The ABCs of Programming

A Java program is made up of classes and objects, which, in turn, are made up of methods and variables. Methods are made up of statements and expressions, which are made up of operators.

At this point, you might be worried that Java is like a set of Russian nesting matryoshka dolls. Each doll has a smaller doll inside it, as intricate and detailed as its larger companion, until you reach the smallest one.

Today's lesson clears away the big dolls to reveal the smallest elements of Java programming. You will set aside classes, objects, and methods for a day and examine the basic things you can do in a single line of Java code.

The following subjects are covered:

- Statements and expressions
- Variables and primitive data types
- Constants
- Comments
- Literals
- Arithmetic
- Comparisons
- Logical operators

Statements and Expressions

All the tasks you want to accomplish in a Java program can be broken into a series of statements. In a programming language, a *statement* is a simple command that causes something to happen.

Statements represent a single action taken in a Java program. Here are three simple Java statements:

```
int weight = 225;
System.out.println("Free the bound periodicals!");
song.duration = 230;
```

Some statements can convey a value, such as when two numbers are added or two variables are compared to find out if they are equal.

A statement that produces a value is called an *expression*. The value can be stored for later use in the program, used immediately in another statement, or disregarded. The value produced by a statement is called its *return value*.

Some expressions produce a numeric return value, as when two numbers are added or multiplied. Others produce a Boolean value—either `true` or `false`—or even can produce a Java object. They are discussed later today.

Although many Java programs contain one statement per line, this is a formatting decision that does not determine where one statement ends and another one begins. Each statement in Java is terminated with a semicolon character `;`. A programmer can put more than one statement on a line and it will compile successfully, as in the following example:

```
spirit.speed = 2; spirit.temperature = -60;
```

To make your program more readable to other programmers (and yourself), you should follow the convention of putting only one statement on each line.

Statements in Java are grouped using an opening brace `{` and a closing brace `}`. A group of statements organized between these characters is called a block (or block statement). You learn more about them later during "Lists, Logic, and Loops."

Variables and Data Types

A variable is a place where information can be
stored while a program is running. The value can be changed at any point in the program—hence the name.

To create a variable, you must give it a name and identify the type of information it will store. You also can give a variable an initial value at the same time you create it.

Java has three kinds of variables: instance variables, class variables, and local variables.

Instance variables, define an object's attributes.

Class variables define the attributes of an entire class of objects and apply to all instances of it.

Local variables are used inside method definitions or even smaller blocks of statements within a method. You can use them only while the method or block is being executed by the Java Virtual Machine. They cease to exist afterward.

Although all three kinds of variables are created in much the same way, class and instance variables are used in a different manner than local variables. You learn about local variables today and explore instance and class variables to the next lectures "Working with Objects."

Creating Variables

Before you can use a variable in a Java program, you must create the variable by declaring its name and the type of information it will store. The type of information is listed first, followed by the name of the variable. The following all are examples of variable declarations:

```
int loanLength;
String message;
boolean gameOver;
```

In these examples, the `int` type represents integers, `String` is an object that holds text, and `boolean` is used for Boolean true/false values.

Local variables can be declared at any place inside a method, like any other Java statement, but they must be declared before they can be used.

In the following example, three variables are declared at the top of a program's `main()` method:

```
public static void main(String[] arguments) {
     int total;
     String reportTitle;
     boolean active;
}
```

If you are creating several variables of the same type, you can declare all of them in the same statement by separating the variable names with commas. The following statement creates three `String` variables named `street`, `city`, and `state`:

```
String street, city, state;
```

Variables can be assigned a value when they are created by using an equal sign (=) followed by the value. The following statements create new variables and give them initial values:

```
String zipCode = “02134”;

int box = 350;
boolean pbs = true;
String name = “Zoom”, city = “Boston”, state = “MA”;
```

As the last statement demonstrates, you can assign values to multiple variables of the same type by using commas to separate them.

You must give values to local variables before you use them in a program, or the program won't compile successfully. For this reason, it is good practice to give initial values to all local variables.

Instance and class variable definitions are given an initial value depending on the type of information they hold, as in the following:

- Numeric variables: `0`
- Characters: `"\0"`
- Booleans: `false`
- Objects: `null`

Naming Variables

Variable names in Java must start with a letter, an underscore character _, or a dollar sign `$`.

Variable names cannot start with a number. After the first character, variable names can include any combination of letters, numbers, underscore characters, or dollar signs.

Note

In addition, the Java language uses the Unicode character set, which includes thousands of character sets to represent international alphabets. Accented characters and other symbols can be used in variable names as long as they have a Unicode character number.

When naming a variable and using it in a program, it's important to remember that Java is case-sensitive—the capitalization of letters must be consistent. Because of this, a program can have a variable named `X` and another named `x` (so `Rose` is not `rose` is not `ROSE`).

In programs in this book and elsewhere, Java variables are given meaningful names that include several joined words. To make it easier to spot the words, the following general rules are used:

- The first letter of the variable name is lowercase.
- Each successive word in the variable name begins with a capital letter.
- All other letters are lowercase.

The following variable declarations follow these naming rules:

```
Button loadFile;
int localAreaCode;
boolean quitGame;
```

Although dollar signs and underscores are permitted in variable names, you should avoid using either of them except in one situation: When a variable's entire name is capitalized, each word is separated by an underscore. Here's an example:

```
static int DAYS_IN_WEEK = 7;
```

You will see why a variable name might be capitalized like this later today.

Dollar signs never should be used in variable names, even though they're permitted. The official documentation for Java always has discouraged their use, so programmers follow this convention.

