Java Basics

Contents

Introduction
  Applications and Applets
  Objects and Classes
  A First Java Application Program

Elements of Java
  Identifiers
  Reserved Words
  Variables
  Primitive Types
  Constants
  Assignment Statements
  Assignment Compatibilities
  Type Casting
  Arithmetic Operators and Expressions
  Parentheses and Precedence Rules
  Increment and Decrement Operators
  Special Assignment Operators
  Named Constants
  The Class Math

Simple Input and Output Using the Keyboard and Screen
  Screen Output
  Keyboard Input Using the Class Scanner

The if-else Statement
  Boolean Expressions
  Nested Statements
  Multiway if-else Statements
  The Conditional Operator (Optional)

The switch Statement

Enumerations

Introduction

Applications and Applets

There are two kinds of Java programs, applications and applets. An application is simply a program that runs on your computer like any other program. It is a stand-alone program. In contrast, an applet is a program that cannot run without the support of a browser or a viewer. Typically, an applet is sent to another location on the Internet and is run there. The term “applet” is meant to suggest a little application.

Applets and applications are almost identical. Once you know how to design and write one, it is easy to learn to write the other. This book uses applications rather than applets.

Objects and Classes

An object is a program construct that contains data and can perform certain actions. When a Java program is run, the objects interact with one another to accomplish a particular task. The actions performed by objects are defined by methods in the program. When you ask an object to perform
an action, you **invoke**, or **call**, a method. Java has two kinds of methods. A **valued method** uses a **return** statement to return a result, but a **void method** does not.

All objects of the same kind are said to be in the same class. So a **class** is a category or kind or type of object. All objects in the same class have the same types of data and the same methods.

You will see some objects and methods in the next section and again when we discuss the classes Math, Scanner, and String later in this appendix. Appendix C reviews classes, objects, and methods in more detail. If you are not familiar with these concepts, you should read at least Segment C.1 in Appendix C now and the rest of Appendix C later.

### A First Java Application Program

**B.3** To give you a feel for the Java language, let’s take a brief, informal look at the following sample Java application program:

```java
import java.util.Scanner;
public class FirstProgram {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Hello out there.");
        System.out.println("Want to talk some more?");
        System.out.println("Answer yes or no.");
        String answer = keyboard.next();
        if (answer.equals("yes"))
            System.out.println("Nice weather we are having.");
        System.out.println("Good-bye.");
    }  // end main
}  // end FirstProgram
```

The program is stored in the file FirstProgram.java.

**Figure B-1** shows two screen displays that might be produced when a user runs and interacts with this program. The text typed by the user is shown in bold.

---

### FIGURE B-1 Two possible results when running the sample program

<table>
<thead>
<tr>
<th>Hello out there.</th>
<th>Hello out there.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Want to talk some more?</td>
<td>Want to talk some more?</td>
</tr>
<tr>
<td>Answer yes or no.</td>
<td>Answer yes or no.</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Nice weather we are having.</td>
<td>Good-bye.</td>
</tr>
<tr>
<td>Good-bye.</td>
<td></td>
</tr>
</tbody>
</table>

---

**B.4** This program uses the class Scanner, which is a part of the **Java Class Library**. This library contains many standard classes that you can use in your Java programs. The classes in the library are organized into groups called **packages**. The import statement indicates that this program uses the class Scanner from the package java.util.

The program contains the definition of a class that begins with the lines

```java
public class FirstProgram {
```

and ends with

```java
}  // end FirstProgram
```
Within the class definition is a method called `main` that begins with the statements

```java
public static void main(String[] args)
{

```

and ends with

```java
} // end main
```

Statements within this pair of braces are the body of the method `main`. Generally, a class contains several methods, each defining a specific task and each having a name of our choosing. An application program, however, must contain a method that is called `main`.

### B.5

The line

```java
Scanner keyboard = new Scanner(System.in);
```

gets us ready to read data from the keyboard using Java’s class `Scanner`. It defines the `Scanner` object `keyboard` and associates it with the object `System.in`. This latter object represents the computer’s keyboard. `System.in` is in the class `System`, which is in the package `java.lang` of the Java Class Library.

**Programming Tip:** No `import` statement is necessary when you use a class from the package `java.lang` of the Java Class Library.

### B.6

The next three lines display text for the user of the program:

```java
System.out.println("Hello out there.");
System.out.println("Want to talk some more?");
System.out.println("Answer yes or no.");
```

Each of these lines causes the quoted characters, or `string`, given within the parentheses to be displayed on the screen. `System.out` is an object within the class `System` that can send output to the screen via its method `println`. You invoke an object’s method by writing the object name followed by a period, or `dot`, followed by the method name and some parentheses that may or may not have something inside them. The text inside the parentheses is called an `argument` and provides the information the method needs to carry out its action. In each of these three lines, the method `println` writes the value of its argument—here, the characters inside the quotes—to the screen.

The method `println` is an example of a void method. It performs an action, but does not return a value.

