

The variables **firstNumber** and **secondNumber** are declared and set in memory.

(B) `int firstNumber, secondNumber;`  
`firstNumber = 234;`  
`secondNumber = 87;`

after (B) is executed

firstNumber

secondNumber

Values are assigned to the variables **firstNumber** and **secondNumber**.

**Figure 3.2** A diagram showing how two memory locations (variables) with names **firstNumber** and **secondNumber** are declared, and values are assigned to them.

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

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

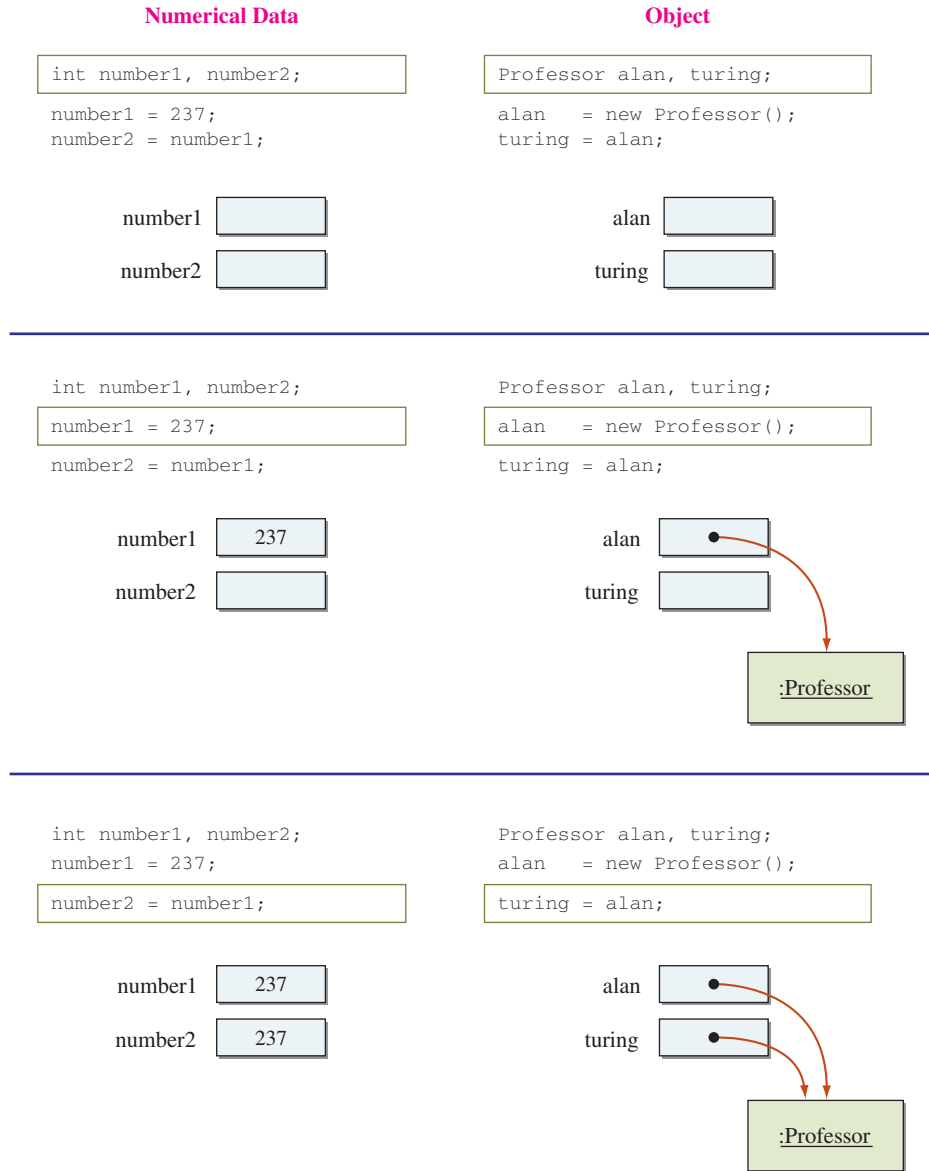
$$x + 4$$

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

$$-x$$





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

In the expression

$$x + 3 * y$$

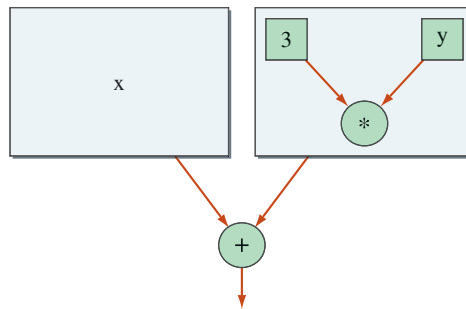
subexpression

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

**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

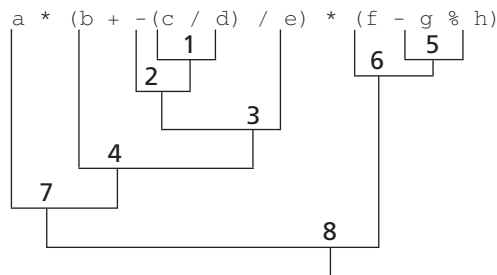
subexpression 3 \* y has operands 3 and y. The following diagram illustrates this relationship:





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.

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



**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.

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 of different data types, then a casting conversion will take place. A **casting conversion**, or **type casting**, 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**. 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 **type cast operator**. For example, to convert the *int* variable *x* in the expression

$$x / 3$$

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

$$(\text{float}) x / 3$$

type casting

numeric promotion

type cast operator

**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 <b>byte</b> or <b>short</b>, then it is converted to <b>int</b>.</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 <b>double</b>, then the other operand is converted to <b>double</b>.</li> <li>2. Otherwise, if either operand is of type <b>float</b>, then the other operand is converted to <b>float</b>.</li> <li>3. Otherwise, if either operand is of type <b>long</b>, then the other operand is converted to <b>long</b>.</li> <li>4. Otherwise, both operands are converted to <b>int</b>.</li> </ol>

The syntax is

type casting  
syntax

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

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

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

will result in the subexpression  $x + y * z$  type casted 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 the 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,

assignment  
conversion

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

is valid, but

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

is not.