Variable Types

In addition to a name, a variable declaration must include the data type of information being stored. The type can be any of the following:

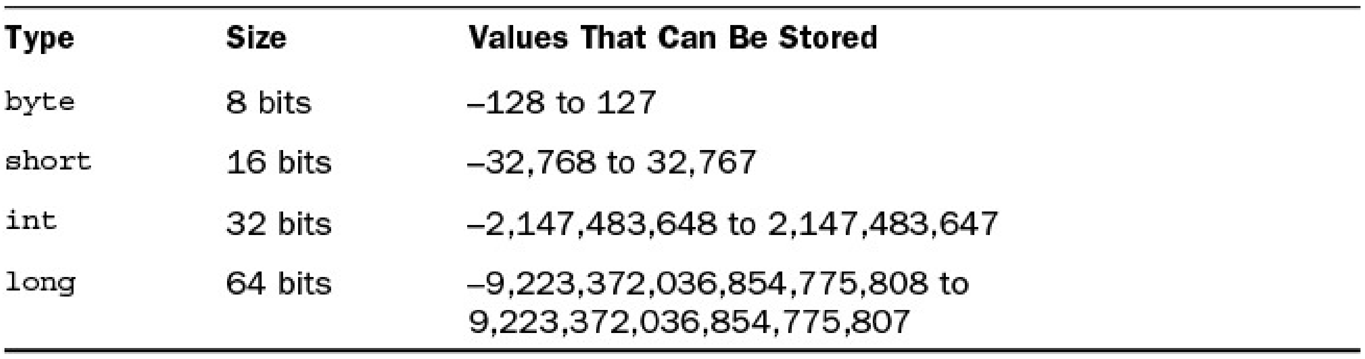
- One of the primitive data types, such as `int` or `boolean`
- The name of a class or interface
- An array

You learn how to declare and use array variables later . Today's lesson focuses on the other variable types.

Data Types

Java has eight basic data types that store integers, floating-point numbers, characters, and Boolean values. These often are called *primitive types* because they are built-in parts of the language rather than objects, which makes them easier to create and use. These data types have the same size and characteristics no matter what operating system and platform you're on, unlike some data types in other programming languages.

You can use four data types to store integers. Which one you use depends on the integer's size, as shown in Table 2.1.



Type	Size	Values That Can Be Stored
`byte`	8 bits	–128 to 127
`short`	16 bits	–32,768 to 32,767
`int`	32 bits	–2,147,483,648 to 2,147,483,647
`long`	64 bits	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

TABLE 2.1 Integer Types

All these types are *signed*, which means that they can hold either positive or negative numbers. The type used for a variable depends on the range of values it might need to hold. None of these integer variables can reliably store a value that is too large or too small for its designated variable type, so take care when designating the type.

Another type of number that can be stored is a floating-point number, which has the type `float` or `double`. *Floating-point* numbers are numbers with a decimal point. The `float` type can handle any number from `1.4E-45` to `3.4E+38`, while the `double` type can be used for more precise numbers ranging from `4.9E-324` to `1.7E+308`. Because `double` has more precision, that type generally is preferred.

The `char` type is used for individual characters, such as letters, numbers, punctuation, and other symbols.

The last of the eight primitive data types is `boolean`. As you have learned, this data type holds either `true` or `false`.

All these variable types appear in lowercase, and you must use them as such in programs. Some classes have the same names as these data types, but with different capitalization, such as `Boolean` and `Double` These are created and referenced differently in a Java program, so you can't use them interchangeably in most circumstances. Later you will see how to use these special classes.

Note

There's actually a ninth primitive data type in Java, `void`, which represents nothing. It's used in methods to indicate that they do not return a value.

Class Types

In addition to the primitive data types, a variable can have a class as its type, as in the following examples:

```
String lastName = “Hopper”;
Color hair;
MarsRobot robbie;
```

When a variable has a class as its type, the variable refers to an object of that class or one of its subclasses.

The last statement in the preceding list creates a variable named `robbie` that is reserved for a `MarsRobot` object. You learn more later about how to associate objects with variables.

Assigning Values to Variables

After a variable has been declared, a value can be assigned to it with the assignment operator, which is an equal sign =. The following are examples of assignment statements:

```
idCode = 8675309;

accountOverdrawn = false;
```

Constants

Variables are useful when you need to store information that can be changed as a program runs.

If the value never should change during a program’s runtime, you can use a type of variable called a constant. A *constant* is a variable with a value that never changes. (This might seem like an oxymoron, given the meaning of the word “variable.”)

Constants are useful in defining shared values for the use of all methods of an object. In Java, you can create constants for all kinds of variables: instance, class, and local.

To declare a constant, use the `final` keyword before the variable declaration and include an initial value for that variable, as in the following:

```
final double PI = 3.141592;
final boolean DEBUG = false;
final int PENALTY = 25;
```

Constants can be handy for naming various states of an object and then testing for those states. Suppose you have a program that takes directional input from the numeric keypad on the keyboard—press 8 to go up, 4 to go left, 6 to go right, and 2 to go down. You can define those values as constant integers:

```
final int LEFT = 4;
final int RIGHT = 6;
final int UP = 8;
final int DOWN = 2;
```

Constants often make a program easier to understand. To illustrate this point, consider which of the following two statements is more informative as to its function:

```
guide.direction = 4;
guide.direction = LEFT;
```

Note

In the preceding statements, the names of the constants such as `DEBUG` and `LEFT` are capitalized. This is a convention adopted by Java programmers to make it clear that the variable is a constant. Java does not require that constants be capitalized in this manner, but it’s a good practice to adopt.

When a constant’s variable name is more than one word, putting it in all caps would make the words run together confusingly, as in ESCAPECODE. Separate the words with an underscore character _, like this:

```
final int ESCAPE_CODE = 27;
```

Today’s first project is a Java application that creates several variables, assigns them initial values, and displays two of them as output. Run NetBeans and create a new Java program by undertaking these steps: : :

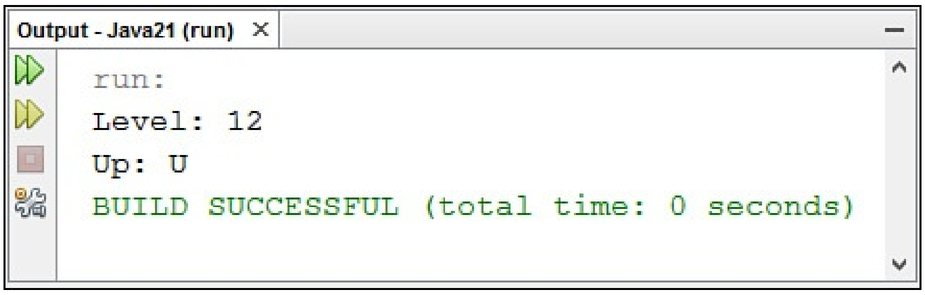
1. Choose the menu command File, New File. The New File dialog box opens.
2. In the Categories pane, choose Java.
3. In the File Types pane, choose Empty Java File and click Next. The Empty Java File dialog box opens.
4. In the Class Name text field, enter Variables, which will give the source code file the name Variables.java.
5. Here’s the different step: In the Package Name text field, enter com.java21days.
6. Click Finish.

