CHAPTER 2

Programming Building Blocks—Java Basics

CHAPTER CONTENTS

Introduction

If you boil it down to the basics, a program has two elements: instructions and data. The instructions tell the CPU what to do with the data. Typically, a program's structure will consist of the following operations:

1. Input the data.
2. Perform some processing on the data.
3. Output the results.

The data used by a program can come from a variety of sources. The user can enter data from the keyboard, as happens when you type a new document into a word processor. The program can read the data from a file, as happens when you load an existing document into the word processor. Or the program can generate the data randomly, as happens when a computer card game deals hands. Finally, some data is already known; for example, the number of hours in a day is 24, the number of days in December is 31, and the value of pi is 3.14159. This type of data is constant. The Java language provides a syntax for describing a program's data using keywords, symbolic names, and data types.

The data may be different in each execution of the program, but the instructions stay the same. In a word processor, the words (data) are different from document to document, but the operation (instructions) of the word processor remains the same. When a line becomes full, for example, the word processor automatically wraps to the next line. It doesn't matter which words are on the line, only that the line is full. When you select a word and change the font to bold, it doesn't matter which word you select; it will become bold. Thus, a program's instructions (its algorithm) must be written to correctly handle any data it may receive.

In Chapter 1, we discussed the types of operations that the computer can perform: input and output of data and various operations related to processing data, such as arithmetic calculations, comparisons of data and subsequent changes to the flow of control, and movement of data from one location in memory to another. We will write our programs by translating our algorithms into these basic operations.

In this chapter, we'll look at basic Java syntax for defining the data to be used in the program, performing calculations on that data, and outputting program results to the screen.

2.1 Java Application Structure

Every Java program consists of at least one class. It is impossible to write a Java program that doesn't use classes. As we said in Chapter 1, classes

```
1 /*  An application shell
2     Anderson, Franceschi
3 */
4 public class ShellApplication
5 {
6    public static void main( String [ ] args ) //required
7    {
8       // write your code here
9    }
10 }
```

EXAMPLE 2.1 A Shell for a Java Application

describe a logical entity that has data as well as methods (the instructions) to manipulate that data. An object is a physical instantiation of the class that contains specific data. We'll begin to cover classes in detail in the next chapter. For now, we'll just say that your source code should take the form of the shell code in Example 2.1.

In Example 2.1, the numbers to the left of each line are not part of the program code; they are included here for your convenience. IDEs typically allow you to display line numbers.

From application to application, the name of the class, *ShellApplication*, will change, because you will want to name your class something meaningful that reflects its function. Each Java source code file must have the same name as the class name with a *.java* extension. In this case, the source file must be *ShellApplication.java*. Whatever name you select for a class must comply with the Java syntax for identifiers.

Java **identifiers** are symbolic names that you assign to classes, methods, and data. Identifiers must start with a **Java letter** and may contain any combination of letters and digits, but no spaces. A Java letter is any character in the range *a–z* or *A–Z*, the underscore (_), or the dollar sign ($), as well as many Unicode characters that are used as letters in other languages. Digits are any character between 0 and 9. The length of an identifier is essentially unlimited. Identifier names are case-sensitive, so *Number1* and *number1* are considered to be different identifiers.

In addition, none of Java's **reserved words** can be used as identifiers. These reserved words, which are listed in Appendix A, consist of keywords used in Java instructions, as well as three special data values: *true*, *false*, and *null*. Given that Java identifiers are case-sensitive, note that it is legal to use *True* or *TRUE* as identifiers, but *true* is not a legal variable name. Table 2.1 lists the rules for creating Java identifiers.

TABLE 2.1 Rules for Creating Identifiers

Java Identifiers
▪ Must start with a Java letter (*A–Z*, *a–z*, _, $, or many Unicode characters)
▪ Can contain an almost unlimited number of letters and/or digits (0–9)
▪ Cannot contain spaces
▪ Are case-sensitive
▪ Cannot be a Java reserved word

The shell code in Example 2.1 uses three identifiers: *ShellApplication*, *main*, and *args*. The remainder of Example 2.1 consists of comments, Java keywords, and required punctuation.

The basic building block of a Java program is the **statement**. A statement is terminated with a semicolon and can span several lines.

SOFTWARE ENGINEERING TIP

Liberal use of white space makes your program more readable. It is good programming style to surround identifiers, operands, and operators with spaces and to skip lines between logical sections of the program.

Any amount of **white space** is permitted between identifiers, Java keywords, operands, operators, and literals. White space characters are the space, tab, newline, and carriage return. Liberal use of white space makes your program more readable. It is good programming style to surround identifiers, operands, and operators with spaces and to skip lines between logical sections of the program.

A **block**, which consists of 0, 1, or more statements, starts with a left curly brace ({) and ends with a right curly brace (}). Blocks are required for class and method definitions and can be used anywhere else in the program that a statement is legal. Example 2.1 has two blocks: the class definition (lines 5 through 10) and the *main* method definition (lines 7 through 9). As you can see, nesting blocks within blocks is perfectly legal. The *main* block is nested completely within the class definition block.

SOFTWARE ENGINEERING TIP

Include a block comment at the beginning of each source file that identifies the author of the program and briefly describes the function of the program.

Comments document the operation of the program and are notes to yourself and to other programmers who read your code. Comments are not compiled and can be coded in two ways. **Block comments** can span several lines; they begin with a forward slash-asterisk (/*) and end with an asterisk-forward slash (*/). Everything between the /* and */ is ignored by the compiler. Note that there are no spaces between the asterisk and forward slash.

Lines 1–3 in Example 2.1 are block comments and illustrate the good software engineering practice of providing at the beginning of your source code a few comments that identify yourself as the author and briefly describe what the program does.

The second way to include comments in your code is to precede the comment with two forward slashes (//). There are no spaces between the forward slashes. The compiler ignores everything from the two forward slashes to the end of the line. In Example 2.1, the compiler ignores all of line 8, but only the part of line 6 after the two forward slashes.

Let's look at an example to get a sense of what a simple program looks like and to get a feel for how a program operates. Example 2.2 calculates the area of a circle.

```
1 /* Calculate the area of a circle
2    Anderson, Franceschi
3 */
4
5 public class AreaOfCircle
6 {
7   public static void main( String [] args )
8   {
9      // define the data we know
10     final double PI = 3.14159;
11
12     // define other data we will use
13     double radius;
14     double area;
15
16     // give radius a value
17     radius = 3.5;
18
19     // perform the calculation
20     area = PI * radius * radius;
21
22     // output the results
23     System.out.println( "The area of the circle is " + area );
24  }
25 }
```

Example 2.2 Calculating the Area of a Circle

Figure 2.1a
Output from Example 2.2 with a Radius of 3.5

```
The area of the circle is 38.4844775
```

Figure 2.1b
Output from Example 2.2 with a Radius of 20

```
The area of the circle is 1256.636
```

Figure 2.1a shows the output when the program is run with a radius of 3.5. To calculate the area of a circle with a different radius, replace the value 3.5 in line 17 with the new radius value. For example, to calculate the area of a circle with a radius of 20, change line 17 to

```
radius = 20;
```

Then recompile the program and run it again. Figure 2.1b shows the output for a radius of 20.

You can see that Example 2.2 has the basic elements that we saw in the *ShellApplication* (Example 2.1). We have added some statements in lines 9 to 23 that do the work of the program. First we identify the data we will need. To calculate the area of a circle, we use the formula (πr^2). We know the value of π (3.14159), so we store that value in a memory location we name PI (line 10). We also need places in memory to hold the radius and the area. We name these locations in lines 13 and 14. In line 17 we give the radius a value; here we have chosen 3.5.

Now we're ready to calculate the area. We want this program to output correct results with any radius, so we need to write the algorithm of the program using the formula for calculating a circle's area given above. Java provides arithmetic operators for performing calculations. We use Java's multiplication operator (*) in line 20 to multiply PI times the radius times the radius and store the result into the memory location we named *area*. Now we're ready to output the result. On line 23, we write a message that includes the *area* value we calculated.

2.2 Data Types, Variables, and Constants

In Example 2.2, we used as data the value of PI and the radius, and we calculated the area of the circle. For each of these values, we assigned a name. We also used the Java keyword *double*, which defines the **data type** of the data. The keyword *double* means that the value will be a floating-point number.

Java allows you to refer to the data in a program by defining **variables**, which are named locations in memory where you can store values. A variable can store one data value at a time, but that value might change as the program executes, and it might change from one execution of the program to the next. The real advantage of using variables is that you can name a variable, assign it a value, and subsequently refer to the name of the variable in an expression rather than hard coding the specific value.

When we use a named variable, we need to tell the compiler which kind of data we will store in the variable. We do this by giving a data type for each variable.

