Module 2

Introducing Data Types and Operators

CRITICAL SKILLS

2.1	Know Java's primitive types
2.2	Use literals
2.3	Initialize variables
2.4	Know the scope rules of variables within a method
2.5	Use the arithmetic operators
2.6	Use the relational and logical operators
2.7	Understand the assignment operators
2.8	Use shorthand assignments
2.9	Understand type conversion in assignments
2.10	Cast incompatible types
2.11	Understand type conversion in expressions

At the foundation of any programming language are its data types and operators, and Java is no exception. These elements define the limits of a language and determine the kind of tasks to which it can be applied. Fortunately, Java supports a rich assortment of both data types and operators, making it suitable for any type of programming.

Data types and operators are a large subject. We will begin here with an examination of Java's foundational data types and its most commonly used operators. We will also take a closer look at variables and examine the expression.

Why Data Types Are Important

Data types are especially important in Java because it is a strongly typed language. This means that all operations are type checked by the compiler for type compatibility. Illegal operations will not be compiled. Thus, strong type checking helps prevent errors and enhances reliability. To enable strong type checking, all variables, expressions, and values have a type. There is no concept of a "type-less" variable, for example. Furthermore, the type of a value determines what operations are allowed on it. An operation allowed on one type might not be allowed on another.

CRITICAL SKILL 2.1

Java's Primitive Types

Java contains two general categories of built-in data types: object-oriented and non-object-oriented. Java's object-oriented types are defined by classes, and a discussion of classes is deferred until later. However, at the core of Java are eight primitive (also called elemental or simple) types of data, which are shown in Table 2-1. The term *primitive* is used here to indicate that these types are not objects in an object-oriented sense, but rather, normal binary values. These primitive types are not objects because of efficiency concerns. All of Java's other data types are constructed from these primitive types.

Java strictly specifies a range and behavior for each primitive type, which all implementations of the Java Virtual Machine must support. Because of Java's portability requirement, Java is uncompromising on this account. For example, an **int** is the same in all execution environments. This allows programs to be fully portable. There is no need to rewrite code to fit a specific platform. Although strictly specifying the size of the primitive types may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

Type	Meaning
boolean	Represents true/false values
byte	8-bit integer
char	Character
double	Double-precision floating point
float	Single-precision floating point
int	Integer
long	Long integer
short	Short integer

Table 2-1 Java's Built-in Primitive Data Types

Integers

Java defines four integer types: **byte**, **short**, **int**, and **long**, which are shown here:

Type	Width in Bits	Range
byte	8	–128 to 127
short	16	–32,768 to 32,767
int	32	–2,147,483,648 to 2,147,483,647
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

As the table shows, all of the integer types are signed positive and negative values. Java does not support unsigned (positive-only) integers. Many other computer languages support both signed and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary.

Technically, the Java run-time system can use any size it wants to store a primitive type. However, in all cases, types must act as specified.

The most commonly used integer type is **int**. Variables of type **int** are often employed to control loops, to index arrays, and to perform general-purpose integer math.

When you need an integer that has a range greater than **int**, use **long**. For example, here is a program that computes the number of cubic inches contained in a cube that is one mile by one mile, by one mile:

```
/*
   Compute the number of cubic inches
   in 1 cubic mile.
*/
class Inches {
  public static void main(String args[]) {
    long ci;
    long im;

    im = 5280 * 12;

    ci = im * im * im;

    System.out.println("There are " + ci +
                       " cubic inches in cubic mile.");

  }
}
```

Here is the output from the program:

```
There are 254358061056000 cubic inches in cubic mile.
```

Clearly, the result could not have been held in an **int** variable.

The smallest integer type is **byte**. Variables of type **byte** are especially useful when working with raw binary data that may not be directly compatible with Java's other built-in types.

The **short** type creates a short integer that has its high-order byte first (called *big-endian* format).

Floating-Point Types

As explained in Module 1, the floating-point types can represent numbers that have fractional components. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Type **float** is 32 bits wide and type **double** is 64 bits wide.

Ask the Expert

Q: **What is *endianness*?**

A: *Endianness* describes how an integer is stored in memory. There are two possible ways to approach storage. The first way stores the most significant byte first. This is called big-endian. The other stores the least significant byte first, which is little-endian. Little-endian is the most common method because it is used by the Intel Pentium processor.

Of the two, **double** is the most commonly used because all of the math functions in Java's class library use **double** values. For example, the **sqrt( )** method (which is defined by the standard ***Math*** class) returns a ***double*** value that is the square root of its ***double*** argument. Here, **sqrt( )** is used to compute the length of the hypotenuse, given the lengths of the two opposing sides:

```
/*
   Use the Pythagorean theorem to
   find the length of the hypotenuse
   given the lengths of the two opposing
   sides.
*/
class Hypot {
  public static void main(String args[]) {
    double x, y, z;

    x = 3;
    y = 4;

    z = Math.sqrt(x*x + y*y);

    System.out.println("Hypotenuse is " +z);
  }
}
```

Notice how **sqrt( )** is called. It is preceded by the name of the class of which it is a member.