On this project, you specify a class name and a package name. Packages are a way to organize related Java programs together. They serve a similar purpose to file folders in a file system. Enter the code shown in Listing 2.1 into the source code editor.

LISTING 2.1 The Full Text of `Variables.java`

```
package variables;

public class Variables {

    public static void main(String[] arguments) {
        final char UP = ‘U’;
        byte initialLevel = 12;
        short location = 13250;
        int score = 3500100;
        boolean newGame = true;

        System.out.println(“Level: “ + initialLevel);
        System.out.println(“Up: “ + UP);
    }

```

Save the file by choosing File, Save. NetBeans automatically compiles the application if it contains no errors. Run the program by choosing Run, Run File. This program produces the output shown in Figure 2.1.



```
run:
Level: 12
Up: U
BUILD SUCCESSFUL (total time: 0 seconds)
```

FIGURE 2.1 Creating and displaying variable values.

The package name of the class is established by the `package` statement, which must be the first line of a Java program when it is used:

```
package com.java21days;
```

This class uses four local variables and one constant, making use of `System.out.println()` in lines 12–13 to produce output.

`System.out.println()` is a method called to display strings and other information to the standard output device, which usually is the monitor.

This method takes a single argument within its parentheses: a string. To present more than one variable or literal as the argument to `println()`, the + operator combines the elements into a single string.

Java also has a `System.out.print()` method that displays a string without terminating it with a newline character. You can call `print()` instead of `println()` to display several strings on the same line.

Comments

One of the most effective ways to improve a program's readability is to use comments. These are text included in a program that explains what's going on in the code. The Java compiler ignores comments when preparing a bytecode version of a Java source file that can be run as a class, so there's no penalty for using them.

You can use three kinds of comments in Java programs.

A single-line comment is preceded by two slash characters `//`. Everything from the slashes to the end of the line is considered a comment and is disregarded by the compiler, as in the following statement:

```
int creditHours = 3; // set up credit hours for course
```

Everything from the slashes onward is ignored. As far as the compiler is concerned, the preceding line is the same as this:

```
int creditHours = 3;
```

A multiline comment begins with `/*` and ends with `*/`. Everything between these two delimiters is considered a comment, even over multiple lines, as in the following code:

```
/* This program occasionally deletes all files on
your hard drive and renders it unusable
forever when you click the Save button. */
```

A Javadoc comment begins with `/**` and ends with `*/`. Everything between these delimiters is considered to be official documentation on how the class and its methods work.

Javadoc comments are designed to be read by utilities such as javadoc, a command-line tool that's part of the Java Development Kit (JDK). This tool uses official comments to create a set of web pages that document the functionality of a Java class, show its place in relation to its superclass and subclasses, and describe each of its methods.

Tip

All the official documentation on each class in the Java Class Library is generated from Javadoc comments. You can view current Java documentation at http://docs.oracle.com/javase/8/docs/api.

Literals

In addition to variables, you can work with values as literals in a Java statement. A *literal* is any number, text, or other information that directly represents a value.

The following assignment statement uses a literal:

```
int year = 2016;
```

The literal 2016 represents the integer value 2016. Numbers, characters, and strings are all examples of literals. Java has types of literals that represent different kinds of numbers, characters, strings, and Boolean values.

Number Literals

Java has several integer literals. The number 4, for example, is an integer literal of the `int` variable type. It also can be assigned to `byte` and `short` variables, because the number is small enough to fit into those integer types. An integer literal larger than an `int` can hold automatically is considered to be of the type `long`. You also can indicate that a literal should be a `long` integer by adding the letter L to the number (either in upper- or lowercase). Here's an example:

```
pennyTotal = pennyTotal + 4L;
```

This statement adds the value 4, formatted as a `long`, to the current value of the `pennyTotal` variable.

To represent a negative number as a literal, precede it with a minus sign (–), as in –45.

Floating-point literals use a period character. for the decimal point, as you would expect. The following statement uses a literal to set up a `double` variable:

```
double gpa = 3.55;
```

All floating-point literals are considered to be of the `double` variable type instead of `float`. To specify a literal of `float`, add the letter F to the literal (upper- or lowercase), as in the following example:

```
float piValue = 3.1415927F;
```

You can use exponents in floating-point literals by using the letter e or E followed by the exponent, which can be a negative number. The following statements use exponential notation:

```
double x = 12e22;

double y = 19E-95;
```

A large integer literal can include an underscore character _ to make it more readable to humans. The underscore serves the same purpose as a comma in a large number, making its value more apparent. Consider these two examples, one of which uses underscores:

```
int jackpot = 3500000;

int jackpot = 3_500_000;
```

Both examples equal 3,500,000, which is easier to see in the second statement. The Java compiler ignores the underscores.

Java also supports numeric literals that use binary, octal, and hexadecimal numbering.

Binary numbers are a base-2 numbering system in which only the values 0 and 1 are used. Values made up of 1s and 0s are the simplest form for a computer and are a fundamental part of computing. Counting up from 0, binary values are 0, 1, 10, 11, 100, 111, and so on. Each digit in the number is called a bit. The combination of eight numbers is a byte. A binary literal is specified by preceding it with 0b, as in 0b101 for 101 (5 in decimal) and 0b01111111 (127).