Java supports eight primitive data types: ***byte***, *short*, *int*, *long*, *float*, *double*, *char*, and *boolean*. They are called primitive data types because they are part of the core Java language.

The data type you specify for a variable tells the compiler how much memory to allocate and the format in which to store the data. For example, if you specify that a data item is an *int*, then the compiler will allocate four bytes of memory for it and store its value as a 32-bit signed binary number. If, however, you specify that a data item is a *double* (a double-precision floating-point number), then the compiler will allocate 8 bytes of memory and store its value as an IEEE 754 floating-point number.

Once you declare a data type for a data item, the compiler will monitor your use of that data item. If you attempt to perform operations that are not allowed for that type or are not compatible with that type, the compiler will generate an error. Because the Java compiler monitors the operations on each data item, Java is called a **strongly typed language**.

Take care in selecting identifiers for your programs. The identifiers should be meaningful and should reflect the data that will be stored in a variable,

SOFTWARE ENGINEERING TIP

When selecting identifiers, choose meaningful names that reflect the use of the identifier in the program; this will make your code self-documented. Use as many characters as necessary to make the identifier clear, but avoid extremely long identifiers. Also, for clarity in your program logic, avoid identifiers that resemble Java keywords.

the concept encapsulated by a class, or the function of a method. For example, the identifier *age* clearly indicates that the variable will hold the age of a person. When you select meaningful variable names, the logic of your program is more easily understood, and you are less likely to introduce errors. Sometimes, it may be necessary to create a long identifier in order to clearly indicate its use, for example, *numberOfStudentsWhoPassedCS1*. Although the length of identifiers is essentially unlimited, avoid creating extremely long identifiers because they are more cumbersome to use. Also, the longer the identifier, the more likely you are to make typos when entering the identifier into your program. Finally, although it is legal to use identifiers, such as *TRUE*, which differ from Java keywords only in case, it isn't a good idea because they easily can be confused with Java keywords, making the program logic less clear.

2.2.1 Declaring Variables

Every variable must be given a name and a data type before it can be used. This is called **declaring a variable**.

The syntax for declaring a variable is:

```
dataType identifier; // this declares one variable
```

or

```
dataType identifier1, identifier2, ...; // this declares multiple
                                        // variables of the same
                                        // data type
```

Note that a comma follows each identifier in the list except the last identifier, which is followed by a semicolon.

SOFTWARE ENGINEERING TIP

Begin variable names with a lowercase letter. If the variable name consists of more than one word, begin each word after the first with a capital letter. Avoid underscores in variable names, and do not begin a variable name with a dollar sign.

By convention, the identifiers for variable names start with a lowercase letter. If the variable name consists of more than one word, then each word after the first should begin with a capital letter. For example, these identifiers are conventional Java variable names: *number1*, *highScore*, *booksToRead*, *ageInYears*, and *xAxis*. Underscores conventionally are not used in variable names; they are reserved for the identifiers of constants, as we shall discuss later in the chapter. Similarly, do not use dollar signs to begin variable names. The dollar sign is reserved for the first letter of programmatically generated variable names—that is, variable names generated by software, not people. Although this may sound arbitrary now, the value of following these conventions will become clearer as you gain more experience in Java and your programs become more complex.

2.2.2 Integer Data Types

An integer data type is one that evaluates to a positive or negative whole number. Java provides four integer data types, *int*, *short*, *long*, and *byte*.

The *int*, *short*, *long*, and *byte* types differ in the number of bytes of memory allocated to store each type and, therefore, the maximum and minimum values that can be stored in a variable of that type. All of Java's integer types are signed, meaning that they can be positive or negative; the high-order, or leftmost, bit is reserved for the sign.

Table 2.2 summarizes the integer data types, their sizes in memory, and their maximum and minimum values.

In most applications, the *int* type will be sufficient for your needs, since it can store positive and negative numbers up into the 2 billion range. The *short* and *byte* data types typically are used only when memory space is critical, and the *long* data type is needed only for data values larger than 2 billion.

Let's look at some examples of integer variable declarations. Note that the variable names clearly indicate the data that the variables will hold.

```
int testGrade;
int numPlayers, highScore, diceRoll;
short xCoordinate, yCoordinate;
long cityPopulation;
byte ageInYears;
```

2.2.3 Floating-Point Data Types

Floating-point data types store numbers with fractional parts. Java supports two floating-point data types: the single-precision *float* and the double-precision *double*.

TABLE 2.2 Integer Data Types

Integer Data Type	Size in Bytes	Minimum Value	Maximum Value
byte	1	−128	127
short	2	−32,768	32,767
int	4	−2,147,483,648	2,147,483,647
long	8	−9,223,372,036,854,775,808	9,223,372,036,854,775,807

REFERENCE POINT

Floating-point numbers are stored using the IEEE 754 standard, which is discussed in Appendix E.

The two types differ in the amount of memory allocated and the size of the number that can be represented. The single-precision type (*float*) is stored in 32 bits, while the double-precision type (*double*) is stored in 64 bits. *Floats* and *doubles* can be positive or negative.

Table 2.3 summarizes Java's floating-point data types, their sizes in memory, and their maximum and minimum positive nonzero values.

Because of its greater precision, the *double* data type is usually preferred over the *float* data type. However, for calculations not requiring such precision, *floats* are often used because they require less memory.

Although integers can be stored as *doubles* or *floats*, it isn't advisable to do so because floating-point numbers require more processing time for calculations.

Let's look at a few examples of floating-point variable declarations:

```
float salesTax;
double interestRate;
double paycheck, sumSalaries;
```

2.2.4 Character Data Type

REFERENCE POINT

The encoding of ASCII and Unicode characters is discussed in Appendix C.

The *char* data type stores one Unicode character. Because Unicode characters are encoded as unsigned numbers using 16 bits, a *char* variable is stored in two bytes of memory.

Table 2.4 shows the size of the *char* data type, as well as the minimum and maximum values. The maximum value is the unsigned hexadecimal number *FFFF*, which is reserved as a special code for "not a character."

TABLE 2.3 Floating-point Data Types

Floating-point Data Type	Size in Bytes	Minimum Positive Nonzero Value	Maximum Value
float	4	1.4E-45	3.4028235E38
double	8	4.9E-324	1.7976931348623157E308

TABLE 2.4 The Character Data Type

Character Data Type	Size in Bytes	Minimum Value	Maximum Value
char	2	The character encoded as *0000*, the *null* character	The value *FFFF*, which is a special code for "not a character"

Obviously, since the *char* data type can store only a single character, such as a *K*, a *char* variable is not useful for storing names, titles, or other text data. For text data, Java provides a *String* class, which we'll discuss later in this chapter.

Here are a few declarations of *char* variables:

```
char finalGrade;
char middleInitial;
char newline, tab, doubleQuotes;
```

2.2.5 Boolean Data Type

The *boolean* data type can store only two values, which are expressed using the Java reserved words *true* and *false*, as shown in Table 2.5.

Booleans are typically used for decision making and for controlling the order of execution of a program.

Here are examples of declarations of *boolean* variables:

```
boolean isEmpty;
boolean passed, failed;
```

2.2.6 The Assignment Operator, Initial Values, and Literals

When you declare a variable, you can also assign an initial value to the data. To do that, use the **assignment operator** (=) with the following syntax:

```
dataType variableName = initialValue;
```

This statement is read as "*variableName* **gets** *initialValue*"

or

```
dataType variable1 = initialValue1, variable2 = initialValue2;
```

Notice that assignment is right to left. The initial value is assigned to the variable.

TABLE 2.5 The *boolean* Data Type

boolean Data Type	Possible Values
boolean	*true*
	false

COMMON ERROR TRAP

Although Unicode characters occupy two bytes in memory, they still represent a single character. Therefore, the literal must also represent only one character.

One way to specify the initial value is by using a **literal value**. In the following statement, the value *100* is an *int* literal value, which is assigned to the variable *testGrade*.

```
int testGrade = 100;
```

Table 2.6 summarizes the legal characters in literals for all primitive data types.

Notice in Table 2.6 under the literal format for *char*, that *\n* and *\t* can be used to format output. We'll discuss these and other escape sequences in the next section of this chapter.