The output from the program is shown here:

```
Hypotenuse is 5.0
```

One other point about the preceding example: As mentioned, **sqrt( )** is a member of the standard **Math** class. Notice how **sqrt( )** is called; it is preceded by the name **Math**. This is similar to the way **System.out** precedes **println( )**. Although not all standard methods are called by specifying their class name first, several are.

Characters

In Java, characters are not 8-bit quantities like they are in most other computer languages. Instead, Java uses Unicode. *Unicode* defines a character set that can represent all of the characters found in all human languages. Thus, in Java, **char** is an unsigned 16-bit type having a range of 0 to 65,536. The standard 8-bit ASCII character set is a subset of Unicode and ranges from 0 to 127. Thus, the ASCII characters are still valid Java characters.

A character variable can be assigned a value by enclosing the character in single quotes. For example, this assigns the variable **ch** the letter X:

```
char ch;
ch = 'X';
```

You can output a **char** value using a **println( )** statement. For example, this line outputs the value in **ch**:

```
System.out.println("This is ch: " + ch);
```

Since **char** is an unsigned 16-bit type, it is possible to perform various arithmetic manipulations on a **char** variable. For example, consider the following program:

```
// Character variables can be handled like integers.
class CharArithDemo {
  public static void main(String args[]) {
    char ch;

    ch = 'X';
    System.out.println("ch contains " + ch);

    ch++; // increment ch
    System.out.println("ch is now " + ch);

    ch = 90; // give ch the value Z
    System.out.println("ch is now " + ch);
  }
}
```

A **char** can be incremented.

A **char** can be assigned an integer value.

Ask the Expert

Q: **Why does Java use Unicode?**

A: Java was designed for worldwide use. Thus, it needs to use a character set that can represent all the world's languages. Unicode is the standard character set designed expressly for this purpose. Of course, the use of Unicode is inefficient for languages such as English, German, Spanish, or French, whose characters can be contained within 8 bits. But such is the price that must be paid for global portability.

The output generated by this program is shown here:

```
ch contains X
ch is now Y
ch is now Z
```

In the program, **ch** is first given the value X. Next, **ch** is incremented. This results in **ch** containing Y, the next character in the ASCII (and Unicode) sequence. Although **char** is not an integer type, in some cases it can be handled as if it were. Next, **ch** is assigned the value 90, which is the ASCII (and Unicode) value that corresponds to the letter Z. Since the ASCII character set occupies the first 127 values in the Unicode character set, all the "old tricks" that you have used with characters in the past will work in Java, too.

The Boolean Type

The **boolean** type represents true/false values. Java defines the values true and false using the reserved words **true** and **false**. Thus, a variable or expression of type **boolean** will be one of these two values.

Here is a program that demonstrates the **boolean** type:

```
// Demonstrate boolean values.
class BoolDemo {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);
```

```
    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
    if(b) System.out.println("This is not executed.");

    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

There are three interesting things to notice about this program. First, as you can see, when a **boolean** value is output by **println( )**, "true" or "false" is displayed. Second, the value of a **boolean** variable is sufficient, by itself, to control the **if** statement. There is no need to write an **if** statement like this:

```
if(b == true) ...
```

Third, the outcome of a relational operator, such as <, is a **boolean** value. This is why the expression **10 > 9** displays the value "true." Further, the extra set of parentheses around **10 > 9** is necessary because the + operator has a higher precedence than the >.

Progress Check

1. What are Java's integer types?
2. What is Unicode?
3. What values can a **boolean** variable have?

1. Java's integer types are **byte**, **short**, **int**, and **long**.
2. Unicode is an international character set.
3. Variables of type **boolean** can be either **true** or **false**.

Project 2-1 How Far Away Is the Lightning?

`Sound.java`

In this project you will create a program that computes how far away, in feet, a listener is from a lightning strike. Sound travels approximately 1,100 feet per second through air. Thus, knowing the interval between the time you see a lightning bolt and the time the sound reaches you enables you to compute the distance to the lightning. For this project, assume that the time interval is 7.2 seconds.

Step by Step

1. Create a new file called **Sound.java**.
2. To compute the distance, you will need to use floating-point values. Why? Because the time interval, 7.2, has a fractional component. Although it would be permissible to use a value of type **float**, we will use **double** in the example.
3. To compute the distance, you will multiply 7.2 by 1,100. You will then assign this value to a variable.
4. Finally, you will display the result.

   Here is the entire **Sound.java** program listing:

```
/*
   Project 2-1
   Compute the distance to a lightning
   strike whose sound takes 7.2 seconds
   to reach you.
*/
class Sound {
  public static void main(String args[]) {
    double dist;

    dist = 7.2 * 1100;

    System.out.println("The lightning is " + dist +
                       " feet away.");

  }
}
```

5. Compile and run the program. The following result is displayed:

```
The lightning is 7920.0 feet away.
```

(continued)

6. Extra challenge: You can compute the distance to a large object, such as a rock wall, by timing the echo. For example, if you clap your hands and time how long it takes for you to hear the echo, then you know the total round-trip time. Dividing this value by two yields the time it takes the sound to go one way. You can then use this value to compute the distance to the object. Modify the preceding program so that it computes the distance, assuming that the time interval is that of an echo.

CRITICAL SKILL 2.2

Literals

In Java, *literals* refer to fixed values that are represented in their human-readable form. For example, the number 100 is a literal. Literals are also commonly called *constants*. For the most part, literals, and their usage, are so intuitive that they have been used in one form or another by all the preceding sample programs. Now the time has come to explain them formally.

Java literals can be of any of the primitive data types. The way each literal is represented depends upon its type. As explained earlier, character constants are enclosed in single quotes. For example, 'a' and ' %' are both character constants.

Integer constants are specified as numbers without fractional components. For example, 10 and –100 are integer constants. Floating-point constants require the use of the decimal point followed by the number's fractional component. For example, 11.123 is a floating-point constant. Java also allows you to use scientific notation for floating-point numbers.

By default, integer literals are of type **int**. If you want to specify a **long** literal, append an l or an L. For example, 12 is an **int**, but 12L is a **long**.

By default, floating-point literals are of type **double**. To specify a **float** literal, append an F or f to the constant. For example, 10.19F is of type **float**.

Although integer literals create an **int** value by default, they can still be assigned to variables of type **char**, **byte**, or **short** as long as the value being assigned can be represented by the target type. An integer literal can always be assigned to a **long** variable.

Hexadecimal and Octal Constants

As you probably know, in programming it is sometimes easier to use a number system based on 8 or 16 instead of 10. The number system based on 8 is called *octal,* and it uses the digits 0 through 7. In octal the number 10 is the same as 8 in decimal. The base 16 number system is called *hexadecimal* and uses the digits 0 through 9 plus the letters A through F, which stand for 10, 11, 12, 13, 14, and 15. For example, the hexadecimal number 10 is 16 in decimal. Because of the frequency with which these two number systems are used, Java allows you to specify integer constants in hexadecimal or octal instead of decimal. A hexadecimal constant must begin with 0x (a zero followed by an x). An octal constant begins with a zero. Here are some examples:

```
hex = 0xFF; // 255 in decimal
oct = 011;  // 9 in decimal
```

Character Escape Sequences

Enclosing character constants in single quotes works for most printing characters, but a few characters, such as the carriage return, pose a special problem when a text editor is used. In addition, certain other characters, such as the single and double quotes, have special meaning in Java, so you cannot use them directly. For these reasons, Java provides special *escape sequences*, sometimes referred to as backslash character constants, shown in Table 2-2. These sequences are used in place of the characters that they represent.

For example, this assigns **ch** the tab character:

```
ch = '\t';
```

The next example assigns a single quote to **ch**:

```
ch = '\'';
```

String Literals

Java supports one other type of literal: the string. A *string* is a set of characters enclosed by double quotes. For example,

```
"this is a test"
```

Escape Sequence	Description
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line
\f	Form feed
\t	Horizontal tab
\b	Backspace
ddd	Octal constant (where *ddd* is an octal constant)
\u*xxxx*	Hexadecimal constant (where *xxxx* is a hexadecimal constant)

Table 2-2 Character Escape Sequences

is a string. You have seen examples of strings in many of the **println( )** statements in the preceding sample programs.

In addition to normal characters, a string literal can also contain one or more of the escape sequences just described. For example, consider the following program. It uses the **\n** and **\t** escape sequences.

```
// Demonstrate escape sequences in strings.
class StrDemo {
  public static void main(String args[]) {
    System.out.println("First line\nSecond line");
    System.out.println("A\tB\tC");
    System.out.println("D\tE\tF") ;
  }
}
```

Use \n to generate a new line.

Use tabs to align output.

The output is shown here:

```
First line
Second line
A       B       C
D       E       F
```

Notice how the **\n** escape sequence is used to generate a new line. You don't need to use multiple **println( )** statements to get multiline output. Just embed **\n** within a longer string at the points where you want the new lines to occur.

Progress Check

1. What is the type of the literal 10? What is the type of the literal 10.0?
2. How do you specify a **long** literal?
3. Is "x" a string or a character literal?

1. The literal 10 is an **int**, and 10.0 is a **double**.
2. A **long** literal is specified by adding the L or l suffix. For example, 100L.
3. The literal "x" is a string.

Ask the Expert

Q: **Is a string consisting of a single character the same as a character literal? For example, is "k" the same as 'k'?**

A: No. You must not confuse strings with characters. A character literal represents a single letter of type **char**. A string containing only one letter is still a string. Although strings consist of characters, they are not the same type.

CRITICAL SKILL 2.3

A Closer Look at Variables

Variables were introduced in Module 1. Here, we will take a closer look at them. As you learned earlier, variables are declared using this form of statement,

type var-name;

where *type* is the data type of the variable, and *var-name* is its name. You can declare a variable of any valid type, including the simple types just described. When you create a variable, you are creating an instance of its type. Thus, the capabilities of a variable are determined by its type. For example, a variable of type **boolean** cannot be used to store floating-point values. Furthermore, the type of a variable cannot change during its lifetime. An **int** variable cannot turn into a **char** variable, for example.

All variables in Java must be declared prior to their use. This is necessary because the compiler must know what type of data a variable contains before it can properly compile any statement that uses the variable. It also enables Java to perform strict type checking.

Initializing a Variable

In general, you must give a variable a value prior to using it. One way to give a variable a value is through an assignment statement, as you have already seen. Another way is by giving it an initial value when it is declared. To do this, follow the variable's name with an equal sign and the value being assigned. The general form of initialization is shown here:

type var = *value*;

Here, *value* is the value that is given to *var* when *var* is created. The value must be compatible with the specified type. Here are some examples:

```
int count = 10; // give count an initial value of 10
char ch = 'X';  // initialize ch with the letter X
float f = 1.2F; // f is initialized with 1.2
```

When declaring two or more variables of the same type using a comma-separated list, you can give one or more of those variables an initial value. For example:

```
int a, b = 8, c = 19, d; // b and c have initializations
```

In this case, only **b** and **c** are initialized.

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example, here is a short program that computes the volume of a cylinder given the radius of its base and its height:

```
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
      double radius = 4, height = 5;

      // dynamically initialize volume
      double volume = 3.1416 * radius * radius * height;

      System.out.println("Volume is " + volume);
    }
}
```

volume is dynamically initialized at run time.

Here, three local variables—**radius**, **height**, and **volume**—are declared. The first two, **radius** and **height**, are initialized by constants. However, **volume** is initialized dynamically to the volume of the cylinder. The key point here is that the initialization expression can use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

CRITICAL SKILL 2.4

The Scope and Lifetime of Variables

So far, all of the variables that we have been using were declared at the start of the **main( )** method. However, Java allows variables to be declared within any block. As explained in Module 1, a block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope.* Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Most other computer languages define two general categories of scopes: global and local. Although supported by Java, these are not the best ways to categorize Java's scopes. The most important scopes in Java are those defined by a class and those defined by a method. A discussion of class scope (and variables declared within it) is deferred until later in this book, when classes are described. For now, we will examine only the scopes defined by or within a method.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope.

As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation.

Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

```
// Demonstrate block scope.
class ScopeDemo {
  public static void main(String args[]) {
    int x; // known to all code within main

    x = 10;
    if(x == 10) { // start new scope

      int y = 20; // known only to this block

       // x and y both known here.
```

```
      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here ←— Here, y is outside of its scope.

    // x is still known here.
    System.out.println("x is " + x);
  }
}
```

As the comments indicate, the variable **x** is declared at the start of **main( )**'s scope and is accessible to all subsequent code within **main( )**. Within the **if** block, **y** is declared. Since a block defines a scope, **y** is visible only to other code within its block. This is why outside of its block, the line **y = 100;** is commented out. If you remove the leading comment symbol, a compile-time error will occur, because **y** is not visible outside of its block. Within the **if** block, **x** can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it.

Here is another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

If a variable declaration includes an initializer, that variable will be reinitialized each time the block in which it is declared is entered. For example, consider this program:

```
// Demonstrate lifetime of a variable.
class VarInitDemo {
  public static void main(String args[]) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y is initialized each time block is entered
      System.out.println("y is: " + y); // this always prints -1
      y = 100;
      System.out.println("y is now: " + y);
    }
  }
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

As you can see, **y** is always reinitialized to –1 each time the inner **for** loop is entered. Even though it is subsequently assigned the value 100, this value is lost.

There is one quirk to Java's scope rules that may surprise you: although blocks can be nested, no variable declared within an inner scope can have the same name as a variable declared by an enclosing scope. For example, the following program, which tries to declare two separate variables with the same name, will not compile.

```
/*
   This program attempts to declare a variable
   in an inner scope with the same name as one
   defined in an outer scope.

   *** This program will not compile. ***
*/
class NestVar {
  public static void main(String args[]) {
    int count;

    for(count = 0; count < 10; count = count+1) {
      System.out.println("This is count: " + count);

      int count; // illegal!!!
      for(count = 0; count < 2; count++)
        System.out.println("This program is in error!");
    }
  }
}
```

Can't declare **count** again because it's already declared.

If you come from a C/C++ background, you know that there is no restriction on the names that you give variables declared in an inner scope. Thus, in C/C++ the declaration of **count** within the block of the outer **for** loop is completely valid, and such a declaration hides the outer variable. The designers of Java felt that this *name hiding* could easily lead to programming errors and disallowed it.

Progress Check

1. What is a scope? How can one be created?
2. Where in a block can variables be declared?
3. In a block, when is a variable created? When is it destroyed?

Operators

Java provides a rich operator environment. An *operator* is a symbol that tells the compiler to perform a specific mathematical or logical manipulation. Java has four general classes of operators: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations. This module will examine the arithmetic, relational, and logical operators. We will also examine the assignment operator. The bitwise and other special operators are examined later.

CRITICAL SKILL 2.5 Arithmetic Operators

Java defines the following arithmetic operators:

Operator	Meaning
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
– –	Decrement

1. A scope defines the visibility and lifetime of an object. A block defines a scope.
2. A variable can be defined at any point within a block.
3. Inside a block, a variable is created when its declaration is encountered. It is destroyed when the block exits.

The operators +, –, *, and / all work the same way in Java as they do in any other computer language (or algebra, for that matter). These can be applied to any built-in numeric data type. They can also be used on objects of type **char**.