Octal numbers are a base-8 numbering system, which means that they can represent only the values 0 through 7 as a single digit. The eighth number in octal is 10. Octal literals begin with a 0, so 010 is the decimal value 8, 012 is 9, and 020 is 16.

Hexadecimal is a base-16 numbering system that can represent 16 numbers as a single digit. The letters A through F represent the last six digits, so the first 16 numbers are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Hexademical literals begin with 0x, as on 0x12 (decimal 18) and 0xFF (255).

The octal and hexadecimal systems are better suited for certain tasks in programming than the normal decimal system. If you ever have edited a web page to set its background color, you could have used hexadecimal numbers for green (0x001100), blue (0x000011), or butterscotch (0xFFCC99).

Boolean Literals

The Boolean literals `true` and `false` are the only two values you can use when assigning a value to a `boolean` variable type or using a Boolean in a statement.

The following statement sets a `boolean` variable:

```
boolean chosen = true;
```

Caution

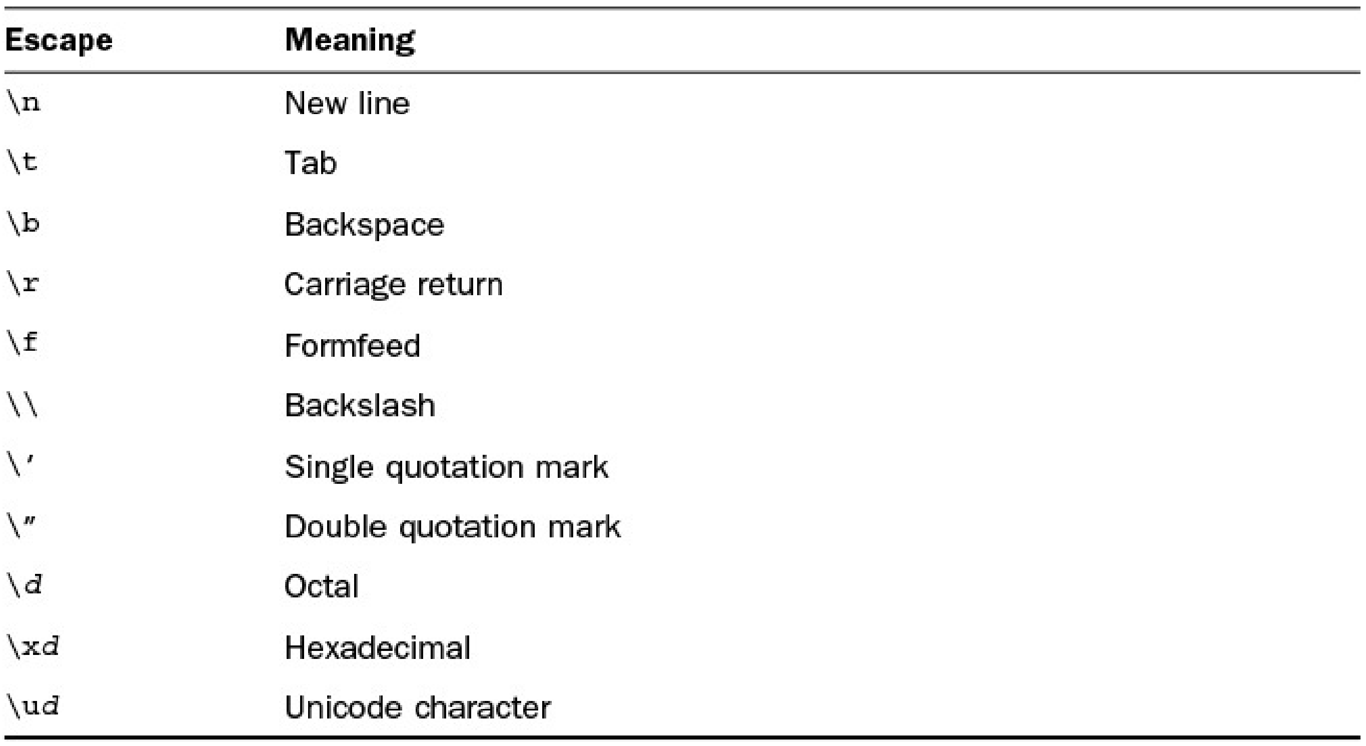
If you have programmed in other languages, you might expect that a value of 1 is equivalent to true and 0 is equivalent to false. This isn't the case in Java; you must use the values `true` and `false` to represent Boolean values.

Note that the literal `true` does not have quotation marks around it. If it did, the Java compiler would assume that it is a string of characters.

Character Literals

Character literals are expressed by a single character surrounded by single quotation marks, such as 'a', '#', and '3'. You might be familiar with the ASCII character set, which includes 128 characters, including letters, numerals, punctuation, and other characters useful in computing. Java supports ASCII along with thousands of additional characters through the 16-bit Unicode standard.

Some character literals represent characters that are not readily printable or accessible from a keyboard. Table 2.2 lists the codes that can represent these special characters as well as characters from the Unicode character set.



Escape	Meaning
`\n`	New line
`\t`	Tab
`\b`	Backspace
`\r`	Carriage return
`\f`	Formfeed
`\\`	Backslash
`\'`	Single quotation mark
`\"`	Double quotation mark
`\`*`d`*	Octal
`\x`*`d`*	Hexadecimal
`\u`*`d`*	Unicode character

TABLE 2.2 Character Escape Codes

In Table 2.2, the letter *d* in the octal, hex, and Unicode escape codes represents a number or a hexadecimal digit (a through f or A through F).

String Literals

The final literal that you can use in a Java program represents strings of characters. A string in Java is an object rather than a primitive data type. Strings are not stored in arrays as they are in languages such as C.

Because string objects are real objects in Java, methods are available to combine strings, modify strings, and determine whether two strings have the same value.