### B.7

The next line of the program reads the characters that are typed at the keyboard and stores them in the variable `answer` as a string:

```java
String answer = keyboard.next();
```

You use the `Scanner` object `keyboard` to invoke its method `next`. In this way, you read the word that the user types at the keyboard. The user presses the Enter key (also called the Return key) after typing the word. Although the method `next` has no arguments, the parentheses are required. You will learn more about the class `Scanner` later in this appendix.

The method `next` is an example of a valued method. It returns the value read. Its `invocation`, `keyboard.next()`, represents this value.

### B.8

The next two lines of the program make a decision to do or not do something based on the value of the variable `answer`. The first line asks whether the string stored in `answer` is the string `yes`. If it is, a message is displayed on the screen. Otherwise, the message is not displayed.

Notice that the first sample dialogue in Figure B-1 displays the string *Nice weather we are having*, and the second one does not. That is because, in the first run of the program, the string `yes` is stored in the variable `answer`, and in the second run of the program, the string `no` is stored in `answer`. 

Of course, precise rules govern how you write each part of a Java program. For example, a final
semicolon ends each Java statement. These rules form the grammar for the Java language, just
as there are rules for the grammar of the English language. The grammar rules for a programming
language (or any language) are called the syntax of the language. We now look at the elements of
Java in more detail.

Elements of Java

In this section, we examine how to use Java to perform arithmetic computations.

Identifiers

You use identifiers to name certain parts of a program. An identifier in Java consists entirely of
letters, digits, the underscore character _, and the dollar sign $. An identifier cannot start with a
digit and must not contain a space or any other special character such as a period or an asterisk.
There is no official limit to the length of a name, though in practice, there is always a limit.
Although Java allows identifiers to contain a dollar sign, it is reserved for special purposes, and so
you should not use $ in a Java identifier.

Java is case sensitive. This means that it treats uppercase letters and lowercase letters as
different characters. For example, mystuff, myStuff, and MyStuff are three different identifiers.
Having these identifiers in the same program could be confusing to human readers, and therefore
doing so is a poor programming practice. But the Java compiler would be happy with them.

Java uses a character set, called Unicode, that includes characters from languages other than
English. Java allows you to use these extra characters in identifiers, but you are not likely to find
them on most keyboards. Segment B.68 discusses Unicode further.

Although it is not required by the Java language, the common practice, and the one followed in
this book, is to start the names of classes with uppercase letters and to start the names of objects,
methods, and variables (which you are about to see) with lowercase letters. These names are usu-
ally spelled using only letters and digits. We separate multiword names by using uppercase letters,
since we cannot use spaces. For example, the following are all legal identifiers that follow this
well-established convention:

```
inputStream   YourClass   CarWash   hotCar   theTimeOfDay
```

Some people use an underscore to separate the words in an identifier, but typically we will not.
The following are all illegal identifiers in Java, and the compiler will complain if you use any
of them:

```
.MyClass   goTeam-   7eleven
```

The first two contain an illegal character, either a dot or a dash. The last name is illegal because it
starts with a digit.

Reserved Words

Some words, such as the word if, have a special predefined meaning in the Java language. You
cannot use these words, called reserved words or keywords, for anything other than their intended
meaning. A full list of reserved words for Java is given on the inside cover of this book. Within a
programming environment, the names of reserved words are often highlighted in some way. In this
book, they will appear in boldface.

Some other words, such as String, name classes that are supplied with Java. They have a
predefined meaning but are not reserved words. This means that you can change their meaning,
but doing so could easily confuse you or somebody else reading your program.
Variables

B.13 A variable in a program represents a memory location that stores data such as numbers and letters. The number, letter, or other data item in a variable is called its value. This value can be changed, so that at one time the variable contains, say, 6, and at another time after the program has run for a while, the variable contains a different value, such as 4.

You use an identifier to name a variable. Besides following the rules and conventions for identifiers, you should choose variable names that suggest their use or the kind of data they will hold. For example, if a variable is used to count something, you might name the variable count. If the variable is used to hold the speed of an automobile, you might call the variable speed. You should almost never use single-letter variable names like x and y. Also by convention, variable names begin with a lowercase letter.

B.14 A variable’s data type—or simply type—determines what kind of value the variable can hold. If the type is int, the variable can hold integers. If the type is double, the variable can hold numbers with a decimal point and a fractional part after the decimal point. If the type is char, the variable can hold any one character from the computer keyboard.

Java has two kinds of types, primitive types and reference types. A primitive type represents simple, indecomposable values, such as a single number or a single letter. The types int, double, and char are primitive types. A reference type represents a reference, or a memory address, instead of the actual item stored at that address. A class type—which is a type for objects of a class—is a reference type. For example, String is a class type. A variable of a reference type is called a reference variable. A second kind of reference type is an array type. We will discuss arrays in Segment B.86.

**Note:** Naming conventions help you to distinguish among identifiers when reading a program. By convention, variables begin with a lowercase letter, as do the names of primitive types. The names of class types begin with an uppercase letter.