TABLE 2.6 Literal Formats for Java Data Types

Data Type	Literal Format
int, short, byte	Optional initial sign (+ or −) followed by digits 0–9 in any combination. A literal in this format is an *int* literal; however, an *int* literal may be assigned to a *byte* or *short* variable if the literal is a legal value for the assigned data type. An integer literal that begins with a 0 digit is considered to be an octal number (base 8) and the remaining digits must be 0–7. An integer literal that begins with 0x is considered to be a hexadecimal number (base 16) and the remaining digits must be 0–F.
long	Optional initial sign (+ or −) followed by digits 0–9 in any combination, terminated with an *L* or *l*. It's preferable to use the capital *L*, because the lowercase *l* can be confused with the number *1*. An integer literal that begins with a 0 digit is considered to be an octal number (base 8) and the remaining digits must be 0–7. An integer literal that begins with 0x is considered to be a hexadecimal number (base 16) and the remaining digits must be 0–F.
float	Optional initial sign (+ or −) followed by a floating-point number in fixed or scientific format, terminated by an *F* or *f*.
double	Optional initial sign (+ or −) followed by a floating-point number in fixed or scientific format.
char	▪ Any printable character enclosed in single quotes. ▪ A decimal value from 0 to 65,535. ▪ '\m', where \m is an escape sequence. For example, '\n' represents a newline, and '\t' represents a tab character.
boolean	*true* or *false*

Example 2.3 shows a complete program illustrating variable declarations, specifying a literal for the initial value of each.

```
1  /* Variables Class
2     Anderson, Franceschi
3  */
4
5  public class Variables
6  {
7   public static void main( String [ ] args )
8   {
9     // This example shows how to declare and initialize variables
10
11    int testGrade = 100;
12    long cityPopulation = 425612340L;
13    byte ageInYears = 19;
14
15    float  salesTax = .05F;
16    double interestRate = 0.725;
17    double avogadroNumber = +6.022E23;
18    // avogadroNumber is represented in scientific notation;
19    //     its value is 6.022 x 10 to the power 23
20
21    char finalGrade = 'A';
22    boolean isEmpty = true;
23
24    System.out.println( "testGrade is " + testGrade );
25    System.out.println( "cityPopulation is " + cityPopulation );
26    System.out.println( "ageInYears is " + ageInYears );
27    System.out.println( "salesTax is " + salesTax );
28    System.out.println( "interestRate is " + interestRate );
29    System.out.println( "avogadroNumber is " + avogadroNumber );
30    System.out.println( "finalGrade is " + finalGrade );
31    System.out.println( "isEmpty is " + isEmpty );
32  }
33 }
```

EXAMPLE 2.3 Declaring and Initializing Variables

Line 9 shows a single-line comment. Line 17 declares a *double* variable named *avogadroNumber* and initializes it with its value in scientific notation. The Avogadro number represents the number of elementary particles in one mole of any substance.

Figure 2.2 shows the output of Example 2.3.

Figure 2.2
Output of Example 2.3

```
testGrade is 100
cityPopulation is 425612340
ageInYears is 19
salesTax is 0.05
interestRate is 0.725
avogadroNumber is 6.022E23
finalGrade is A
isEmpty is true
```

Another way to specify an initial value for a variable is to assign the variable the value of another variable, using this syntax:

```
dataType variable2 = variable1;
```

Two things need to be true for this assignment to work:

- *variable1* needs to be declared and assigned a value before this statement appears in the source code.
- *variable1* and *variable2* need to be compatible data types; in other words, the precision of *variable1* must be lower than or equal to that of *variable2*.

For example, in these statements:

```
boolean isPassingGrade = true;
boolean isPromoted = isPassingGrade;
```

isPassingGrade is given an initial value of *true*. Then *isPromoted* is assigned the value already given to *isPassingGrade*. Thus, *isPromoted* is also assigned the initial value *true*. If *isPassingGrade* were assigned the initial value *false*, then *isPromoted* would also be assigned the initial value *false*.

And in these statements:

```
float salesTax = .05f;
double taxRate = salesTax;
```

the initial value of .05 is assigned to *salesTax* and then to *taxRate*. It's legal to assign a *float* value to a *double*, because all values that can be stored as *floats* are also valid *double* values. However, these statements are *not* valid:

```
double taxRate = .05;
float salesTax = taxRate; // invalid; float is lower precision
```

Even though .05 is a valid *float* value, the compiler will generate a "`possible loss of precision`" error.

Similarly, you can assign a lower-precision integer value to a higher-precision integer variable.

Table 2.7 summarizes compatible data types; a variable or literal of any type in the right column can be assigned to a variable of the data type in the left column.

Variables need to be declared before they can be used in your program, but be careful to declare each variable only once; that is, specify the data type of the variable only the first time that variable is used in the program. If you attempt to declare a variable that has already been declared, as in the following statements:

```
double twoCents;
double twoCents = 2; // incorrect, second declaration of twoCents
```

you will receive a compiler error similar to the following:

```
twoCents is already defined
```

TABLE 2.7 Valid Data Types for Assignment

Data Type	Compatible Data Types
byte	*byte*
short	*byte, short*
int	*byte, short, int, char*
long	*byte, short, int, char, long*
float	*byte, short, int, char, long, float*
double	*byte, short, int, char, long, float, double*
boolean	*boolean*
char	*char*

COMMON ERROR TRAP

Declare each variable only once, the first time the variable is used. After the variable has been declared, its data type cannot be changed.

Similarly, once you have declared a variable, you cannot change its data type. Thus, these statements:

```
double cashInHand;
int cashInHand; // incorrect, data type cannot be changed
```

will generate a compiler error similar to the following:

```
cashInHand is already defined
```

CODE IN ACTION

On the CD-ROM included with this book, you will find a Flash movie showing a step-by-step illustration of declaring variables and assigning initial values. Click on the link for Chapter 2 to view the movie.

2.2.7 *String* Literals and Escape Sequences

In addition to literals for all the primitive data types, Java also supports *String* literals. *Strings* are objects in Java, and we will discuss them in greater depth in Chapter 3.

A ***String* literal** is a sequence of characters enclosed by double quotes. One set of quotes "opens" the *String* literal and the second set of quotes "closes" the literal. For example, these are all *String* literals:

```
"Hello"
"Hello world"
"The value of x is "
```

We used a *String* literal in our first program in Chapter 1 in this statement:

```
System.out.println( "Programming is not a spectator sport!" );
```

We also used *String* literals in output statements in Example 2.3 to label the data that we printed:

```
System.out.println( "The area of the circle is " + area );
```

The + operator is the ***String* concatenation operator**. Among other uses, the concatenation operator allows us to print primitive data types along with *Strings*. We'll discuss the concatenation operator in more detail in Chapter 3.

String literals cannot extend over more than one line. If the compiler finds a newline character in the middle of your *String* literal, it will generate a compiler error. For example, the following statement is not valid:

```
System.out.println( "Never pass a water fountain
           without taking a drink." );
```

In fact, that statement will generate several compiler errors:

```
StringTest.java:9: unclosed string literal
      System.out.println( "Never pass a water fountain
                          ^
StringTest.java:9: ';' expected
      System.out.println( "Never pass a water fountain
                                                      ^
StringTest.java:10: ';' expected
           without taking a drink." );
                           ^
StringTest.java:10: unclosed string literal
           without taking a drink." );
                                  ^

StringTest.java:10: not a statement
           without taking a drink." );
                                  ^
5 errors
```

If you have a long *String* to print, break it into several strings and use the concatenation operator. This statement is a correction of the invalid statement above:

```
System.out.println( "Never pass a water fountain,"
                  + " without taking a drink." );
```

Another common programming error is omitting the closing quotes. Be sure that all open quotes have matching closing quotes on the same line.

Now that we know that quotes open and close *String* literals, how can we define a literal that includes quotes? This statement

```
System.out.println( "She said, "Java is fun"" ); // illegal quotes
                                                 // within literal
```

generates this compiler error:

```
StringTest.java:24: ')' expected
      System.out.println( "She said, "Java is fun"" ); // illegal quotes
                                     ^
StringTest.java:24: ';' expected
      System.out.println( "She said, "Java is fun"" ); // illegal quotes
                                                   ^
2 errors
```

And since *String* literals can't extend over two lines, how can we create a *String* literal that includes a newline character? Java solves these problems

SOFTWARE ENGINEERING TIP

Add a space to the end of a *String* literal before concatenating a value for more readable output.

COMMON ERROR TRAP

All open quotes for a *String* literal should be matched with a set of closing quotes, and the closing quotes must appear before the line ends.

TABLE 2.8 Java Escape Sequences

Character	Escape Sequence
newline	\n
tab	\t
double quotes	\"
single quote	\'
backslash	\\
backspace	\b
carriage return	\r
form feed	\f

by providing a set of escape sequences that can be used to include a special character within *String* and *char* literals. The escape sequences *\n, \t, \b, \r,* and *\f* are nonprintable characters. Table 2.8 lists the Java escape sequences.