String literals consist of a series of characters inside double quotation marks, as in the following statements:

```
String quitMsg = “Are you sure you want to quit?”;

String password = “drowssap”;
```

Strings can include the character escape codes listed in Table 2.2, as shown here:

```
String example = “Socrates asked, "Hemlock is poison?"”;

System.out.println(“Sincerely,\nMillard Fillmore\n”);

String title = “Sams Teach Yourself Node in the John\u2122”;
```

In the last example, the Unicode code sequence `\u2122` produces a ™ symbol on systems that have been configured to support Unicode.

Caution

Although Java supports the transmission of Unicode characters, a computer also must support it for the characters to be displayed when the program is run. Unicode support provides a way to encode its characters for systems that support the standard. Java supports the display of any Unicode character that can be represented by a host font.

For more information about Unicode, visit the Unicode Consortium website at www.unicode.org.

Although string literals are used in a manner similar to other literals in a program, they are handled differently behind the scenes.

With a string literal, Java stores that value as a `String` object. You don’t have to explicitly create a new object, as you must when working with other objects, so they are as easy to work with as primitive data types. Strings are unusual in this respect—none of the basic types are stored as an object when used. You’ll learn more about strings and the `String` class later .

Expressions and Operators

An *expression* is a statement that can convey a value. Some of the most common expressions are mathematical, such as in the following examples:

```
int x = 3;
int y = x;
int z = x * y;
```

All three of these statements can be considered expressions; they convey values that can be assigned to variables. The first assigns the literal 3 to the variable `x`. The second assigns the value of the variable `x` to the variable `y`. In the third expression, the multiplication operator * is used to multiply the `x` and `y` integers, and the result is stored in the `z` integer.

Expressions can be any combination of variables, literals, and operators. They also can be method calls because methods send back a value to the object or class that called the method.

The value conveyed by an expression is called a *return value*. This value can be assigned to a variable and used in many other ways in your Java programs.

Most of the expressions in Java use operators such as *. *Operators* are special symbols used for mathematical functions, assignment statements, and logical comparisons.

Arithmetic

Five operators are used to accomplish basic arithmetic in Java, as shown in Table 2.3.

Operator	Meaning	Example
`+`	Addition	`3 + 4`
`-`	Subtraction	`5 - 7`
`*`	Multiplication	`5 * 5`
`/`	Division	`14 / 7`
`%`	Modulus	`20 % 7`

TABLE 2.3 Arithmetic Operators

Each operator takes two operands, one on each side of the operator. The subtraction operator also can be used to negate a single operand, which is equivalent to multiplying that operand by –1.

One thing to be mindful of when performing division is the type of numbers being used. If you store a division operation in an integer, the result is truncated to the next-lower whole number, because the `int` data type can't handle floating-point numbers.

For example, the expression `31 / 9` results in 3 if stored as an integer.

Modulus division, which uses the `%` operator, produces the remainder of a division operation. The expression `31 % 9` results in 4 because 31 divided by 9, with the whole number result of 3, leaves a remainder of 4.

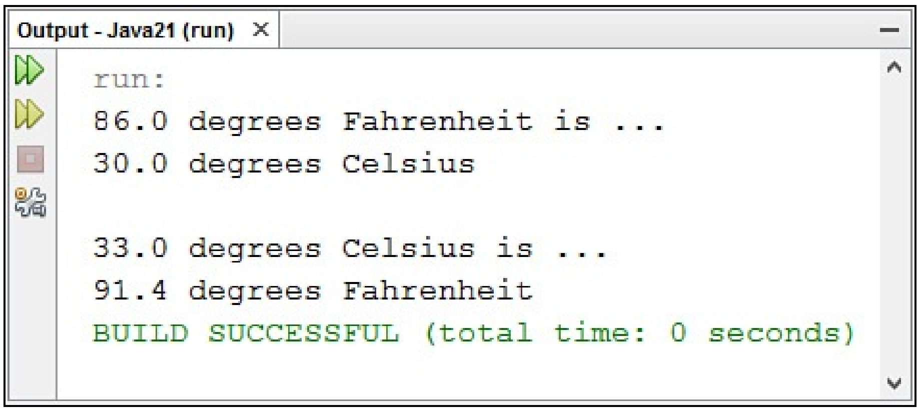
Note that many arithmetic operations involving integers produce an `int` regardless of the original type of the operands. If you're working with other numbers, such as floating-point numbers or `long` integers, you should make sure that the operands have the same type you're trying to end up with.

The next project is a Java class that demonstrates how to perform simple arithmetic in the language. Create a new empty Java file in NetBeans called `Weather` in the `com.java21days` package and enter the code shown in Listing 2.2 into the source code editor. Save the file with the menu command File, Save when you're done.

LISTING 2.2 The Full Text of `Weather.java`

```
package com.java21days;

public class Weather {
    public static void main(String[] arguments) {
        float fah = 86;
        System.out.println(fah + " degrees Fahrenheit is …");
        // To convert Fahrenheit into Celsius
        // begin by subtracting 32
        fah = fah - 32;
        // Divide the answer by 9
        fah = fah / 9;
        // Multiply that answer by 5
        fah = fah * 5;
        System.out.println(fah + " degrees Celsius\n");

        float cel = 33;
        System.out.println(cel + " degrees Celsius is …");
        // To convert Celsius into Fahrenheit
        // begin by multiplying by 9
        cel = cel * 9;
        // Divide the answer by 5
        cel = cel / 5;
        // Add 32 to the answer
        cel = cel + 32;
        System.out.println(cel + " degrees Fahrenheit");
    }
}
```

Run the program by selecting Run, Run File. It produces the output shown in Figure 2.2.



```
run:
86.0 degrees Fahrenheit is ...
30.0 degrees Celsius

33.0 degrees Celsius is ...
91.4 degrees Fahrenheit
BUILD SUCCESSFUL (total time: 0 seconds)
```

FIGURE 2.2 Converting temperatures with expressions.

In lines 5–14 of this Java application, a temperature in Fahrenheit is converted to Celsius using the arithmetic operators:

- **Line 5**—The floating-point variable `fah` is created with a value of 86.
- **Line 6**—The current value of `fah` is displayed.
- **Line 7**—The first of several comments explains what the program is doing. The Java compiler ignores these comments.
- **Line 9**—`fah` is set to its current value minus 32.
- **Line 11**—`fah` is set to its current value divided by 9.
- **Line 13**—`fah` is set to its current value multiplied by 5.
- **Line 14**—Now that `fah` has been converted to a Celsius value, `fah` is displayed again.