Although the actions of arithmetic operators are well known to all readers, a few special situations warrant some explanation. First, remember that when / is applied to an integer, any remainder will be truncated; for example, 10/3 will equal 3 in integer division. You can obtain the remainder of this division by using the modulus operator **%**. It works in Java the way it does in other languages: it yields the remainder of an integer division. For example, 10 % 3 is 1. In Java, the **%** can be applied to both integer and floating-point types. Thus, 10.0 % 3.0 is also 1. The following program demonstrates the modulus operator.

```
// Demonstrate the % operator.
class ModDemo {
  public static void main(String args[]) {
    int iresult, irem;
    double dresult, drem;

    iresult = 10 / 3;
    irem = 10 % 3;

    dresult = 10.0 / 3.0;
    drem = 10.0 % 3.0;

    System.out.println("Result and remainder of 10 / 3: " +
                       iresult + " " + irem);
    System.out.println("Result and remainder of 10.0 / 3.0: " +
                       dresult + " " + drem);

  }
}
```

The output from the program is shown here:

```
Result and remainder of 10 / 3: 3 1
Result and remainder of 10.0 / 3.0: 3.3333333333333335 1.0
```

As you can see, the **%** yields a remainder of 1 for both integer and floating-point operations.

Increment and Decrement

Introduced in Module 1, the ++ and the – – are Java's increment and decrement operators. As you will see, they have some special properties that make them quite interesting. Let's begin by reviewing precisely what the increment and decrement operators do.

The increment operator adds 1 to its operand, and the decrement operator subtracts 1. Therefore,

```
x = x + 1;
```

is the same as

```
x++;
```

and

```
x = x - 1;
```

is the same as

```
--x;
```

Both the increment and decrement operators can either precede (prefix) or follow (postfix) the operand. For example,

```
x = x + 1;
```

can be written as

```
++x; // prefix form
```

or as

```
x++; // postfix form
```

In the foregoing example, there is no difference whether the increment is applied as a prefix or a postfix. However, when an increment or decrement is used as part of a larger expression, there is an important difference. When an increment or decrement operator precedes its operand, Java will perform the corresponding operation prior to obtaining the operand's value for use by the rest of the expression. If the operator follows its operand, Java will obtain the operand's value before incrementing or decrementing it. Consider the following:

```
x = 10;
y = ++x;
```

In this case, **y** will be set to 11. However, if the code is written as

```
x = 10;
y = x++;
```

then **y** will be set to 10. In both cases, **x** is still set to 11; the difference is when it happens. There are significant advantages in being able to control when the increment or decrement operation takes place.

CRITICAL SKILL 2.6

Relational and Logical Operators

In the terms *relational operator* and *logical operator, relational* refers to the relationships that values can have with one another, and *logical* refers to the ways in which true and false values can be connected together. Since the relational operators produce true or false results, they often work with the logical operators. For this reason they will be discussed together here.

The relational operators are shown here:

Operator	Meaning
= =	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The logical operators are shown next:

Operator	Meaning
&	AND
|	OR
^	XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	NOT

The outcome of the relational and logical operators is a **boolean** value.

In Java, all objects can be compared for equality or inequality using = = and **!=**. However, the comparison operators, **<**, **>**, **<=**, or **>=**, can be applied only to those types that support an ordering relationship. Therefore, all of the relational operators can be applied to all numeric types and to type **char**. However, values of type **boolean** can only be compared for equality or inequality, since the **true** and **false** values are not ordered. For example, **true > false** has no meaning in Java.

For the logical operators, the operands must be of type **boolean**, and the result of a logical operation is of type **boolean**. The logical operators, **&**, |, ^, and **!**, support the basic logical operations AND, OR, XOR, and NOT, according to the following truth table.

p	q	p & q	p | q	p ^ q	!p
False	False	False	False	False	True
True	False	False	True	True	False
False	True	False	True	True	True
True	True	True	True	False	False

As the table shows, the outcome of an exclusive OR operation is true when exactly one and only one operand is true.

Here is a program that demonstrates several of the relational and logical operators:

```
// Demonstrate the relational and logical operators.
class RelLogOps {
  public static void main(String args[]) {
    int i, j;
    boolean b1, b2;

    i = 10;
    j = 11;
    if(i < j) System.out.println("i < j");
    if(i <= j) System.out.println("i <= j");
    if(i != j) System.out.println("i != j");
    if(i == j) System.out.println("this won't execute");
    if(i >= j) System.out.println("this won't execute");
    if(i > j) System.out.println("this won't execute");

    b1 = true;
    b2 = false;
    if(b1 & b2) System.out.println("this won't execute");
    if(!(b1 & b2)) System.out.println("!(b1 & b2) is true");
    if(b1 | b2) System.out.println("b1 | b2 is true");
    if(b1 ^ b2) System.out.println("b1 ^ b2 is true");
  }
}
```

The output from the program is shown here:

```
i < j
i <= j
i != j
!(b1 & b2) is true
b1 | b2 is true
b1 ^ b2 is true
```

Short-Circuit Logical Operators

Java supplies special *short-circuit* versions of its AND and OR logical operators that can be used to produce more efficient code. To understand why, consider the following. In an AND operation, if the first operand is false, the outcome is false no matter what value the second operand has. In an OR operation, if the first operand is true, the outcome of the operation is true no matter what the value of the second operand. Thus, in these two cases there is no need to evaluate the second operand. By not evaluating the second operand, time is saved and more efficient code is produced.