### Take my Advice



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

$$x = y = 1;$$

which is equivalent to saying

$$y = 1;$$

$$x = 1;$$

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

### Quick CHECK

1. Evaluate the following expressions:

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

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

- a.  $(3 + 5) / 7$
- b.  $(3 + 5) / (\text{float}) 7$
- c.  $(\text{float}) ((3 + 5) / 7)$

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

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

### 3.3 Constants

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

constant

## 14 Chapter 3 Numerical Data

Here's an example 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 use 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* 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 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 ;
```

Literal Constants

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

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

How about the literal constant `345.79`? Since the literal constant contains a decimal point, its data type can only be `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
```

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

### Take my Advice



Since a numerical constant such as 345.79 represents a *double* value, the following 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 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 people make in writing Java programs, especially those with prior programming experience.

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

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

which in Java is expressed as

<number> E <exponent>  *exponential  
notation in Java*

where <number> is a literal constant that may or may not contain a decimal point and <exponent> is a signed or 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/sec
final short MAX_WGT_ALLOWED = 400;
```

### 3.4 Getting Numerical Input Values

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

```
int age;

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

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

```
String num = 14;
```

or

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

type  
mismatch

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

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

type  
conversion

wrapper  
classes

Integer

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

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

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

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

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

If the user enters a string that cannot be converted to an `int`, for example, 12.34 or abc123, an error will result. Table 3.5 lists common wrapper classes and their corresponding conversion method.



**Table 3.5** Common wrapper classes and their conversion method.

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

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

```

/*
Chapter 3 Sample Program: Compute Area and Circumference

File: Ch3Circle.java

*/

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

class Ch3Circle {

    public static void main( String[] args ) {

        final double PI = 3.14159;

        String radiusStr;

        double radius, area, circumference;

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

        radius = Double.parseDouble(radiusStr);

        //compute area and circumference

```

## 18 Chapter 3 Numerical Data

```

area = 2.0 * PI * radius;
circumference = PI * radius * radius;

JOptionPane.showMessageDialog(null,
    We are allowed to concatenate String
    and numerical data. + "Given Radius: " + radius + "\n"
    "Area: " + area+ "\n"
    + "Circumference: " + circumfer-
ence);
}
}

```

the + symbol means addition or concatenation

operator overloading

Notice the long expression we pass as the second parameter for the `showMessageDialog` method. In Chapter 2, we use the plus symbol to concatenate strings. We can use the same symbol to concatenate strings and numerical data. Numerical data are automatically converted to string representation and then concatenated. We learned in this chapter that the plus symbol is used for arithmetic addition also. A symbol that is used to represent more than one operation is called an *overloaded operator*. When the Java compiler encounters an overloaded operator, how does it know which operation the symbol represents? The Java compiler determines the meaning of a symbol by its context. If the left and the right operand of the plus symbol are numerical values, then the compiler will treat the symbol as addition. Otherwise, it will treat the symbol as concatenation. The plus symbol operator is evaluated from left to right, and the result of concatenation is a text, so the code

```

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

```

will result in output being set to

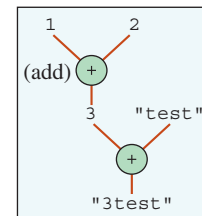
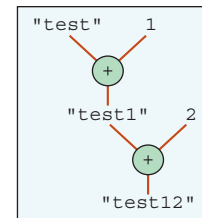
```
test12
```

while the statement

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

will result in output being set to

```
3test
```



DecimalFormat

Getting back to the program, when we run the program and enter 2.35, the dialog shown in Figure 3.4 appears on the screen. Notice the precision of decimal places displayed for the results, especially the one for the circumference. We can specify the number of decimal places to display by using the `DecimalFormat` class from the `java.text` package. The usage is similar to the one for the `SimpleDateFormat` class.

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

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

and use it format the number as

```
double num = 234.5698709;

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

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

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

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

## Quick CHECK

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

### 3.5 Standard Output

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

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

standard  
output window

System.out

## Helpful Reminder



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

The `System` class includes a number of useful class data values. One of them is an instance of the `PrintStream` class named `out`. Since this is a class data value, we refer to it through the class name, as `System.out`, and this `PrintStream` object is tied to the standard output window. Every text we send to `System.out` will appear on the standard output window.

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

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

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

```
Hello, Dr. Caffeine
```

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

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

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

Notice that in the statement

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

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

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

is equivalent to

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

```
Hello, Dr. Caffeine. x = 123 x + x = 246 THE END
```

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

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

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

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

```
Hello, Dr. Caffeine.
x = 123
x + x = 246
THE END
```

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

## Quick CHECK

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

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

### 3.6 (Optional) Standard Input

For programs that require simple input routines, the use of `showInputDialog` would suffice. As an alternative, we will introduce an approach that works nicely with `System.out`. Some people may prefer this approach because of the tighter integration with `System.out`.

System.in

Analogous to `System.out` for output, we have `System.in` for input. However, using `System.in` for input is slightly more complicated than using `System.out` for output. `System.in` is an instance of the `InputStream` class that provides a facility for inputting bytes. A byte consists of 8 bits, and it is the basic unit of organizing the memory. For more explanation of bytes and bits, please refer to Chapter 0. We cannot use `System.in` directly because it allows us to read only a single byte with its `read` method. But multiple bytes are used to represent primitive data types and strings. For example, the primitive data types `int` and `float` require 4 bytes and string data of  $N$  characters requires  $2N$  bytes (note: a Unicode-based Java character requires 2 bytes).

BufferedReader

The most common way of using `System.in` for input is to associate a `BufferedReader` object to the `System.in` object. This `BufferedReader` object will allow us to read a single line of text just as the `showInputDialog` method of `JOptionPane`. Once the string data is read, we can then convert it to a data of primitive data type using the type conversion techniques mentioned earlier in the chapter. To associate a `BufferedReader` object to `System.in`, we write