A similar thing happens in lines 16–25, but in the reverse direction. A temperature in Celsius is converted to Fahrenheit.

More About Assignment

Assigning a value to a variable is an expression because it produces a value. Because of this feature, you can combine assignment statements in this unusual way:

```
x = y = z = 7;
```

In this statement, all three variables `x`, `y`, and `z` end up with the value 7.

The right side of an assignment expression always is calculated before the assignment takes place. This makes it possible to use an expression statement as in the following code:

```
int x = 5;
x = x + 2;
```

In the expression `x = x + 2`, the first thing that happens is that `x + 2` is calculated. The result of this calculation, 7, is then assigned to `x`.

Using an expression to change a variable’s value is a common task in programming. Several operators are used strictly in these cases.

Table 2.4 shows these assignment operators and the expressions they are functionally equivalent to.

Expression	Meaning
`x += y`	x = x + y
`x -= y`	x = x – y
`x *= y`	x = x * y
`x /= y`	x = x / y

TABLE 2.4 Assignment Operators

Caution

These shorthand assignment operators are functionally equivalent to the longer assignment statements for which they substitute. If either side of your assignment statement is part of a complex expression, however, there are cases where the operators are not equivalent. For example, if `x` equals 20 and `y` equals 5, the following two statements do not produce the same value:

```
x = x / y + 5;
x /= y + 5;
```

The first statement produces an `x` value of 9 and the second an `x` value of 2. When in doubt about what an expression is doing, simplify it by using multiple assignment statements and don't use the shorthand operators.

Incrementing and Decrementing

Another common task required in programming is to add or subtract 1 from an integer variable. These expressions have special operators, which are called increment and decrement operators. *Incrementing* a variable means adding 1 to its value, and *decrementing* a variable means subtracting 1 from its value.

The increment operator is `++`, and the decrement operator is `--`. These operators are placed immediately after or before a variable name, as in the following code:

```
int x = 7;
x++;
```

In this example, the statement `x++` increments the `x` variable from 7 to 8.

These increment and decrement operators can be placed before or after a variable name. This affects the value of expressions that involve these operators.

Increment and decrement operators are called *prefix* operators if listed before a variable name and *postfix* operators if listed after a name.

In a simple expression such as `count--;`, using a prefix or postfix operator produces the same result, making the operators interchangeable. When increment and decrement operations are part of a larger expression, however, the choice between prefix and postfix operators is important.

Consider the following code:

```
int x, y, z;
x = 42;
y = x++;
z = ++x;
```

The three expressions in this code yield different results because of the difference between prefix and postfix operations.

When you use postfix operators on a variable in an expression, the variable's value is evaluated in the expression before it is incremented or decremented. So in `y = x++`, `y` receives the value of `x` before it is incremented by 1.

When using prefix operators on a variable in an expression, the variable is incremented or decremented before its value is evaluated in that expression. Therefore, in `z = ++x`, `x` is incremented by 1 before the value is assigned to `z`.

The end result of the preceding codes example is that `y` equals 42, `z` equals 44, and `x` equals 44.

If you’re still having some trouble figuring this out, here’s the example again with comments describing each step:

```
int x, y, z; // x, y, and z are declared
x = 42;      // x is given the value 42
y = x++;     // y is given x’s value (42) before it is incremented
             // and x is then incremented to 43
z = ++x;     // x is incremented to 44, and z is given x’s value
```

Caution

Using increment and decrement operators in complex expressions can produce results you might not expect.

The concept of “assigning `x` to `y` before `x` is incremented” isn’t precisely right, because Java evaluates everything on the right side of an expression before assigning its value to the left side.

Java stores some values before handling an expression to make postfix work the way it has been described in this section.

If you’re not getting the results you expect from a complex expression that includes prefix and postfix operators, try breaking the expression into multiple statements to simplify it.

Comparisons

Java has several operators for making comparisons among variables, variables and literals, or other types of information in a program.

These operators are used in expressions that return Boolean values of `true` or `false`, depending on whether the comparison being made is true or not. Table 2.5 shows the comparison operators.

Operator	Meaning	Example
`==`	Equal to	`x == 3`
`!=`	Not equal to	`x != 3`
`<`	Less than	`x < 3`
`>`	Greater than	`x > 3`
`<=`	Less than or equal to	`x <= 3`
`>=`	Greater than or equal to	`x >= 3`

TABLE 2.5 Comparison Operators

The following example shows a comparison operator in use:

```
boolean isHip;
int age = 37;
isHip = age < 25;
```

The expression `age < 25` produces a result of either `true` or `false`, depending on the value of the integer `age`. Because `age` is 37 in this example (which is not less than 25), `isHip` is given the Boolean value `false`.

Logical Operators

Expressions that result in Boolean values, such as comparison operations, can be combined to form more complex expressions. This is handled through logical operators, which are used for the logical combinations `AND`, `OR`, `XOR`, and logical `NOT`.

For `AND` combinations, the `&` or `&&` logical operator is used. When two Boolean expressions are linked by these operators, the combined expression returns a `true` value only if both Boolean expressions are true.

Consider this example:

```
boolean extraLife = (score > 75000) & (playerLives < 10);
```

This expression combines two comparison expressions: `score > 75000` and `playerLives < 10`. If both expressions are true, the Boolean value `true` is assigned to the variable `extraLife`. In any other circumstance, the value `false` is assigned to the variable.

The difference between `&` and `&&` lies in how much work Java does on the combined expression. If `&` is used, the expressions on both sides of the `&` are evaluated no matter what. If `&&` is used and the left side of the `&&` is false, the expression on the right side of the `&&` never is evaluated.

This makes `&&` more efficient because no unnecessary work is performed. In the preceding example if `score` is not greater than 75,000, there's no need to consider whether `playerLives` is less than 10.