The short-circuit AND operator is **&&**, and the short-circuit OR operator is **||**. Their normal counterparts are **&** and **|**. The only difference between the normal and short- circuit versions is that the normal operands will always evaluate each operand, but short-circuit versions will evaluate the second operand only when necessary.

Here is a program that demonstrates the short-circuit AND operator. The program determines whether the value in **d** is a factor of **n**. It does this by performing a modulus operation. If the remainder of **n / d** is zero, then **d** is a factor. However, since the modulus operation involves a division, the short-circuit form of the AND is used to prevent a divide-by-zero error.

```
// Demonstrate the short-circuit operators.
class SCops {
  public static void main(String args[]) {
    int n, d, q;

    n = 10;
    d = 2;
    if(d != 0 && (n % d) == 0)
      System.out.println(d + " is a factor of " + n);

    d = 0; // now, set d to zero

    // Since d is zero, the second operand is not evaluated.
    if(d != 0 && (n % d) == 0)
      System.out.println(d + " is a factor of " + n);

    /* Now, try same thing without short-circuit operator.
```

The short-circuit operator prevents a division by zero.

```
      This will cause a divide-by-zero error.
    */
    if(d != 0 & (n % d) == 0)
      System.out.println(d + " is a factor of " + n);
  }
}
```

Now both expressions are evaluated, allowing a division by zero to occur.

To prevent a divide-by-zero, the **if** statement first checks to see if **d** is equal to zero. If it is, the short-circuit AND stops at that point and does not perform the modulus division. Thus, in the first test, **d** is 2 and the modulus operation is performed. The second test fails because **d** is set to zero, and the modulus operation is skipped, avoiding a divide-by-zero error. Finally, the normal AND operator is tried. This causes both operands to be evaluated, which leads to a run-time error when the division by zero occurs.

Progress Check

1. What does the **%** operator do? To what types can it be applied?
2. What type of values can be used as operands of the logical operators?
3. Does a short-circuit operator always evaluate both of its operands?

CRITICAL SKILL 2.7

The Assignment Operator

You have been using the assignment operator since Module 1. Now it is time to take a formal look at it. The *assignment operator* is the single equal sign, =. This operator works in Java much as it does in any other computer language. It has this general form:

var = *expression*;

Here, the type of *var* must be compatible with the type of *expression.*

1. The **%** is the modulus operator, which returns the remainder of an integer division. It can be applied to all of the numeric types.
2. The logical operators must have operands of type **boolean**.
3. No, a short-circuit operator evaluates its second operand only if the outcome of the operation cannot be determined solely by its first operand.

Ask the Expert

Q: **Since the short-circuit operators are, in some cases, more efficient than their normal counterparts, why does Java still offer the normal AND and OR operators?**

A: In some cases you will want both operands of an AND or OR operation to be evaluated because of the side effects produced. Consider the following:

```
// Side effects can be important.
class SideEffects {
  public static void main(String args[]) {
    int i;

    i = 0;

    /* Here, i is still incremented even though
       the if statement fails. */
    if(false & (++i < 100))
       System.out.println("this won't be displayed");
    System.out.println("if statements executed: " + i); // displays 1

    /* In this case, i is not incremented because
       the short-circuit operator skips the increment. */
    if(false && (++i < 100))
      System.out.println("this won't be displayed");
    System.out.println("if statements executed: " + i); // still 1 !!
  }
}
```

As the comments indicate, in the first **if** statement, **i** is incremented whether the **if** succeeds or not. However, when the short-circuit operator is used, the variable **i** is not incremented when the first operand is false. The lesson here is that if your code expects the right-hand operand of an AND or OR operation to be evaluated, you must use Java's non-short-circuit forms of these operations.

The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;

x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement. This works because the **=** is an operator that yields the value of the right-hand expression. Thus, the value of **z = 100**

is 100, which is then assigned to **y**, which in turn is assigned to **x**. Using a "chain of assignment" is an easy way to set a group of variables to a common value.

CRITICAL SKILL 2.8

Shorthand Assignments

Java provides special *shorthand* assignment operators that simplify the coding of certain assignment statements. Let's begin with an example. The assignment statement shown here

```
x = x + 10;
```

can be written, using Java shorthand, as

```
x += 10;
```

The operator pair += tells the compiler to assign to **x** the value of **x** plus 10.

Here is another example. The statement

```
x = x - 100;
```

is the same as

```
x -= 100;
```

Both statements assign to **x** the value of **x** minus 100.

This shorthand will work for all the binary operators in Java (that is, those that require two operands). The general form of the shorthand is

var op = expression;

Thus, the arithmetic and logical assignment operators are the following:

+=	**–=**	***=**	**/=**
%=	**&=**	**|=**	**^=**

Because these operators combine an operation with an assignment, they are formally referred to as *compound assignment operators*.