In Example 2.4, we see how escape sequences can be used in *Strings*.

```
1  /*  Literals Class
2      Anderson, Franceschi
3  */
4
5  public class Literals
6  {
7    public static void main( String [ ] args )
8    {
9      System.out.println( "One potato\nTwo potatoes\n" );
10     System.out.println( "\tTabs can make the output easier to read"  );
11     System.out.println( "She said, \"Java is fun\"" );
12   }
13 }
```

EXAMPLE 2.4 Using Escape Sequences

Figure 2.3 shows the output of Example 2.4. Line 9 shows how *\n* causes the remainder of the literal to be printed on the next line. The tab character, *\t*,

```
One potato
Two potatoes

        Tabs can make the output easier to read
She said, "Java is fun"
```

Figure 2.3
Output of Example 2.4

used in line 10, will cause the literal that follows it to be indented one tab stop when output. Line 11 outputs a sentence with embedded double quotes; the embedded double quotes are printed with the escape sequence \".

2.2.8 Constants

Sometimes you know the value of a data item, and you know that its value will not (and should not) change during program execution, nor is it likely to change from one execution of the program to another. In this case, it is a good software engineering practice to define that data item as a **constant.**

Defining constants uses the same syntax as declaring variables, except that the data type is preceded by the keyword *final*.

```
final dataType CONSTANT_IDENTIFIER = assignedValue;
```

Assigning a value is optional when the constant is defined, but you must assign a value before the constant is used in the program. Also, once the constant has been assigned a value, its value cannot be changed (reassigned) later in the program. Any attempt by your program to change the value of a constant will generate the following compiler error:

```
cannot assign a value to final variable
```

Think of this as a service of the compiler in preventing your program from unintentionally corrupting its data.

By convention, *CONSTANT_IDENTIFIER* consists of all capital letters, and embedded words are separated by an underscore. This makes constants stand

out in the code and easy to identify as constants. Also, constants are usually defined at the top of a program where their values can be seen easily.

Example 2.5 shows how to use constants in a program.

```
1 /* Constants Class
2    Anderson, Franceschi
3 */
4
5 public class Constants
6 {
7   public static void main( String [ ] args )
8   {
9     final char ZORRO = 'Z';
10    final double PI = 3.14159;
11    final int DAYS_IN_LEAP_YEAR = 366, DAYS_IN_NON_LEAP_YEAR = 365;
12
13    System.out.println( "The value of constant ZORRO is " + ZORRO );
14    System.out.println( "The value of constant PI is " + PI );
15    System.out.println( "The number of days in a leap year is "
16                          + DAYS_IN_LEAP_YEAR );
17    System.out.println( "The number of days in a non-leap year is "
18                          + DAYS_IN_NON_LEAP_YEAR );
19
20    // PI = 3.14;
21    // The statement above would generate a compiler error
22    // You cannot change the value of a constant
23  }
24 }
```

EXAMPLE 2.5 Using Constants

SOFTWARE ENGINEERING TIP

Use all capital letters for a constant's identifier; separate words with an underscore (_). Declare constants at the top of the program so their value can be seen easily.

SOFTWARE ENGINEERING TIP

Declare as a constant any data that should not change during program execution. The compiler will then flag any attempts by your program to change the value of the constant, thus preventing any unintentional corruption of the data.

Lines 9, 10, and 11 define four constants. On line 11, note that both *DAYS_IN_LEAP_YEAR* and *DAYS_IN_NON_LEAP_YEAR* are constants. You don't need to repeat the keyword *final* to define two (or more) constants of the same data types. Lines 13 to 18 output the values of the four constants. If line 20 were not commented out, it would generate a compiler error because once a constant is assigned a value, its value cannot be changed. Figure 2.4 shows the output of Example 2.5.

Constants can make your code more readable: PI is more meaningful than 3.14159 when used inside an arithmetic expression. Another advantage of using constants is to keep programmers from making logic errors: Let's say

```
The value of constant ZORRO is Z
The value of constant PI is 3.14159
The number of days in a leap year is 366
The number of days in a non-leap year is 365
```

Figure 2.4
Output of Example 2.5

we set a constant to a particular value and it is used at various places throughout the code (for instance, a constant representing a tax rate); we then discover that the value of that constant needs to be changed. All we have to do is make the change in one place, most likely at the beginning of the code. If we had to change the value at many places throughout the code, that could very well result in logic errors or typos.

Skill Practice
with these end-of-chapter questions

2.6.1 Multiple Choice

Questions 1, 2

2.6.2 Reading and Understanding Code

Questions 4, 5, 6

2.6.3 Fill In the Code

Questions 23, 24, 25, 26

2.6.4 Identifying Errors in Code

Questions 33, 34, 38, 39

2.6.5 Debugging Area

Questions 40, 41

2.6.6 Write a Short Program

Question 46

2.6.8 Technical Writing

Question 52

2.3 Expressions and Arithmetic Operators

2.3.1 The Assignment Operator and Expressions

In a previous section, we mentioned using the assignment operator to assign initial values to variables and constants. Now let's look at the assignment operator in more detail.

The syntax for the assignment operator is:

```
target = expression;
```

An expression consists of operators and operands that evaluate to a single value. The value of the expression is then assigned to *target* (*target* gets *expression*), which must be a variable or constant having a data type compatible with the value of the expression.

If *target* is a variable, the value of the expression replaces any previous value the variable was holding. For example, let's look at these instructions:

```
int numberOfPlayers = 10; // numberOfPlayers value is 10
numberOfPlayers = 8;      // numberOfPlayers value is now 8
```

The first instruction declares an *int* named *numberOfPlayers*. This allocates four bytes in memory to a variable named *numberOfPlayers* and stores the value 10 in that variable. Then, the second statement changes the value stored in the variable *numberOfPlayers* to 8. The previous value, 10, is discarded.

An expression can be a single variable name or a literal of any type, in which case, the value of the expression is simply the value of the variable or the literal. For example, in these statements,

```
int legalAge = 18;
int voterAge = legalAge;
```

the literal *18* is an expression. Its value is *18*, which is assigned to the variable *legalAge*. Then, in the second statement, *legalAge* is an expression, whose value is *18*. Thus the value *18* is assigned to *voterAge*. So after these statements have been executed, both *legalAge* and *voterAge* will have the value *18*.

One restriction, however, is that an assignment expression cannot include another variable unless that variable has been defined previously. The statement defining the *length* variable that follows is **invalid**, because it refers to *width*, which is not defined until the next line.

```
int length = width * 2; // invalid, width is not yet defined
int width;
```

The compiler flags the statement defining *length* as an error

```
cannot find symbol
```

because *width* has not yet been defined.

An expression can be quite complex, consisting of multiple variables, constants, literals, and operators. Before we can look at examples of more complex expressions, however, we need to discuss the *arithmetic operators*.

2.3.2 Arithmetic Operators

Java's arithmetic operators are used for performing calculations on numeric data. Some of these operators are shown in Table 2.9.

All these operators take two operands, which are espressions; thus, they are called **binary operators**.

TABLE 2.9 Arithmetic Operators

Operator	Operation
+	addition
–	subtraction
*	multiplication
/	division
%	modulus (remainder after division)

In Example 2.6, we make a variety of calculations to demonstrate the addition, subtraction, multiplication, and division arithmetic operators. We will discuss integer division and the modulus operator later in the chapter. The output from this program is shown in Figure 2.5.

```
1 /* Arithmetic Operators
2    Anderson, Franceschi
3 */
4
5 public class ArithmeticOperators
6 {
7   public static void main( String [] args )
8   {
9     // calculate the cost of lunch
10    double salad = 5.95;
11    double water = .89;
12    System.out.println( "The cost of lunch is $"
13                          + ( salad + water ) );
14
15    // calculate your age as of a certain year
16    int targetYear = 2011;
17    int birthYear = 1993;
18    System.out.println( "Your age in " + targetYear + " is "
19                          + ( targetYear - birthYear ) );
20
21    // calculate the total calories of apples
22    int caloriesPerApple = 127;
23    int numberOfApples = 3;
24    System.out.println( "The calories in " +  numberOfApples
25                         + " apples is " +
26                         + ( caloriesPerApple * numberOfApples ) );
27
28    // calculate miles per gallon
29    double miles = 426.8;
30    double gallons = 15.2;
31    double mileage = miles / gallons;
32    System.out.println( "The mileage is "
33                        + mileage + " miles per gallon." );
34  }
35 }
```

Example 2.6 Using Arithmetic Operators

```
The cost of lunch is $6.84
Your age in 2011 is 18
The calories in 3 apples is 381
The mileage is 28.078947368421055 miles per gallon.
```

Figure 2.5
Output from Example 2.6

Example 2.6 demonstrates a number of small operations. To calculate a total price (lines 12 and 13), we add the individual prices. To calculate an age (lines 18 and 19), we subtract the birth year from the target year. To calculate the number of calories in multiple apples (lines 24–26), we multiply the number of calories in one apple by the number of apples. We calculate miles per gallon by dividing the number of miles driven by the number of gallons of gas used (line 31). Note that we can either store the result in another variable, as we did in line 31, and subsequently output the result (lines 32–33), or we can output the result of the calculation directly by writing the expression in the *System.out.println* statement, as we did in the other calculations in this example.