For `OR` combinations, the `|` or `||` logical operator is used. These combined expressions return a `true` value if either Boolean expression is true.

Consider this example:

```
    boolean extralife = (score > 75000) || (playerLevel == 0);
```

This expression combines two comparison expressions: `score > 75000` and `playerLevel == 0`. If either of these expressions is true, the Boolean value `true` is assigned to the variable `extraLife`. Only if both of these expressions are false is the value `false` assigned to `extraLife`.

Note the use of `||` instead of `|`. Because of this usage, if `score > 75000` is true, `extraLife` is set to `true`, and the second expression never is evaluated.

The `XOR` combination has one logical operator, `^`. This results in a `true` value only if the Boolean expressions it combines have opposite values. If both are true or both are false, the `^` operator produces a `false` value.

The `NOT` combination uses the `!` logical operator followed by a single expression. It reverses the value of a Boolean expression in the same way that a minus sign reverses the positive or negative sign on a number. For example, if `age < 25` returns a `true` value, `!(age < 25)` returns a `false` value.

The logical operators may seem illogical when you first encounter them. You get plenty of opportunities to work with them during the nextlectures.

Operator Precedence

When more than one operator is used in an expression, Java has an established precedence hierarchy to determine the order in which operators are evaluated. In many cases, this precedence determines the expression’s overall value.

For example, consider the following expression:

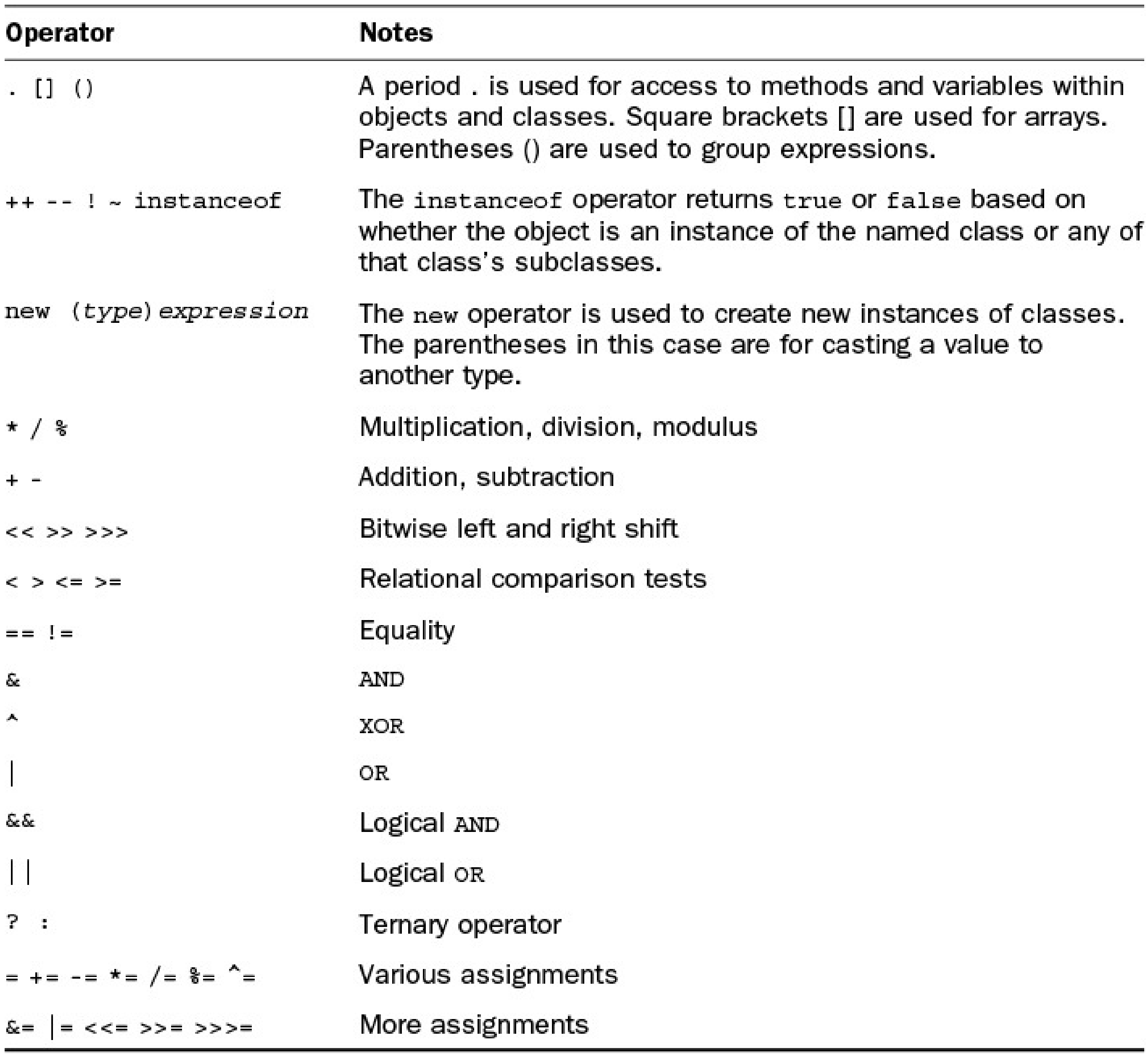
```
    y = 6 + 4 / 2;
```

The `y` variable will equal the value 5 or the value 8, depending on which arithmetic operation is handled first. If the `6 + 4` expression comes first, `y` has the value of 5. Otherwise, `y` equals 8.

In general, the order of evaluation from first to last is as follows:

1. Increment and decrement operations
2. Arithmetic operations
3. Comparisons
4. Logical operations
5. Assignment expressions

If two operations have the same precedence, the one on the left in the expression is handled before the one on the right. Table 2.6 shows the specific precedence of the various operators in Java. Operators higher up in the table are evaluated first.