```
BufferedReader bufReader;
bufReader = new BufferedReader(
    new InputStreamReader( System.in ) );
```

InputStream-Reader

Notice that we create an intermediate `InputStreamReader` object. An `InputStreamReader` object allows us to read a single character at a time, and a `BufferedReader` object allows us to read a single line of text, that consists of multiple characters, at a time. In other words, a succession of three objects incrementally add more reading capability as illustrated in Figure 3.10.

Once the `BufferedReader` object is established, we use its `readLine` method to read a single line of text as a `String`. We designate the end of a line by pressing the Enter (or Return) key. Here's an example:

```
String name;
try {
    name = bufReader.readLine( );
} catch (IOException e) { }
```

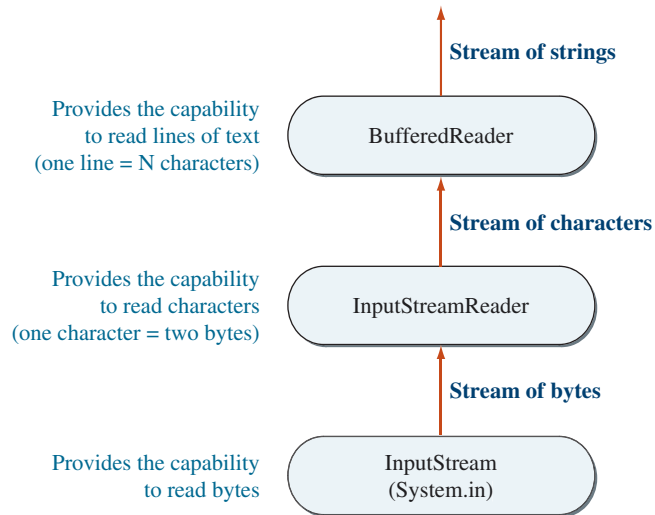
exception

Calling the `readLine` method can result in an error condition called an *exception*. We will defer the full explanation of exceptions and how to process them until Chapter 8. For now, it suffices just to remember that we have to surround the statement that calls the `readLine` method with the following try-catch block:

try-catch

```
try {
    ...
} catch (IOException e) { }
```

**No Semicolon**  
Notice there's no semicolon here.



**Figure 3.10** How the sequence of I/O objects incrementally add more capabilities.

To input a primitive data, say, `int`, we employ the same type conversion technique we used in conjunction with the `showInputDialog`. For example, to input an age, we write

```

String inputStr;
int age;

try {
    inputStr = bufReader.readLine( );
} catch (IOException e) { }

age = Integer.parseInt(inputStr);
  
```

When we use the `showInputDialog` method of `JOptionPane`, we see a dialog appears on the screen, and the characters we enter are displayed in the dialog. With `System.in`, we see what we enter on the standard output window. When an input dialog appears on the screen, it serves as a visual clue that the program is waiting for our input. With `System.in`, there will no such dialog, so we have to display some form of prompt on the standard output window in the following manner:

```

System.out.print("Enter your age: ");

try {
    inputStr = bufReader.readLine( );
} catch (IOException e) { }

age = Integer.parseInt(inputStr);
  
```

Finally, if we need to input multiple data values, it is more convenient to include all input routines in a single **try-catch** block instead of using multiple **try-catch** blocks, one for each **readLine** method. Here's an example:

```
try {
    System.out.print("Enter your age: ");
    inputStr = bufReader.readLine( );
    age      = Integer.parseInt(inputStr);

    System.out.print("Enter your weight: ");
    inputStr = bufReader.readLine( );
    wgt     = Double.parseDouble(inputStr);

} catch (IOException e) { }
```

## Quick CHECK

1. Using the standard input, write a code fragment to input the user's height in inches and assign the value to an `int` variable named `height`.
2. Using the standard input and output, write a code fragment to input the user's weight in pounds and display the weight in kilograms. 1 lb = 453.592 grams.

### 3.7 | The Math Class

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

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

we need the trigonometric sine and square root functions. The **Math** class in the `java.lang` package contains class methods for commonly used mathematical functions. Table 3.6 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) )
```