SOFTWARE ENGINEERING TIP

For readable code, insert a space between operators and operands.

SOFTWARE ENGINEERING TIP

Developing and testing your code in steps makes it easier to find and fix errors.

2.3.3 Operator Precedence

The statements in Example 2.6 perform simple calculations, but what if you want to make more complex calculations using several operations, such as calculating how much money you have in coins? Let's say you have two quarters, three dimes, and two nickels. To calculate the value of these coins in pennies, you might use this expression:

```
int pennies = 2 * 25 + 3 * 10 + 2 * 5;
```

In which order should the computer do the calculation? If the value of the expression were calculated left to right, then the result would be

```
= 2 * 25 + 3 * 10 + 2 * 5
=   50   + 3 * 10 + 2 * 5
=        53   * 10 + 2 * 5
=             530  + 2 * 5
=                  532 * 5
=                    2660
```

Clearly, 2,660 pennies is not the right answer. To calculate the correct number of pennies, the multiplications should be performed first, then the additions. This, in fact, is the order in which Java will calculate the preceding expression.

The Java compiler follows a set of rules called **operator precedence** to determine the order in which the operations should be performed.

Table 2.10 provides the order of precedence of the operators we've discussed so far. The operators in the first row—parentheses—are evaluated first, then the operators in the second row (*, /, %) are evaluated, and so on with the operators in each row. When two or more operators on the same level appear in the same expression, the order of evaluation is left to right, except for the assignment operator, which is evaluated right to left.

As we introduce more operators, we'll add them to the Order of Precedence chart. The complete chart is provided in Appendix B.

Using Table 2.10 as a guide, let's recalculate the number of pennies:

```
int pennies = 2 * 25 + 3 * 10 + 2 * 5;
            =  50    +   30   +  10
            =  90
```

As you can see, *90* is the correct number of pennies in two quarters, three dimes, and two nickels.

We also could have used parentheses to clearly display the order of calculation. For example,

```
int pennies = (2 * 25) + (3 * 10) + (2 * 5);
            =  50      +    30    +   10
            =  90
```

The result is the same, 90 pennies.

TABLE 2.10 Operator Precedence

Operator Hierarchy	Order of Same-Statement Evaluation	Operation
()	left to right	parentheses for explicit grouping
*, /, %	left to right	multiplication, division, modulus
+, −	left to right	addition, subtraction
=	right to left	assignment

It sometimes helps to use parentheses to clarify the order of calculations, but parentheses are essential when your desired order of evaluation is different from the rules of operator precedence. For example, to calculate the value of this formula:

$$\frac{x}{2y}$$

you could write this code:

```
double result = x / 2 * y;
```

This would generate incorrect results because, according to the rules of precedence, *x/2* would be calculated first, then the result of that division would be multiplied by *y*. In algebraic terms, the preceding statement is equivalent to

$$\frac{x}{2} * y$$

To code the original formula correctly, you need to use parentheses to force the multiplication to occur before the division:

```
double result = x / ( 2 * y );
```

2.3.4 Programming Activity 1: Converting Inches to Centimeters

Now that we know how to define variables and constants and make calculations, let's put this all together by writing a program that converts inches into the equivalent centimeters.

Locate the *MetricLength.java* source file found in the Chapter 2, Programming Activity 1 folder on the CD-ROM accompanying this book. Copy the file to your computer.

Open the *MetricLength.java* source file. You'll notice that the class already contains some source code. Your job is to fill in the blanks.

When we write a program, we begin by considering these questions:

1. What data values does the program require?
   a. What data values do we know?
   b. What data values will change from one execution of the program to the next?
2. What processing (algorithm) do we need to implement?
3. What is the output?

The comments in the source file will guide you through the answers to these questions, and by doing so, you will complete the program. Search for five asterisks in a row (*****). This will position you to the places in the source code where you will add your code. The *MetricLength.java* source code is shown in Example 2.7. Sample output for a value of 5.2 inches is shown in Figure 2.6.

```
1 /* MetricLength - converts inches to centimeters
2    Anderson, Franceschi
3 */
4
5 public class MetricLength
6 {
7    public static void main( String [] args )
8    {
9
10       /***** 1. What data values do we know?
11       /* We know that there are 2.54 centimeters in an inch.
12       /* Declare a double constant named CM_PER_INCH.
13       /* Assign CM_PER_INCH the value 2.54.
14       */
15
16
17       /***** 2.   What other data does the program require?
18       /* For this program, we require the number of inches.
19       /* Declare a double variable named inches.
20       /* Assign any desired value to this variable.
21       */
22
23
24       /***** 3. Calculation: convert inches to centimeters
25       /* Declare a double variable named centimeters.
26       /* Multiply inches by CM_PER_INCH
27       /* and store the result in centimeters.
28       */
29
30
31       /***** 4. Output
32       /* Write one or two statements that output
33       /* the original inches and the equivalent centimeters.
34       /* Try to match the sample output in Figure 2.6.
35       */
36
```

```
5.2 inches are equivalent to 13.208 centimeters.
```

Figure 2.6
Sample Output for Programming Activity 1

```
37
38
39    }
40 }
```

Example 2.7 Converting Feet and Inches to Centimeters

? DISCUSSION QUESTIONS

1. How do you know that your program results are correct?
2. If you change the inches data value, does your program still produce correct results?

2.3.5 Integer Division and Modulus

Division with two integer operands is performed in the Arithmetic Logic Unit (ALU), which can calculate only an integer result. Any fractional part is truncated; no rounding is performed. The remainder after division is available, however, as an integer, by taking the modulus (%) of the two integer operands. Thus, in Java, the integer division (/) operator will calculate the quotient of the division, whereas the modulus (%) operator will calculate the remainder of the division.

```
 1 /* DivisionAndModulus Class
 2    Anderson, Franceschi
 3 */
 4
 5 public class DivisionAndModulus
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     final int PENNIES_PER_QUARTER = 25;
10     int pennies = 113;
11
12     int quarters = pennies / PENNIES_PER_QUARTER;
13     System.out.println( "There are " + quarters + " quarters in "
14             + pennies + " pennies" );
15
```

```
16      int penniesLeftOver = pennies % PENNIES_PER_QUARTER;
17      System.out.println( "There are " + penniesLeftOver
18              + " pennies left over" );
19
20      final double MONTHS_PER_YEAR = 12;
21      double annualSalary = 50000.0;
22
23      double monthlySalary = annualSalary / MONTHS_PER_YEAR;
24      System.out.println( "The monthly salary is " + monthlySalary );
25    }
26 }
```

EXAMPLE 2.8 How Integer Division and Modulus Work

In Example 2.8, we have 113 pennies and we want to convert those pennies into quarters. We can find the number of quarters by dividing 113 by 25. The *int* variable *pennies* is assigned the value 113 at line 10. At line 12, the variable *quarters* is assigned the result of the integer division of *pennies* by the constant *PENNIES_PER_QUARTER*. Since the quotient of the division of 113 by 25 is 4, *quarters* will be assigned 4. At line 16, we use the modulus operator to assign to the variable *penniesLeftOver* the remainder of the division of *pennies* by *PENNIES_PER_QUARTER*. Since the remainder of the division of 113 by 25 is 13, 13 will be assigned to *penniesLeftOver*. Notice that integer division and modulus are independent calculations. You can perform a division without also calculating the modulus, and you can calculate the modulus without performing the division.

At line 23, we divide a *double* by a *double;* therefore, a floating-point division will be performed by the floating-point unit (FPU), and the result will be assigned to the variable *monthlySalary*. Figure 2.7 shows the output of the program.

Figure 2.7
Output of Example 2.8

```
There are 4 quarters in 113 pennies
There are 13 pennies left over
The monthly salary is 4166.666666666667
```

The modulus is actually a useful operator. As you will see later in this book, it can be used to determine whether a number is even or odd, to control the number of data items that are written per line, to determine if one number is a factor of another, and for many other uses.

CODE IN ACTION

To see arithmetic operators used in a program, look for the Chapter 2 Flash movie on the CD-ROM accompanying this book. Click on the link for Chapter 2 to start the movie.

Skill Practice
with these end-of-chapter questions

2.6.2 Reading and Understanding Code

Questions 7, 8, 9, 10, 11, 12, 13

2.6.3 Fill In the Code

Questions 27, 29, 32

2.6.4 Identifying Errors in Code

Question 35

2.6.6 Write a Short Program

Question 44

2.3.6 Division by Zero

As you might expect, Java does not allow integer division by 0. If you include this statement in your program:

```
int result = 4 / 0;
```

the code will compile without errors, but at run time, when this statement is executed, the JVM will generate an exception and print an error message on the Java console:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

In most cases, this stops the program. In Chapter 5, we show you how to avoid dividing by zero by first testing whether the divisor is zero before performing the division.