Operator	Notes
`. [] ()`	A period . is used for access to methods and variables within objects and classes. Square brackets [] are used for arrays. Parentheses () are used to group expressions.
`++ -- ! ~ instanceof`	The `instanceof` operator returns `true` or `false` based on whether the object is an instance of the named class or any of that class's subclasses.
`new (`*`type`*`)`*`expression`*	The `new` operator is used to create new instances of classes. The parentheses in this case are for casting a value to another type.
`* / %`	Multiplication, division, modulus
`+ -`	Addition, subtraction
`<< >> >>>`	Bitwise left and right shift
`< > <= >=`	Relational comparison tests
`== !=`	Equality
`&`	AND
`^`	XOR
`|`	OR
`&&`	Logical AND
`||`	Logical OR
`? :`	Ternary operator
`= += -= *= /= %= ^=`	Various assignments
`&= |= <<= >>= >>>=`	More assignments

TABLE 2.6 Operator Precedence

Several of the operators listed in Table 2.6 are covered later.

Returning to the expression `y = 6 + 4 / 2`, Table 2.6 shows that division is evaluated before addition, so the value of `y` equals 8.

To change the order in which expressions are evaluated, place parentheses around the expressions that should be evaluated first. You can nest one set of parentheses inside another to make sure that expressions are evaluated in the desired order; the innermost parenthetic expression is evaluated first.

The following expression results in a value of 5:

```
y = (6 + 4) / 2
```

The value of 5 is the result because `6 + 4` is calculated first, and then the result, 10, is divided by 2.

Parentheses also can improve an expression’s readability. If an expression’s precedence isn’t immediately clear to you, adding parentheses to impose the desired precedence can make the statement easier to understand.

String Arithmetic

As stated earlier, the + operator has a double life outside the world of mathematics. It can concatenate two or more strings.

The word “concatenate” means to link two things. For reasons unknown, it is the verb of choice in computer programming when describing the act of combining two strings, winning out over paste, glue, affix, combine, link, smush together, and conjoin.

In several examples, you have seen statements that look something like this:

```
String brand = “Jif”;
System.out.println(“Choosy mothers choose “ + brand);
```

These two lines result in the display of the following text:

```
Choosy mothers choose Jif
```

The + operator combines strings, other objects, and variables to form a single string. In the preceding example, the literal “Choosy mothers choose” is concatenated to the value of the `String` object `brand`.

Working with the concatenation operator is made easier in Java by the fact that the operator can handle any variable type and object value as if it were a string. If any part of a concatenation operation is a `String` or a string literal, all elements of the operation are treated as if they were strings:

```
System.out.println(4 + ” score and “ + 7 + ” years ago”);
```

This produces the output text “4 score and 7 years ago”, as if the integer literals 4 and 7 were strings.

There also is a += shorthand operator to append something to the end of a string. For example, consider the following expression:

```
myName += ” Jr.”;
```

This expression is equivalent to the following:

```
myName = myName + ” Jr.”;
```

In this example, += changes the value of `myName`, which might be something like “Robert Downey,” by adding “Jr.” at the end to form the string “Robert Downey Jr.”

To summarize today’s material, Table 2.7 lists the operators you have learned about. Be a doll and look them over carefully.

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to
&&	Logical AND
||	Logical OR
!	Logical NOT
&	AND
|	OR
^	XOR
=	Assignment
++	Increment
--	Decrement
+=	Add and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign

TABLE 2.7 Operator Summary

Summary

Anyone who pops open a set of matryoshka dolls has to be a bit disappointed upon reaching the smallest doll in the group.

Today you reached Java's smallest nesting doll. Using statements and expressions enables you to begin building effective methods, which makes effective objects and classes possible.

Today you learned about creating variables and assigning values to them. You also used literals to represent numeric, character, and string values and worked with operators. Later , you'll put these skills to use developing classes.

Q&A

Q What happens if I assign an integer value to a variable that is too large for that variable to hold?

A Logically, you might think that the variable is converted to the next-larger type, but this isn't what happens. Instead, an *overflow* occurs—a situation in which the number wraps around from one size extreme to the other. An example of overflow would be a `byte` variable that goes from 127 (an acceptable value) to 128 (unacceptable). It would wrap around to the lowest acceptable value, which is –128, and start counting upward from there. Overflow isn't something you can readily detect in a program, so be sure to give your numeric variables plenty of living space in their chosen data type.

Small data types like `byte` were more necessary when computers had much less memory than they do today and every byte counted. Today, with plentiful memory and hard disk space measured in terabytes, it is better to use larger data types like `int` to ensure that you have enough space to store all possible values in a particular variable.

Q Why does Java have all these shorthand operators for arithmetic and assignment? It's really hard to read that way.

A Java's syntax is based on C++, which is based on C (more Russian nesting doll behavior). C is an expert language that values programming power over readability, and the shorthand operators are one of the legacies of that design priority. Using them in a program isn't required because effective substitutes are available, so you can avoid them in your own programming if you prefer.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. Which of the following is a valid value for a `boolean` variable?

   A. “false”

   B. `false`

   C. 10

2. Which of these is NOT a convention for naming variables in Java?

   A. After the first word in the variable name, each successive word begins with a capital letter.

   B. The first letter of the variable name is lowercase.

   C. All letters are capitalized.

3. Which of these data types holds numbers from –32,768 to 32,767?

   A. `char`

   B. `byte`

   C. `short`

Quiz

Review today’s material by taking this three-question quiz.

Questions

1. Which of the following is a valid value for a boolean variable?
   - A. “false”
   - B. false
   - C. 10
2. Which of these is NOT a convention for naming variables in Java?
   - A. After the first word in the variable name, each successive word begins with a capital letter.
   - B. The first letter of the variable name is lowercase.
   - C. All letters are capitalized.
3. Which of these data types holds numbers from –32,768 to 32,767?
   - A. char
   - B. byte
   - C. short

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create a program that calculates how much a $14,000 investment would be worth if it increased in value by 40% during the first year, lost $1,500 in value the second year, and increased 12% in the third year.
2. Write a program that displays two numbers and uses the / and % operators to display the result and remainder after they are divided. Use the \t character escape code to make a tab character separate the result and remainder in your output.