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>
```



**Table 3.6** *Math* class methods for commonly used mathematical functions.

Class Method	Argument Type	Result Type	Description	Example
abs( a )	int	int	Returns the absolute int value of <b>a</b> .	abs(10) → 10 abs(-5) → 5
	long	long	Returns the absolute long value of <b>a</b> .	
	float	float	Returns the absolute float value of <b>a</b> .	
acos( a ) <sup>†</sup>	double	double	Returns the arc cosine of <b>a</b> .	acos(-1) → 3.14159
asin( a ) <sup>†</sup>	double	double	Returns the arc sine of <b>a</b> .	asin(1) → 1.57079
atan( a ) <sup>†</sup>	double	double	Returns the arc tangent of <b>a</b> .	atan(1) → 0.785398
ceil( a )	double	double	Returns the smallest whole number greater than or equal to <b>a</b> .	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 <b>a</b> .	cos( $\pi/2$ ) → 0.0
exp( a )	double	double	Returns the natural number e (2.718...) raised to the power of <b>a</b> .	exp(2) → 7.389056099
floor( a )	double	double	Returns the largest whole number less than or equal to <b>a</b> .	floor(5.6) → 5.0 floor(5.0) → 5.0 floor(-5.6) → -6.0
log( a )	double	double	Returns the natural logarithm (base e) of <b>a</b> .	log(2.7183) → 1.0
max( a, b )	int	int	Returns the larger of <b>a</b> and <b>b</b> .	max(10, 20) → 20
	long	long	Same as above.	
	float	float	Same as above.	

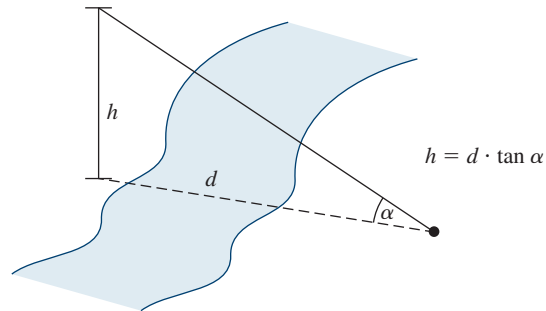
**Table 3.6** *Math* class methods for commonly used mathematical functions.—*continued*

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

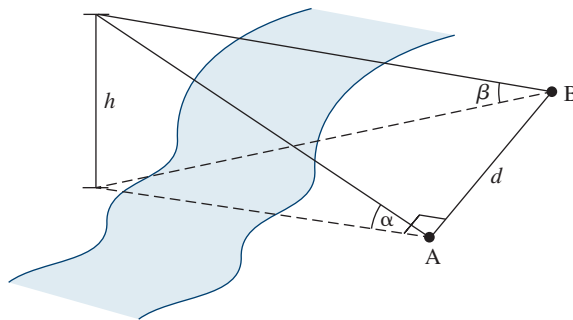
<sup>†</sup>All trigonometric functions are computed in radians.

Let's conclude this section with a sample program. Our Zen master Koan has thrown us the riddle of a day. Across the river we see a giant statue of Buddha. The Master wants us to determine the height of the statue using our mind's eye. Since our mind's eye is still very much in need of further polishing, we decided to use some trigonometry instead. If we can find the distance *d* to the statue, then we can use the tangent of angle

to determine the statue's height  $h$  as follows:



Unfortunately, there's no means for us to go across the river. Thus, the riddle. After a moment of deep meditation, it hit upon us that there's no need to go across the river. We can determine the statue's height by measuring angles from two points on this side of the river bank 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 Tower Height
File: Ch3BuddhaHeight.java
*/

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

```

**28 Chapter 3 Numerical Data**

```
class Ch3BuddhaHeight {  
  
    public static void main( String[] args ) {  
  
        double height;           //height of the clock tower  
        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  
  
        String inputStr;  
  
        //Get three input values  
        inputStr      = JOptionPane.showInputDialog(null,  
                                                    "Angle alpha (in degree):");  
        alpha         = Double.parseDouble(inputStr);  
        inputStr      = JOptionPane.showInputDialog(null,  
                                                    "Angle beta (in degree):");  
        beta          = Double.parseDouble(inputStr);  
  
        inputStr      = JOptionPane.showInputDialog(null,  
                                                    "Distance between points A and B (ft):");  
        distance      = Double.parseDouble(inputStr);  
  
        //compute the height of the tower  
        alphaRad = Math.toRadians(alpha);  
        betaRad  = Math.toRadians(beta);  
  
        height = ( distance * Math.sin(alphaRad) * Math.sin(betaRad) )  
                /  
                ( Math.sqrt( Math.sin(alphaRad + betaRad) *  
                             Math.sin(alphaRad + betaRad) ) );  
  
        DecimalFormat df = new DecimalFormat("0.000");  
  
        System.out.println("Estimating the height of the clock tower");  
        System.out.println("");  
        System.out.println("Angle at point A (degree):      "  
                            + df.format(alpha));  
        System.out.println("Angle at point B (degree):      "  
                            + df.format(beta));  
        System.out.println("Distance between A and B (ft): "  
                            + df.format(distance));  
  
        System.out.println("");  
        System.out.println("Estimated tower height (ft):  "  
                            + df.format(height));  
  
    }  
}
```

## Quick CHECK

1. What's wrong with the following?

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

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

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

### 3.8 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 a 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 which the class belongs to.

#### Helpful Reminder



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

#### Gregorian- Calendar

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

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

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

```
GregorainCalendar 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 confusion, we can use constants defined for months in the superclass `Calendar` (`GregorianCalendar`

**30 Chapter 3** Numerical Data

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

**Table 3.7** Constants defined in the `Calendar` class for retrieved different pieces of calendar/time information.

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

When the date and time is April 29, 2002 2:45 PM, and we run the `Ch3TestCalendar` program, we will see the result shown in Figure 3.11.

```
/*
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));  
  
    }  
}
```

```
Mon Apr 29 14:45:06 PDT 2002
```

```
YEAR:          2002  
MONTH:         3  
DATE:          29  
DAY_OF_YEAR:   119  
DAY_OF_MONTH:  29  
DAY_OF_WEEK:   2  
WEEK_OF_YEAR:  18  
WEEK_OF_MONTH: 5  
AM_PM:         1  
HOUR:          2  
HOUR_OF_DAY:   14  
MINUTE:        45
```

**Figure 3.11** Result of running the Ch3TestCalendar program at April 29, 2002 2:45 PM.

## 32 Chapter 3 Numerical Data

getTime

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

Notice also that we get only the numerical values when we retrieve the day of the week or month information. We can spell out the information by using the `SimpleDateFormat` class. Since the constructor of the `SimpleDateFormat` class accepts only the `Date` object, we need to first 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 which our Declaration of Independence was signed in Philadelphia (`Ch3IndependenceDay.java`):

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

SimpleDateFormat sdf = new SimpleDateFormat("EEEE");

JOptionPane.showMessageDialog("It's signed on "
    + sdf.format(independenceDay.getTime()));
```