In contrast, floating-point division by zero does not generate an exception. If the dividend is non-zero, the answer is *Infinity*. If both the dividend and divisor are zero, the answer is *NaN*, which stands for "Not a Number."

Example 2.9 illustrates the three cases of dividing by zero. As we can see on the output shown in Figure 2.8, line 16 of Example 2.9 never executes. The exception is generated at line 15 and the program halts execution.

```
1 /* DivisionByZero Class
2    Anderson, Franceschi
3 */
4
5 public class DivisionByZero
6 {
7   public static void main( String [ ] args )
8   {
9     double result1 = 4.3 / 0.0;
10    System.out.println( "The value of result1 is " + result1 );
11
12    double result2 = 0.0 / 0.0;
13    System.out.println( "The value of result2 is " + result2 );
14
15    int result3 = 4 / 0;
16    System.out.println( "The value of result3 is " + result3 );
17  }
18 }
```

EXAMPLE 2.9 Results of Division by Zero

Although floating-point division by zero doesn't bring your program to a halt, it doesn't provide useful results either. It's a good practice to avoid dividing by zero in the first place. We'll give you tools to do that in Chapter 5.

Figure 2.8
Output of Example 2.9

```
The value of result1 is Infinity
The value of result2 is NaN
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivisionByZero.main(DivisionByZero.java:15)
```

2.3.7 Mixed-Type Arithmetic and Type Casting

So far, we've used a single data type in the expressions we've evaluated. But life isn't always like that. Calculations often involve data of different primitive types.

When calculations of mixed types are performed, lower-precision operands are converted, or **promoted**, to the type of the operand that has the higher precision.

The promotions are performed using the *first* of these rules that fits the situation:

1. If either operand is a *double*, the other operand is converted to a *double*.
2. If either operand is a *float*, the other operand is converted to a *float*.
3. If either operand is a *long*, the other operand is converted to a *long*.
4. If either operand is an *int*, the other operand is promoted to an *int*.
5. If neither operand is a *double*, *float*, *long*, or an *int*, both operands are promoted to *int*.

Table 2.11 summarizes these rules of promotion.

This arithmetic promotion of operands is called **implicit type casting** because the compiler performs the promotions automatically, without our specifying that the conversions should be made. Note that the data type of

TABLE 2.11 Rules of Operand Promotion

Data Type of One Operand	Data Type of Other Operand	Promotion of Other Operand	Data Type of Result
double	*char, byte, short, int, long, float*	*double*	*double*
float	*char, byte, short, int, long*	*float*	*float*
long	*char, byte, short, int*	*long*	*long*
int	*char, byte, short*	*int*	*int*
short	*char, byte*	Both operands are promoted to *int*	*int*
byte	*char*	Both operands are promoted to *int*	*int*

any promoted variable is not permanently changed; its type remains the same after the calculation has been performed.

Table 2.11 shows many rules, but essentially, any arithmetic expression involving integers and floating-point numbers will evaluate to a floating-point number.

Lines 9 to 12 of Example 2.10 illustrate the rules of promotion. At line 11, the expression *PI * radius * radius* is a mixed-type expression. This expression will be evaluated left to right, evaluating the mixed-type expression *PI * radius* first. *PI* is a *double* and *radius* is an *int*. Therefore, *radius* is promoted to a *double* (4.0) and the result of *PI * radius* is a *double* (12.56636). Then, the next calculation (*12.56636 * radius*) also involves a mixed-type expression, so *radius* is again promoted to a *double* (4.0). The final result, 50.26544, is a *double* and is assigned to *area*. Figure 2.9 shows the output of the complete program.

Sometimes, it's useful to instruct the compiler specifically to convert the type of a variable. In this case, you use **explicit type casting**, which uses this syntax:

```
(dataType) ( expression )
```

The expression will be converted, or type cast, to the data type specified. The parentheses around *expression* are needed only when the expression consists of a calculation that you want to be performed before the type casting.

Type casting is useful in calculating an average. Example 2.10 shows how to calculate your average test grade. Your test scores are 94, 86, 88, and 97, making the combined total score 365. We expect the average to be 91.25.

```
 1 /*  MixedDataTypes Class
 2     Anderson, Franceschi
 3 */
 4
 5 public class MixedDataTypes
 6 {
 7    public static void main( String [ ] args )
 8    {
 9       final double PI = 3.14159;
10       int radius = 4;
```

```
11        double area = PI * radius * radius;
12        System.out.println( "The area is " + area );
13
14        int total = 365, count = 4;
15        double average = total / count;
16        System.out.println( "\nPerforming integer division, "
17                            + "then implicit typecasting" );
18        System.out.println( "The average test score is " + average );
19        // 91.0 INCORRECT ANSWER!
20
21        average = ( double ) ( total / count );
22        System.out.println( "\nPerforming integer division, "
23                           +  "then explicit typecasting" );
24        System.out.println( "The average test score is " + average );
25        // 91.0 INCORRECT ANSWER!
26
27        average = ( double ) total / count;
28        System.out.println( "\nTypecast one variable to double, "
29                           +  "then perform division" );
30        System.out.println( "The average test score is " + average );
31        // 91.25 CORRECT ANSWER
32     }
33  }
```

EXAMPLE 2.10 Mixed Data Type Arithmetic

Line 15 first attempts to calculate the average but results in a wrong answer because both *total* and *count* are integers. So integer division is performed, which truncates any remainder. Thus, the result of *total / count* is 91. Then 91 is assigned to *average*, which is a *double*, so 91 becomes 91.0.

Line 21 is a second attempt to calculate the average; again, this code does not work correctly because the parentheses force the division to be performed before the type casting. Thus, because *total* and *count* are both integers, integer division is performed again. The quotient, 91, is then cast to a *double*, 91.0, and that *double* value is assigned to *average*.

At line 27, we correct this problem by casting only one of the operands to a *double*. This forces the other operand to be promoted to a *double*. Then floating-point division is performed, which retains the remainder. It doesn't matter whether we cast *total* or *count* to a *double*. Casting either to a *double* forces the division to be a floating-point division.

Figure 2.9
Output of Example 2.10

```
The area is 50.26544

Performing integer division, then implicit typecasting
The average test score is 91.0

Performing integer division, then explicit typecasting
The average test score is 91.0

Typecast one variable to double, then perform division
The average test score is 91.25
```

CODE IN ACTION

To see the calculation of an average using mixed data types, look for the Chapter 2 Flash movie on the CD-ROM accompanying this book. Click on the link for Chapter 2 to view the movie.

2.3.8 Shortcut Operators

A common operation in programming is adding 1 to a number (**incrementing**) or subtracting 1 from a number (**decrementing**). For example, if you were counting how many data items the user entered, every time you read another data item, you would add 1 to a count variable.

Because incrementing or decrementing a value is so common in programming, Java provides shortcut operators to do this: `++` and `--`. (Note that there are no spaces between the two plus and minus signs.) The statement

```
count++;
```

adds 1 to the value of *count*, and the statement

```
count--;
```

subtracts 1 from the value of *count*. Thus,

```
count++;
```

is equivalent to

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

and

```
count--;
```

is equivalent to

```
count = count - 1;
```

Both of these operators have **prefix** and **postfix** versions. The prefix versions precede the variable name (`++a` or `--a`) whereas the postfix versions follow the variable name (`a++` or `a--`). Both increment or decrement the variable. If they are used as a single, atomic statement (as in the preceding statements), there is no difference between the two versions. So

```
a++;
```

is functionally equivalent to

```
++a;
```

and

```
a--;
```

is functionally equivalent to

```
--a;
```

However, if they are used inside a more complex expression, then they differ as follows. The prefix versions increment or decrement the variable first, then the new value of the variable is used in evaluating the expression. The postfix versions increment or decrement the variable after the old value of the variable is used in the expression.

Example 2.11 illustrates this difference.

```
1 /* ShortcutOperators Class
2    Anderson, Franceschi
3 */
4
5 public class ShortcutOperators
6 {
7   public static void main( String [ ] args )
8   {
9     int a = 6;
10    int b = 2;
11
12    System.out.println( "At the beginning, a is " + a );
13    System.out.println( "Increment a with prefix notation: " + ++a );
14    System.out.println( "In the end, a is " + a );
```

```
15
16      System.out.println( "\nAt the beginning, b is " + b );
17      System.out.println( "Increment b with postfix notation: " + b++ );
18      System.out.println( "In the end, b is " + b );
19    }
20 }
```

EXAMPLE 2.11 Prefix and Postfix Increment Operators

Lines 9 and 10 declare and initialize two *int* variables, *a* and *b*, to 6 and 2, respectively. In order to illustrate the effect of both the prefix and postfix increment operators, we output their original values at lines 12 and 16. At line 13, we use the prefix increment operator to increment *a* inside an output statement; *a* is incremented before the output statement is executed, resulting in the output statement using the value 7 for *a*. At line 17, we use the postfix increment operator to increment *b* inside an output statement; *b* is incremented after the output statement is executed, resulting in the output statement using the value 2 for *b*. Lines 14 and 18 simply output the values of *a* and *b* after the prefix and postfix operators were used at lines 13 and 17. Figure 2.10 shows the output of this example.