The compound assignment operators provide two benefits. First, they are more compact than their "longhand" equivalents. Second, they are implemented more efficiently by the Java run-time system. For these reasons, you will often see the compound assignment operators used in professionally written Java programs.

CRITICAL SKILL 2.9

Type Conversion in Assignments

In programming, it is common to assign one type of variable to another. For example, you might want to assign an **int** value to a **float** variable, as shown here:

```
int i;
float f;

i = 10;
f = i; // assign an int to a float
```

When compatible types are mixed in an assignment, the value of the right side is automatically converted to the type of the left side. Thus, in the preceding fragment, the value in **i** is converted into a **float** and then assigned to **f**. However, because of Java's strict type checking, not all types are compatible, and thus, not all type conversions are implicitly allowed. For example, **boolean** and **int** are not compatible.

When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte** values, and both **int** and **byte** are integer types, so an automatic conversion from **byte** to **int** can be applied.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. For example, the following program is perfectly valid since **long** to **double** is a widening conversion that is automatically performed.

```
// Demonstrate automatic conversion from long to double.
class LtoD {
  public static void main(String args[]) {
    long L;
    double D;

    L = 100123285L;
    D = L; <-- Automatic conversion from long to double

    System.out.println("L and D: " + L + " " + D);

  }
}
```

Although there is an automatic conversion from **long** to **double**, there is no automatic conversion from **double** to **long** since this is not a widening conversion. Thus, the following version of the preceding program is invalid.

```
// *** This program will not compile. ***
class LtoD {
  public static void main(String args[]) {
    long L;
    double D;

    D = 100123285.0;
    L = D; // Illegal!!!  <-- No automatic conversion from double to long

    System.out.println("L and D: " + L + " " + D);

  }
}
```

There are no automatic conversions from the numeric types to **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other. However, an integer literal can be assigned to **char**.

CRITICAL SKILL 2.10

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all programming needs because they apply only to widening conversions between compatible types. For all other cases you must employ a cast. A *cast* is an instruction to the compiler to convert one type into another. Thus, it requests an explicit type conversion. A cast has this general form:

(*target-type*) *expression*

Here, *target-type* specifies the desired type to convert the specified expression to. For example, if you want to convert the type of the expression **x/y** to **int**, you can write

```
double x, y;
// ...
(int) (x / y)
```

Here, even though **x** and **y** are of type **double**, the cast converts the outcome of the expression to **int**. The parentheses surrounding **x / y** are necessary. Otherwise, the cast to **int** would apply only to the **x** and not to the outcome of the division. The cast is necessary here because there is no automatic conversion from **double** to **int**.

When a cast involves a *narrowing conversion,* information might be lost. For example, when casting a **long** into a **short**, information will be lost if the **long**'s value is greater than

the range of a **short** because its high-order bits are removed. When a floating-point value is cast to an integer type, the fractional component will also be lost due to truncation. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 is lost.

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casting.
class CastDemo {
  public static void main(String args[]) {
    double x, y;
    byte b;
    int i;
    char ch;

    x = 10.0;
    y = 3.0;
    i = (int) (x / y); // cast double to int
    System.out.println("Integer outcome of x / y: " + i);

    i = 100;
    b = (byte) i;
    System.out.println("Value of b: " + b);

    i = 257;
    b = (byte) i;
    System.out.println("Value of b: " + b);

    b = 88; // ASCII code for X
    ch = (char) b;
    System.out.println("ch: " + ch);
  }
}
```

Truncation will occur in this conversion.

No loss of info here. A **byte** can hold the value 100.

Information loss this time. A **byte** cannot hold the value 257.

Cast between incompatible types

The output from the program is shown here:

```
Integer outcome of x / y: 3
Value of b: 100
Value of b: 1
ch: X
```

In the program, the cast of **(x / y)** to **int** results in the truncation of the fractional component, and information is lost. Next, no loss of information occurs when **b** is assigned the value 100 because a **byte** can hold the value 100. However, when the attempt is made to assign **b** the value 257, information loss occurs because 257 exceeds a **byte**'s maximum value. Finally, no information is lost, but a cast is needed when assigning a **byte** value to a **char**.

Progress Check

1. What is a cast?
2. Can a **short** be assigned to an **int** without a cast? Can a **byte** be assigned to a **char** without a cast?
3. How can the following statement be rewritten?

```
x = x + 23;
```

Operator Precedence

The following table shows the order of precedence for all Java operators, from highest to lowest. This table includes several operators that will be discussed later in this book.

highest			
()	[]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	*op*=		
lowest			

1. A cast is an explicit conversion.
2. Yes. No.
3. x += 23;

Project 2-2 Display a Truth Table for the Logical Operators

LogicalOpTable.java

In this project you will create a program that displays the truth table for Java's logical operators. You must make the columns in the table line up. This project makes use of several features covered in this module, including one of Java's escape sequences and the logical operators. It also illustrates the differences in the precedence between the arithmetic + operator and the logical operators.

Step by Step

1. Create a new file called **LogicalOpTable.java**.
2. To ensure that the columns line up, you will use the **\t** escape sequence to embed tabs into each output string. For example, this **println( )** statement displays the header for the table:

```
System.out.println("P\tQ\tAND\tOR\tXOR\tNOT");
```

3. Each subsequent line in the table will use tabs to position the outcome of each operation under its proper heading.
4. Here is the entire **LogicalOpTable.java** program listing. Enter it at this time.