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 4th, but the actual signing did not 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.nara.gov/exhall/charters/declaration/dechist.html> or <http://www.ushistory.org/declaration/>

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



```
/*
   Chapter 3 Sample Program: Find the Day of Week of Given Date
   File: Ch3FindDayOfWeek.java
*/

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

class Ch3FindDayOfWeek {

    public static void main( String[] args ) {

        String  inputStr;
        int     year, month, day;

        GregorianCalendar cal;
        SimpleDateFormat  sdf;

        inputStr = JOptionPane.showInputDialog( null, "Year:");
        year     = Integer.parseInt( inputStr );

        inputStr = JOptionPane.showInputDialog( null, "Month (0-11):");
        month    = Integer.parseInt( inputStr );

        inputStr = JOptionPane.showInputDialog( null, "Day:");
        day      = Integer.parseInt( inputStr );

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

        JOptionPane.showMessageDialog( null, "It was "
            + sdf.format( cal.getTime() ) );

    }
}
```

### 3.9 Sample Development

#### Loan Calculator

In this section, we will develop a simple loan calculator program. We will develop this program using an incremental development technique, which will develop the program in small incremental steps. We start out with a barebones 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

### 3.9 Sample Development—continued

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

Next time you buy a new TV or a stereo, watch out for those “0% down, 0% interest till 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, a 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 + 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 will now identify the classes we can use to implement the three tasks. First, we need an object to handle the input of three values. At this point, we have only learned about the **JOptionPane** class, so we will use it here. Second, we need an object to display the monthly and total payments. We can use either **JOptionPane** or **System.out**. Since we plan multiple lines of text for output, we will use **System.out**. Finally, we need to consider how we are going to compute the monthly and total payments. There are no objects in standard packages that will do the computation, so we have to write our own code to do the computation.

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

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

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, e.g., 0.01 for 1 percent

monthly rate. Once the monthly payment is derived, the total payment can be determined by multiplying the monthly payment and the number of months the payment is made. Since the formula includes exponentiation, we will have to use the `pow` method of the **Math** class.

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

program  
classes

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

The program diagram based on the classes listed in the design document is shown in Figure 3.12. 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.

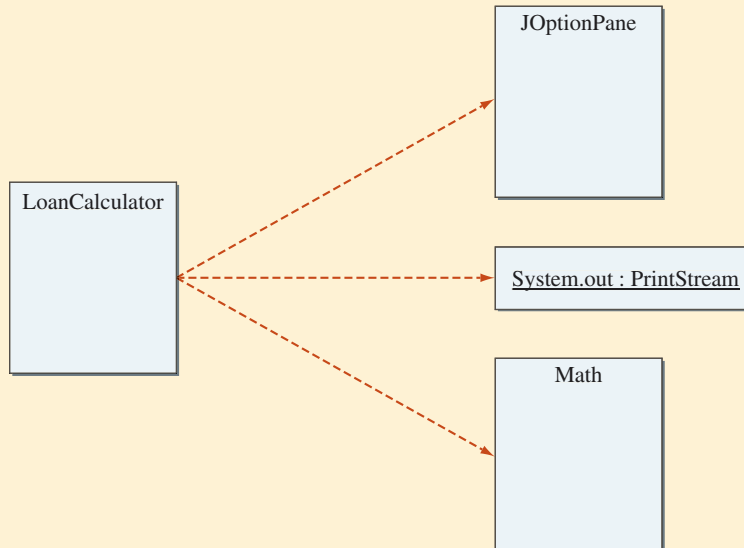
Before we can actually start our development, we must sketch the steps we will follow to implement the program. There are more than one possible sequence of steps to implement a program, and the number of possible sequences will increase as the program becomes more complex. For this program, we will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

develop-  
ment tasks

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 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 till the end. It's possible, but if we implement the computation part in the second incremental step, then we need to code some

## 3.9 Sample Development—continued



**Figure 3.12** The object diagram for the program **LoanCalculator**.

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

### Step 1 Development: Input Three Data Values

Step 1  
Design

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

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

Be aware that we need to convert the annual interest rate to the monthly interest rate and the input value loan period to the number of monthly payments to use the given formula. In this case, the conversion is very simple, but even if the conversion routines were more complicated, we must do the conversion. It is not acceptable to ask users to enter an input value that is unnatural to them. For example, people do not think of interest rates in fractional values such as 0.07. They think of interest in terms of percentages such as 7%. Computer programs work for humans, not the other way around. 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 converts to a negative number, the program should respond accordingly, such as by printing an error message. We do not possess enough skills to implement such a robust program yet, so we will make the following assumptions: (1) the input values are nonnegative numbers and (2) the loan period is a whole number.

One important objective of this step is to verify that the input values are read in correctly by the program. To verify this, we will use **System.out** to print out the values accepted by **JOptionPane**. This method of printing out the values just entered is called *echo printing*. Since we are going to use **System.out** in the final program, we will use it for echo printing in this step.

Here's our Step 1 program:

echo  
printing

Step 1  
Code

```
/*
Chapter 3 Sample Development: Loan Calculator (Step 1)
File: Step1/Ch3LoanCalculator.java
Step 1: Input Data Values
*/
import javax.swing.*;

class Ch3LoanCalculator
{
    public static void main (String[] args)
    {
        double  loanAmount,
               annualInterestRate;
        int     loanPeriod;
        String  inputStr;
        //get input values
```

**3.9 Sample Development**—continued

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

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

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

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

**Step 1 Test**

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

**Step 2 Development: Output Values****Step 2  
Design**

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

```
132.151 15858.1
```

We must label the output values so the user can tell what the numbers represent. In addition, we must display the input values with the computed result so it will not be

meaningless. Which of the two shown in Figure 3.13 do you think is more meaningful to him? The output format of this program will be with **<amount>**, **<annual interest rate>**, and others replaced by the actual figures.