Another set of shortcut operators simplify common calculations that change a single value. For example, the statement

```
a = a + 2: // add 2 to a
```

can be simplified as

```
a += 2; // add 2 to a
```

The value added to the target variable can be a variable name or a larger expression.

Figure 2.10
Output of Example 2.11

```
At the beginning, a is 6
Increment a with prefix notation: 7
In the end, a is 7

At the beginning, b is 2
Increment b with postfix notation: 2
In the end, b is 3
```

The shortcut addition operator (+=) is a single operator; there are no spaces between the + and the =. Also, be careful not to reverse the order of the operators. For example, in the following statement, the operators are reversed, so the compiler interprets the statement as "assign a positive 2 to a."

```
a =+ 2 ; // Incorrect! Assigns a positive 2 to a
```

COMMON ERROR TRAP

No spaces are allowed between the arithmetic operator (+) and the equal sign. Note also that the sequence is +=, not =+.

Java provides shortcut operators for each of the basic arithmetic operations: addition, subtraction, multiplication, division, and modulus. These operators are especially useful in performing repetitive calculations and in converting values from one scale to another. For example, to convert feet to inches, we multiply the number of feet by 12. So we can use the *= shortcut operator:

```
int length = 3; // length in feet
length *= 12;   // length converted to inches
```

Converting from one scale to another is a common operation in programming. For example, earlier in the chapter we converted quarters, dimes, and nickels to pennies. You might also need to convert hours to seconds, feet to square feet, or Fahrenheit temperatures to Celsius.

Example 2.12 demonstrates each of the shortcut arithmetic operators. The output is shown in Figure 2.11.

```
1 /*  Shortcut Arithmetic Operators
2     Anderson, Franceschi
3 */
4
5 public class ShortcutArithmeticOperators
6 {
7    public static void main( String [ ] args )
8    {
9       int a = 5;
10      System.out.println( "a is " + a );
11
12      a += 10;     //  a = a + 10;
13      System.out.println( "\nAfter a += 10;  a is " + a );
14
15      a -= 3;      //  a = a - 3;
16      System.out.println( "\nAfter a -= 3; a is " + a );
17
18      a *= 2;      //  a = a * 2;
19      System.out.println( "\nAfter a *= 2; a is " + a );
20
21      a /= 6;      //  a = a / 6;
```

```
22        System.out.println( "\nAfter a /= 6; a is " + a );
23
24        a %= 3;       //  a = a % 3;
25        System.out.println( "\nAfter a %= 3; a is " + a );
26    }
27 }
```

Example 2.12 Shortcut Arithmetic Operators

Figure 2.11
Output of Example 2.12

```
a is 5

After a += 10; a is 15

After a -= 3; a is 12

After a *= 2; a is 24

After a /= 6; a is 4

After a %= 3; a is 1
```

Table 2.12 summarizes the shortcut operators and Table 2.13 shows where the shortcut operators fit into the order of operator precedence.

TABLE 2.12 Shortcut Operators

Shortcut Operator	Example	Equivalent Statement
++	a++; or ++a;	a = a + 1;
−−	a−−; or −−a;	a = a − 1;
+=	a += 3;	a = a + 3;
−=	a −=10;	a = a − 10;
*=	a *= 4;	a = a • 4;
/=	a /= 7;	a = a / 7;
%=	a %= 10;	a = a % 10;

TABLE 2.13 Order of Operator Precedence

Operator Hierarchy	Order of Same-Statement Evaluation	Operation
( )	left to right	parentheses for explicit grouping
++, −−	**right to left**	**shortcut postincrement**
++, −−	**right to left**	**shortcut preincrement**
*, /, %	left to right	multiplication, division, modulus
+, −	left to right	addition or *String* concatenation, subtraction
=, **+=, −=, *=, /=, %=**	right to left	assignment operator and **shortcut assignment operators**

Skill Practice
with these end-of-chapter questions

2.6.1 Multiple Choice Exercises
Question 3

2.6.2 Reading and Understanding Code
Questions 14, 15, 16, 17, 18, 19, 20, 21, 22

2.6.3 Fill In the Code
Questions 28, 30, 31

2.6.4 Identifying Errors in Code
Questions 36, 37

2.6.5 Debugging Area
Questions 42, 43

2.6.6 Write a Short Program
Question 45

2.6.8 Technical Writing
Question 51

2.4 Programming Activity 2: Temperature Conversion

For this Programming Activity, you will write a program to convert a temperature in Fahrenheit to Celsius. The conversion formula is the following:

$$T_c = 5 / 9 \ (T_f - 32)$$

where T_c is the temperature in Celsius and T_f is the temperature in Fahrenheit, and 32 is the freezing point of water.

Locate the *TemperatureConversion.java* source file found in the Chapter 2, Programming Activity 2 folder on the CD-ROM accompanying this book. Copy the file to your computer. The source code is shown in Example 2.13.

```
1 /* Temperature Conversion
2    Anderson, Franceschi
3 */
4
5 public class TemperatureConversion
6 {
7    public static void main( String [] args )
8    {
9       //***** 1. declare any constants here
10
11
12       //***** 2.  declare the temperature in Fahrenheit as an int
13
14
15       //***** 3. calculate equivalent Celsius temperature
16
17
18       //***** 4. output the temperature in Celsius
19
20
21       //***** 5. convert Celsius temperature back to Fahrenheit
22
23
24       //***** 6. output Fahrenheit temperature to check correctness
25
26
27    }
28 }
```

Example 2.13 ***TemperatureConversion.java***

Open the *TemperatureConversion.java* source file. You'll notice that the class already contains some source code. Your job is to fill in the blanks.

To verify that your code produces the correct output, add code to convert your calculated Celsius temperature back to Fahrenheit and compare that value to the original Fahrenheit temperature. The formula for converting Celsius to Fahrenheit is:

$$T_f = 9 / 5 * T_c + 32$$

Before writing this program, you need to design a plan of attack. Ask yourself:

- What data do I need to define?
- What calculations should I make?
- What is the output of the program?
- How do I select data values so they will provide good test data for my code?

Choose any input value for the Fahrenheit temperature. After you write the program, try changing the original temperature value, recompiling and rerunning the program to verify that the temperature conversion works for multiple input values.

? DISCUSSION QUESTIONS

1. How did you change the expression 5 / 9 so that the value was not 0?
2. What constant(s) did you define?
3. What data type did you use for the Celsius temperature? Why?

2.5 Chapter Summary

SUMMARY

- Java programs consist of at least one class.
- Identifiers are symbolic names for classes, methods, and data. Identifiers should start with a letter and may contain any combination of letters and digits, but no spaces. The length of an identifier is essentially unlimited. Identifier names are case-sensitive.
- Java's reserved words cannot be used as identifiers.
- The basic building block of a Java program is the statement. A statement is terminated with a semicolon and can span several lines.

- Any amount of white space is permitted between identifiers, Java keywords, operands, operators, and literals. White space characters are the space, tab, newline, and carriage return.
- A block, which consists of 0, 1, or more statements, starts with a left curly brace and ends with a right curly brace. Blocks can be used anywhere in the program that a statement is legal.
- Comments are ignored by the compiler. Block comments are delineated by /* and */. Line comments start with // and continue to the end of the line.
- Java supports eight primitive data types: *double*, *float*, *long*, *int*, *short*, *byte*, *char*, and *boolean*.
- Variables must be declared before they are used. Declaring a variable is specifying the data item's identifier and data type. The syntax for declaring a variable is: `dataType identifier1, identifier2, ...;`
- Begin variable names with a lowercase letter. If the variable name consists of more than one word, begin each word after the first with a capital letter. Do not put spaces between words.
- An integer data type is one that evaluates to a positive or negative whole number. Java recognizes four integer data types: *int*, *short*, *long*, and *byte*.
- Floating-point data types store numbers with fractional parts. Java supports two floating-point data types: the single-precision type *float*, and the double-precision type *double*.
- The *char* data type stores one Unicode character. Because Unicode characters are encoded as unsigned numbers using 16 bits, a *char* variable is stored in two bytes of memory.
- The *boolean* data type can store only two values, which are expressed using the Java reserved words *true* and *false*.
- The assignment operator (=) is used to give a value to a variable.
- To assign an initial value to a variable, use this syntax when declaring the variable:

```
dataType variable1 = initialValue1;
```

- Literals can be used to assign initial values or to reassign the value of a variable.