```
// Project 2-2: a truth table for the logical operators.
class LogicalOpTable {
  public static void main(String args[]) {

    boolean p, q;

    System.out.println("P\tQ\tAND\tOR\tXOR\tNOT");

    p = true; q = true;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));

    p = true; q = false;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));

    p = false; q = true;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));
```

(continued)

```
    p = false; q = false;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));
  }
}
```

Notice the parentheses surrounding the logical operations inside the **println( )** statements. They are necessary because of the precedence of Java's operators. The + operator is higher than the logical operators.

5. Compile and run the program. The following table is displayed.

```
P       Q       AND     OR      XOR     NOT
true    true    true    true    false   false
true    false   false   true    true    false
false   true    false   true    true    true
false   false   false   false   false   true
```

6. On your own, try modifying the program so that it uses and displays 1's and 0's, rather than true and false. This may involve a bit more effort than you might at first think!

CRITICAL SKILL 2.11

Expressions

Operators, variables, and literals are the constituents of *expressions*. An expression in Java is any valid combination of those pieces. You probably already know the general form of an expression from your other programming experience, or from algebra. However, a few aspects of expressions will be discussed now.

Type Conversion in Expressions

Within an expression, it is possible to mix two or more different types of data as long as they are compatible with each other. For example, you can mix **short** and **long** within an expression because they are both numeric types. When different types of data are mixed within an expression, they are all converted to the same type. This is accomplished through the use of Java's *type promotion rules*.

First, all **char**, **byte**, and **short** values are promoted to **int**. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float** operand, the entire expression is promoted to **float**. If any of the operands is **double**, the result is **double**.

It is important to understand that type promotions apply only to the values operated upon when an expression is evaluated. For example, if the value of a **byte** variable is promoted to **int** inside an expression, outside the expression, the variable is still a **byte**. Type promotion only affects the evaluation of an expression.

Type promotion can, however, lead to somewhat unexpected results. For example, when an arithmetic operation involves two **byte** values, the following sequence occurs: First, the **byte** operands are promoted to **int**. Then the operation takes place, yielding an **int** result. Thus, the outcome of an operation involving two **byte** values will be an **int**. This is not what you might intuitively expect. Consider the following program:

```
// A promotion surprise!
class PromDemo {
  public static void main(String args[]) {
    byte b;
    int i;

    b = 10;
    i = b * b; // OK, no cast needed

    b = 10;
    b = (byte) (b * b); // cast needed!!

    System.out.println("i and b: " + i + " " + b);
  }
}
```

No cast needed because result is already elevated to **int**.

Cast is needed here to assign an **int** to a **byte**!

Somewhat counterintuitively, no cast is needed when assigning **b * b** to **i**, because **b** is promoted to **int** when the expression is evaluated. However, when you try to assign **b * b** to **b**, you do need a cast—back to **byte**! Keep this in mind if you get unexpected type-incompatibility error messages on expressions that would otherwise seem perfectly OK.

This same sort of situation also occurs when performing operations on **char**s. For example, in the following fragment, the cast back to **char** is needed because of the promotion of **ch1** and **ch2** to **int** within the expression.

```
char ch1 = 'a', ch2 = 'b';

ch1 = (char) (ch1 + ch2);
```

Without the cast, the result of adding **ch1** to **ch2** would be **int**, which can't be assigned to a **char**.

Casts are not only useful when converting between types in an assignment. For example, consider the following program. It uses a cast to **double** to obtain a fractional component from an otherwise integer division.

```
// Using a cast.
class UseCast {
  public static void main(String args[]) {
    int i;
```

```
    for(i = 0; i < 5; i++) {
      System.out.println(i + " / 3: " + i / 3);
      System.out.println(i + " / 3 with fractions: "
                         + (double) i / 3);
      System.out.println();
    }
  }
}
```

The output from the program is shown here:

```
0 / 3: 0
0 / 3 with fractions: 0.0

1 / 3: 0
1 / 3 with fractions: 0.3333333333333333

2 / 3: 0
2 / 3 with fractions: 0.6666666666666666

3 / 3: 1
3 / 3 with fractions: 1.0

4 / 3: 1
4 / 3 with fractions: 1.3333333333333333
```

Spacing and Parentheses

An expression in Java may have tabs and spaces in it to make it more readable. For example, the following two expressions are the same, but the second is easier to read:

```
x=10/y*(127/x);

x = 10 / y * (127/x);
```

Parentheses increase the precedence of the operations contained within them, just like in algebra. Use of redundant or additional parentheses will not cause errors or slow down the execution of the expression. You are encouraged to use parentheses to make clear the exact order of evaluation, both for yourself and for others who may have to figure out your program later. For example, which of the following two expressions is easier to read?

```
x = y/3-34*temp+127;

x = (y/3) - (34*temp) + 127;
```