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

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

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 3.13** 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 the 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 newly added portion surrounded by a rectangle and in white background:

**3.9 Sample Development**—continued

```
/*
Chapter 3 Sample Development: Loan Calculator (Step 2)
File: Step2/Ch3LoanCalculator.java
Step 2: Display the Result
*/
import javax.swing.*;
class Ch3LoanCalculator
{
    public static void main (String[] args)
    {
        double  loanAmount,
               annualInterestRate;

        double  monthlyPayment,
               totalPayment;

        int     loanPeriod;
        String  inputStr;

        //get input values
        inputStr      = JOptionPane.showInputDialog(null,
            "Loan Amount (Dollars+Cents):");
        loanAmount    = Double.parseDouble(inputStr);
        inputStr      = JOptionPane.showInputDialog(null,
            "Annual Interest Rate (e.g. 9.5):");
        annualInterestRate = Double.parseDouble(inputStr);
        inputStr      = JOptionPane.showInputDialog(null,
            "Loan Period - # of years:");
        loanPeriod    = Integer.parseInt(inputStr);

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

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



```

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

```

## Step 2 Test

To verify the output routine is working correctly, we run the program and verify the layout. Most likely, we have to run the program several times to finetune the arguments for the **printLine** 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 number of years. So we need to convert the annual interest rate to a monthly interest rate and the loan period to the number of monthly payments. The two input values are converted as

```

monthlyInterestRate = annualInterestRate / 100.0 /
MONTHS_IN_YEAR;

numberOfPayments    = loanPeriod * MONTHS_IN_YEAR;

```

where **MONTHS\_IN\_YEAR** is a symbolic constant with value **12**. Notice that we need to divide the input annual interest rate by 100 first because the formula for loan computation requires that the interest rate is a fractional value, for example, 0.01, but the input annual interest rate is entered as a percentage point, for example, 12.0. Please read exercise 23 on page 154 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;

```

**3.9 Sample Development**—continuedStep 3  
Code

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

```
/*
Chapter 3 Sample Development: Loan Calculator (Step 3)
File: Step3/Ch3LoanCalculator.java
Step 3: Compute the monthly and total payments
*/
import javax.swing.*;
class Ch3LoanCalculator
{
    public static void main (String[] args)
    {
        final int MONTHS_IN_YEAR = 12;

        double  loanAmount,
               annualInterestRate;

        double  monthlyPayment,
               totalPayment;

        double  monthlyInterestRate;

        int     loanPeriod;

        int     numberOfPayments;

        String  inputStr;

        //get input values
        inputStr      = JOptionPane.showInputDialog(null,
                                                    "Loan Amount (Dollars+Cents):");
        loanAmount    = Double.parseDouble(inputStr);
        inputStr      = JOptionPane.showInputDialog(null,
                                                    "Annual Interest Rate (e.g. 9.5):");
        annualInterestRate = Double.parseDouble(inputStr);
        inputStr      = JOptionPane.showInputDialog(null,
                                                    "Loan Period - # of years:");
        loanPeriod    = Integer.parseInt(inputStr);
    }
}
```

```

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

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

totalPayment = monthlyPayment * numberOfPayments;

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

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

```

**Step 3 Test**

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

Input			Output	
Loan Amount	Annual Interest Rate	Loan Period (in years)	Monthly Payment	Total Payment
10000	10	10	132.151	15858.1
15000	7	15	134.824	24268.4
10000	12	10	143.471	17216.5

### 3.9 Sample Development—continued

#### 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 program description. One of the necessary features of any nontrivial programs is the description of what the program does to 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 javax.swing.*;
import javax.text.*;
class Ch3LoanCalculator
{
    public static void main (String[] args)
    {
        final int MONTHS_IN_YEAR = 12;

        double loanAmount,
            annualInterestRate;

        double monthlyPayment,
            totalPayment;

        double monthlyInterestRate;

        int loanPeriod;

        int numberOfPayments;

        String inputStr;

        DecimalFormat df = new DecimalFormat("0.00");
        //describe the program
        System.out.println("This program computes the monthly and
            total");
    }
}
```

```
System.out.println("payments for a given loan amount, annual ");
System.out.println("interest rate, and loan period.");
System.out.println("Loan amount in dollars and cents, e.g.
    12345.50");
System.out.println("Annual interest rate in percentage, e.g.
    12.75");
System.out.println("Loan period in number of years, e.g. 15");
System.out.println("\n"); //skip two lines

//get input values
inputStr          = JOptionPane.showInputDialog(null,
    "Loan Amount (Dollars+Cents):");
loanAmount        = Double.parseDouble(inputStr);
inputStr          = JOptionPane.showInputDialog(null,
    "Annual Interest Rate (e.g. 9.5):");
annualInterestRate = Double.parseDouble(inputStr);
inputStr          = JOptionPane.showInputDialog(null,
    "Loan Period - # of years:");
loanPeriod        = Integer.parseInt(inputStr);

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

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

totalPayment = monthlyPayment * numberOfPayments;

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

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

### 3.9 Sample Development—continued

#### Step 4 Test

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

### Helpful Reminder



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

### 3.10 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

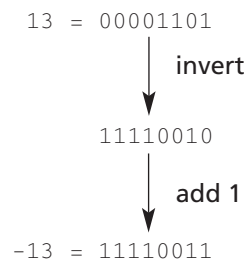
sign bit

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 left-most 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 using twos-complement representation. Notice that zero has only one representation.