- Constants are data items whose value, once assigned, cannot be changed. Data items that you know should not change throughout the execution of a program should be declared as a constant, using this syntax:

```
final dataType CONSTANT_IDENTIFIER = initialValue;
```

- Constant identifiers, by convention, are composed of all capital letters with underscores separating words.
- An expression consists of operators and operands that evaluate to a single value.
- The value of an expression can be assigned to a variable or constant, which must be a data type compatible with the value of the expression and cannot be a constant that has been assigned a value already.
- Java provides binary operators for addition, subtraction, multiplication, division, and modulus.
- Calculation of the value of expressions follows the rules of operator precedence.
- Integer division truncates any fractional part of the quotient.
- When an arithmetic operator is invoked with operands that are of different primitive types, the compiler temporarily converts, or promotes, one or both of the operands.
- An expression or a variable can be temporarily cast to a different data type using this syntax:

```
(dataType) ( expression )
```

- Shortcut operators ++ and –– simplify incrementing or decrementing a value by 1. The prefix versions precede the variable name and increment or decrement the variable, then use its new value in evaluation of the expression. The postfix versions follow the variable name and increment or decrement the variable after using the old value in the expression.
- Java provides shortcut operators for each of the basic arithmetic operations: addition, subtraction, multiplication, division, and modulus.

EXERCISES, PROBLEMS, AND PROJECTS

2.6 Exercises, Problems, and Projects

2.6.1 Multiple Choice Exercises

1. What is the valid way to declare an integer variable named *a*? (Check all that apply.)
   - ❑ `int a;`
   - ❑ `a int;`
   - ❑ `integer a;`
2. Which of the following identifiers are valid?
   - ❑ `a`
   - ❑ `sales`
   - ❑ `sales&profit`
   - ❑ `int`
   - ❑ `inter`
   - ❑ `doubleSales`
   - ❑ `TAX_RATE`
   - ❑ `1stLetterChar`
   - ❑ `char`
3. Given three declared and initialized *int* variables *a*, *b*, and *c*, which of the following statements are valid?
   - ❑ `a = b;`
   - ❑ `a = 67;`
   - ❑ `b = 8.7;`
   - ❑ `a + b = 8;`
   - ❑ `a * b = 12;`
   - ❑ `c = a – b;`
   - ❑ `c = a / 2.3;`
   - ❑ `boolean t = a;`

- `a /= 4;`
- `a += c;`

2.6.2 Reading and Understanding Code

4. What is the output of this code sequence?

```
double a = 12.5;
System.out.println( a );
```

5. What is the output of this code sequence?

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

6. What is the output of this code sequence?

```
float a = 13f;
System.out.println( a );
```

7. What is the output of this code sequence?

```
double a = 13 / 5;
System.out.println( a );
```

8. What is the output of this code sequence?

```
int a = 13 / 5;
System.out.println( a );
```

9. What is the output of this code sequence?

```
int a = 13 % 5;
System.out.println( a );
```

10. What is the output of this code sequence?

```
int a = 12 / 6 * 2;
System.out.println( a );
```

11. What is the output of this code sequence?

```
int a = 12 / ( 6 * 2 );
System.out.println( a );
```

12. What is the output of this code sequence?

```
int a = 4 + 6 / 2;
System.out.println( a );
```

13. What is the output of this code sequence?

```
int a = ( 4 + 6 ) / 2;
System.out.println( a );
```

14. What is the output of this code sequence?

```
double a = 12.0 / 5;
System.out.println( a );
```

15. What is the output of this code sequence?

```
int a = (int) 12.0 / 5;
System.out.println( a );
```

16. What is the output of this code sequence?

```
double a = (double) ( 12 ) / 5;
System.out.println( a );
```

17. What is the output of this code sequence?

```
double a = (double) ( 12 / 5 );
System.out.println( a );
```

18. What is the output of this code sequence?

```
int a = 5;
a++;
System.out.println( a );
```

19. What is the output of this code sequence?

```
int a = 5;
System.out.println( a-- );
```

20. What is the output of this code sequence?

```
int a = 5;
System.out.println( --a );
```

21. What is the output of this code sequence?

```
int a = 5;
a += 2;
System.out.println( a );
```

22. What is the output of this code sequence?

```
int a = 5;
a /= 6;
System.out.println( a );
```

2.6.3 Fill In the Code

23. Write the code to declare a *float* variable named *a* and assign *a* the value 34.2.

```
// your code goes here
```

24. Write the code to assign the value 10 to an *int* variable named *a*.

```
int a;
// your code goes here
```

25. Write the code to declare a *boolean* variable named *a* and assign *a* the value *false*.

```
// your code goes here
```

26. Write the code to declare a *char* variable named *a* and assign *a* the character B.

```
// your code goes here
```

27. Write the code to calculate the total of three *int* variables *a*, *b*, and *c* and print the result.

```
int a = 3;
int b = 5;
int c = 8;

// your code goes here
```

28. Write the code to calculate the average of two *int* variables *a* and *b* and print the result. The average should be printed as a floating-point number.

```
int a = 3;
int b = 5;

// your code goes here
```

29. Write the code to calculate and print the remainder of the division of two *int* variables with the values 10 and 3 (the value printed will be 1).

```
int a = 10;
int b = 3;

// your code goes here
```

30. This code increases the value of a variable *a* by 1, using the shortcut increment operator.

```
int a = 7;

// your code goes here
```

31. This code multiplies the value of a variable *a* by 3, using a shortcut operator.

```
int a = 7;

// your code goes here
```

32. Assume that we have already declared and initialized two *int* variables, *a* and *b*. Convert the following sentences to legal Java expressions and statements.

- ❑ b gets a plus 3 minus 7
- ❑ b gets a times 4
- ❑ a gets b times b
- ❑ a gets b times 3 times 5
- ❑ b gets the quotient of the division of a by 2
- ❑ b gets the remainder of the division of a by 3

2.6.4 Identifying Errors in Code

33. Where is the error in this code sequence?

```
int a = 3.3;
```

34. Where is the error in this code sequence?

```
double a = 45.2;
float b = a;
```

35. Where is the error in this code sequence?

```
int a = 7.5 % 3;
```

36. What would happen when this code sequence is compiled and executed?

```
int a = 5 / 0;
```

37. Where is the error in this code sequence?

```
int a = 5;
a - = 4;
```

38. Is there an error in this code sequence? Explain.

```
char c = 67;
```

39. Is there an error in this code sequence? Explain.

```
boolean a = 1;
```

2.6.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

40. You coded the following on line 8 of class *Test.java*:

```
int a = 26.4;
```

When you compile, you get the following message:

```
Test.java:8: possible loss of precision
  int a = 26.4;
          ^
  required: int
  found : double
1 error
```

Explain what the problem is and how to fix it.

41. You coded the following on line 8 of class *Test.java*:

```
int a = 3
```

When you compile, you get the following message:

```
Test.java:8: ';' expected
        int a = 3
                ^
```

Explain what the problem is and how to fix it.

42. You coded the following in class *Test.java*:

```
int a = 32;
int b = 10;
double c = a / b;
System.out.println( "The value of c is " + c );
```

The code compiles properly and runs, but the result is not what you expected. The output is

```
The value of c is 3.0
```

You expected the value of *c* to be 3.2. Explain what the problem is and how to fix it.

43. You coded the following in class *Test.java*:

```
int a = 5;
a =+ 3;
System.out.println( "The value of a is " + a );
```

The code compiles properly and runs, but the result is not what you expected. The output is

```
The value of a is 3
```

You expected the value of *a* to be 8. Explain what the problem is and how to fix it.

2.6.6 Write a Short Program

44. Write a program that calculates and outputs the square of each integer from 1 to 9.

45. Write a program that calculates and outputs the average of integers 1, 7, 9, and 34.

46. Write a program that outputs the following:

```
****
```

2.6.7 Programming Projects

47. Write a program that prints the letter X composed of asterisks (*). Your output should look like this:

```
*   *
 * *
  *
 * *
*   *
```

48. Write a program that converts 10, 50, and 100 kilograms to pounds (1 lb = 0.454 kg).

49. Write a program that converts 2, 5, and 10 inches to millimeters (1 inch = 25.4 mm).

50. Write a program to compute and output the perimeter and the area of a circle having a radius of 3.2 inches.

2.6.8 Technical Writing

51. Some programmers like to write code that is as compact as possible, for instance, using the increment (or decrement) operator in the middle of another statement. Typically, these programmers document their programs with very few comments. Discuss whether this is a good idea, keeping in mind that a program "lives" through a certain period of time.

52. Compare the following data types for integer numbers: *int*, *short*, and *long*. Discuss their representation in binary, how much space they take in memory, and the purpose of having these data types available to programmers.