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

**48 Chapter 3 Numerical Data**

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

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

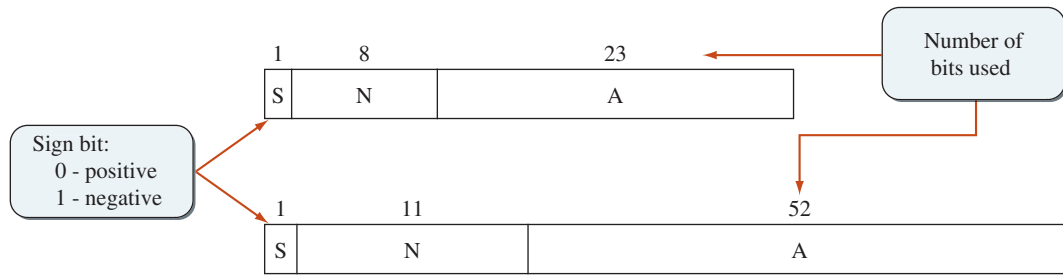
$$A \times 10^N$$

where  $A$  is a real number and  $N$  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

$$A \times 2^N$$

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



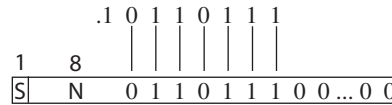
normalized fraction

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
.1010100	1.100111
.100011	.000000001
.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.

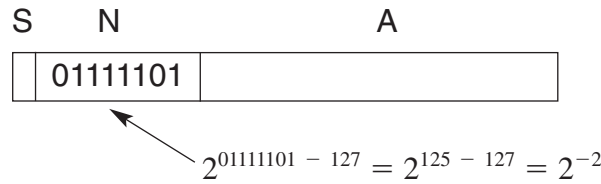




The sign bit *S* indicates the sign of a number, so *A* is stored in memory as an unsigned number. The integral exponent *N* can be negative or positive. Instead of using twos-complement for storing *N*, we use a format called *excess format*. The 8-bit exponent uses the excess-127 format, and the 11-bit 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  $2^{125-127} = 2^{-2}$ .



**Exercises**

1. Suppose we have the following declarations:

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

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

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

2. Suppose we have the following declarations:

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

**50 Chapter 3 Numerical Data**

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

- |  |  |
|--|--|
| a. <code>w = Math.pow(3, Math.pow(i, j));</code> | f. <code>m = n + i * j;</code>                           |
| b. <code>v = x / i;</code>                       | g. <code>n = k / (j * i)</code><br><code>* x + y;</code> |
| c. <code>w = Math.ceil(y) % k;</code>            | h. <code>i = i + 1;</code>                               |
| d. <code>n = (int) x / y * i / 2;</code>         | i. <code>w = float(x + i);</code>                        |
| e. <code>x = Math.sqrt(i*i - 4*j*k);</code>      | j. <code>x = x / i /</code><br><code>y / j;</code>       |

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 the following:

- The square root of  $B^2 + 4AC$  ( $A$  and  $C$  are distinct variables).
- The square root of  $X + 4Y^3$ .
- The cube root of the product of  $X$  and  $Y$ .
- The area  $\pi R^2$  of a circle.
- $\frac{\sin C}{\sin A}$ .

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

```
/*
   Program TestOutputBox
*/
import javabook.*;
class TestOutputBox
{
    public static void main (String args[])
    {
        MainWindow mainWindow;
        OutputBox outputBox;

        mainWindow = new MainWindow("Program TestOutputBox");
        outputBox = new OutputBox(mainWindow);
    }
}
```

```

mainWindow.setVisible( true );
outputBox.setVisible( true );

outputBox.println( "One" );
outputBox.print( "Two" );
outputBox.skipLine(1);

outputBox.print( "Three" );
outputBox.println( "Four" );
outputBox.skipLine(1);

outputBox.print( "Five" );
outputBox.println( "Six" );
}
}

```

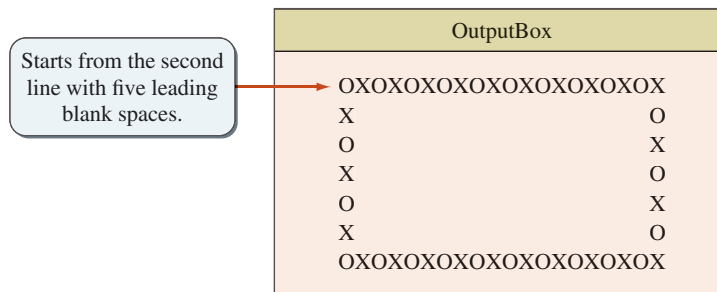
6. Determine the output of the following code:

```

int x, y;
x = 1;
y = 2;
messageBox.show( "The output is " + x + y );
messageBox.show( "The output is " + (x + y) );

```

7. Write an application that displays the following pattern in an **OutputBox** dialog:



Note: The **OutputBox** is not drawn to scale.

8. Write an application to convert centimeters (input) to feet and inches (output). Use **InputBox** for input and **OutputBox** for output. 1 inch = 2.54 centimeters.
9. Write an application that inputs temperature in Celsius and prints out the temperature in Fahrenheit. Use **InputBox** for input and **OutputBox** for output. The formula to convert Celsius to the equivalent Fahrenheit is

$$fahrenheit = 1.8 \times celsius + 32$$

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

$$\text{calories} = \text{bodyWeight} * 19;$$

Use an **OutputBox** dialog for display. Draw the object diagram of the program. (Note: We are not distinguishing between genders.)

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

$$\text{BMI} = \frac{w}{h^2}$$

where  $w$  is weight in kilograms and  $h$  is height in meters. A BMI of about 20 to 25 is considered "normal." Write an application that accepts weight and height (both integers) and outputs the BMI.

12. Your weight is actually the amount of gravitational attraction exerted on you by the earth. Since the moon's gravity is only 1/6 of the earth's gravity, on the moon you would weigh only 1/6 of what you weigh on the earth. Write an application that inputs the user's earth weight and outputs his/her weight on Mercury, Venus, Jupiter, and Saturn. Use the values in the table below:

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

13. When you say you are 18 years old, you are really saying that the earth has circled the sun eighteen times. Since other planets take less or more days than the 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 the 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/her age on Mercury, Venus, Jupiter,

and Saturn. The values for  $d$  are listed in the table below:

Planet	$d$ = Approximate Number of Earth Days for This Planet to Travel around the Sun
Mercury	88
Venus	225
Jupiter	4380
Saturn	10767

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

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

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

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

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

$$B^2 \geq 4AC$$

holds, so there will be real-number solutions for  $x$ .

15. Write 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, 3450 for \$34.50 and 70 for \$0.70. Use `InputBox` for input and `OutputBox` for output. Display the output in the following format:

```
Purchase Price:  $ 34.50
Amount Tendered: $ 40.00

Your change is:  $ 5.50

                    5 one-dollar bill(s)
                    2 quarter(s)

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).

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

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

where 5.99 is the cost per pound and 0.0725 is the sales tax. The letter f after 5.99 and 0.0725 designates that these two numbers are of type float. Display the result in the following manner:

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

  Total price: $ 1027.884
```

Draw the object diagram of the program.

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

$$\frac{P \left( 1 - \left( \frac{R}{100} \right)^{N+1} \right)}{1 - \left( \frac{R}{100} \right)}$$

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

18. Leonardo Fibonacci of Pisa was one of the greatest mathematicians of the Middle Ages. He is perhaps most famous for the Fibonacci sequence that can be applied to many diverse problems. One amusing application of the Fibonacci sequence is finding the growth rate of rabbits. Suppose a pair of rabbits matures in two 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 one year? (We assume here that no pairs die.) The table below shows the sequence for the first seven months. Notice that at the end of the second month, the first pair matures and bears its first offspring in the third month, making the total two pairs.

Month #	# of pairs
1	1
2	1
3	2
4	3
5	5
6	8
7	13

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

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

Write an application that accepts  $N$  and displays  $F_N$ . Note that the result of computation using the `Math` class is `double`. You need to display it as an integer.

19. Java<sup>2</sup> 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 2-lb bags, the medium 10 bags, and the small 5 bags. The cost of a large box is \$2.00, a medium box \$1.00, and a small box \$0.50. The order is shipped using the least number of boxes with the cheapest cost. For example, the order of 25 bags will be shipped in two boxes, one large and one small. Write an application that computes the total cost of an order. Use `InputBox` to accept the number of bags for an order and `OutputBox` to display the total cost including the cost of boxes. Display the output in the following format:

```
Number of Bags Ordered:  52 - $ 286.00
    Boxes Used:
                2 Large  - $4.00
                1 Medium - $1.00
                1 Small  - $0.50

    Your total cost is:  $ 291.50
```

Remember that you can format a double value  $X$  to two decimal places by the expression

```
Math.round( X * 100 ) / 100.0
```

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

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

where  $d$  is the distance between the two bodies and  $k$  is a positive real number called the *gravitational constant*. The gravitational constant  $k$  is approximately equal to  $6.67E-8$  dyne  $\text{cm}^2/\text{gm}^2$ . Write an application that accepts the mass for two bodies in grams and the distance between the two bodies in centimeters, and compute the force  $F$ . Use the appropriate format for the output. For your information, the force between the earth and the moon is  $1.984E25$  dynes. The mass of the earth is  $5.983E27$  grams, the mass of the moon is  $7.347E25$  grams, and the distance between the two is  $3.844E10$  centimeters.

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

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

where  $k$  is Ms. Latte's constant,  $C$  is the number of lines in the program that contain comments,  $T$  is the time spent (in minutes) by the programmer developing the program, and  $V$  is the number of lines in the program that contain nondescriptive variable names. Write an application to compute the program readability  $R$ . Ms. Latte's constant is  $2.5E2$  mocha lines<sup>2</sup>/min<sup>2</sup>. (Note: This is just for fun. Develop your own law using various functions from the `Math` class.)

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

$$y = ce^{kx}$$

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

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

Then we determine the value of  $k$  as

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

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

$$k = \frac{1}{20} \ln \left( \frac{2,000,000}{1,000,000} \right) \approx 0.03466$$

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

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

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.

23. In Section 3.9, we use the formula

$$mr = \frac{ar}{12}$$

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



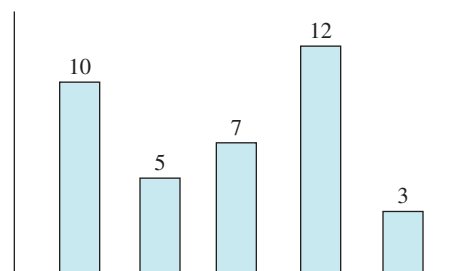
fractional value such as 0.083). This annual interest rate  $ar$  is called the *stated annual interest rate* to distinguish it from the *effective annual interest rate*, which is the true cost of a loan. If the stated annual interest rate is 9%, for example, then the effective annual interest rate is actually 9.38%. Naturally, the rate the financial institutions advertise more prominently is the stated interest rate. The loan calculator program in Section 3.7 treats the annual interest rate the user enters as the stated annual interest rate. If the input is the effective annual interest rate, then we compute the monthly rate as

$$mr = (1 + ear)^{\frac{1}{12}} - 1$$

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 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 the loan amount of 1, 10, and 50 million dollars with the loan period of 10, 20, and 30 years and the annual interest rate of 0.07, 0.10, and 0.18, 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](http://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.

24. Using a `Turtle` object from the `galapagos` package (see exercise 27 on page 88), draw three rectangles. Use an `InputBox` to accept the width and the length of the smallest rectangle from the user. The middle and the largest rectangles are 40 and 80 percent larger than the smallest rectangle. The `galapagos` package and its documentation are available at [www.drcaffeine.com/packages](http://www.drcaffeine.com/packages).
25. Write a program that draws a bar chart using a `Turtle` object (see exercise 27 on page 88). 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.