B.15 A variable declaration indicates the type of data the variable will hold. Different types of data are stored in the computer’s memory in different ways. Variable declarations are necessary so that the value of a variable can be correctly stored in or retrieved from the computer’s memory. Even though the mechanisms for storing values in the variables of class types differ from the mechanisms used for primitive types, you declare variables for class types and primitive types in the same way.

You declare a variable by writing a type name followed by a list of variable names separated by commas and ending with a semicolon. All the variables named in the list will have the type given at the start of the declaration. For example:

```java
int numberOfBaskets, eggsPerBasket, totalEggs;
String myName;
```

The first line declares that the three variables numberOfBaskets, eggsPerBasket, and totalEggs will contain values of type int. The second line declares that myName will store a String object.

You must declare a variable in a Java program before you use it. Normally, you declare a variable either just before it is used or at the start of a method definition.

Primitive Types

B.16 A whole number without a decimal point, such as 0, 1, or −2, is called an integer. A number with a decimal point, such as 3.14159, −8.63, or 5.0, is called a floating-point number. Notice that 5.0 is a floating-point number, not an integer. If a number has a fractional part, even if the fractional part is zero, it is a floating-point number.
All the Java primitive types appear inside the cover of this book. Notice that there are four types for integers—namely byte, short, int, and long. The only difference among the various integer types is the range of integers they can store and the amount of computer memory they use. If you cannot decide which integer type to use, use the type int. It has a large enough range for most purposes and does not use as much memory as the type long.

Java has two types for floating-point numbers, float and double. If you cannot decide between the types float and double, use double. It allows a wider range of values and is used as a default type for floating-point numbers.

You use the primitive type char for single characters, such as letters, digits, or punctuation. For example, the following declares the variable symbol to be of type char, stores the character for uppercase A in symbol, and then displays that value—the A—on the screen:

```
char symbol;
symbol = 'A';
System.out.println(symbol);
```

Notice that we enclose the character A in single quotes. Again note that uppercase letters and lowercase letters are different characters. For example, 'a' and 'A' represent two different characters.

Finally, the primitive type boolean has two values, true and false. You can use a variable of type boolean to store the answer to a true/false question such as “Is myTime less than yourTime?”

### Constants

**B.17** A variable can have its value changed; its value varies. A literal number like 2 cannot change. It is always 2. It is never 3. Values like 2 or 4.8 are called constants, or literals, because their values do not change.

You write constants of integer types with an optional plus sign or minus sign, but without commas or decimal points. Floating-point constants have an optional plus sign or a minus sign and no commas. You can write a floating-point constant in one of two ways. One way looks like the everyday way of writing numbers. For example, 9.8, -3.14, and 5.0 are floating-point constants, because they contain a decimal point. The second way is to include a multiplier that is a power of 10. You use the letter e to represent both the multiplication sign and the 10. For example, the number 8.65 \times 10^8 appears in Java as 8.65e8 (or in the less convenient form 86500000.0). The two forms, 8.65e8 and 86500000.0, are equivalent in a Java program. Similarly, the number 4.83 \times 10^{-4}, which is equal to 0.000483, can be written as 4.83e-4 in Java.

The e stands for “exponent,” since it is followed by a number that is thought of as an exponent of 10. The number before the e can be a number with or without a decimal point. The number after the e cannot contain a decimal point.

Other types of literal expressions are also called constants. You write constants of type char by placing the character within single quotes. For example, 'Y' is a constant of type char. A string constant is a sequence of characters enclosed in double quotes, as in "Java".

### Assignment Statements

**B.18** You can use an assignment statement to give a value to a variable. For example, if answer is a variable of type int and we want to give it the value 42, we could use the following assignment statement:

```
answer = 42;
```

An assignment statement always consists of a single variable on the left-hand side of an equal sign and an expression on the right-hand side followed by a semicolon. The expression can be another variable, a constant, or a more complicated expression made up by combining operators, such as + and *, with variables and constants. The value of the expression is assigned to the variable on the left of the equal sign.
For example, the following are all examples of assignment statements:

```java
amount = 3.99;
firstInitial = 'B';
score = numberOfCards + handicap;
```

Here we assume that `amount` is a variable of type `double`, `firstInitial` is of type `char`, and the rest of the variables are of type `int`. If the variable `numberOfCards` has the value 7 and `handicap` has the value 2, the value of the variable `score` is 9.

The equal sign, `=`, which is called the assignment operator, does not mean equality. You can think of the assignment operator as saying, “Make the value of the variable equal to what follows.” For example, in the statement

```java
eggsPerBasket = eggsPerBasket - 2;
```

the variable `eggsPerBasket` occurs on both sides of the assignment operator. This statement subtracts 2 from the present value of `eggsPerBasket` and assigns the new value to `eggsPerBasket`. In effect, the statement decreases the value of `eggsPerBasket` by 2.

**B.19** A variable that has been declared but that has not yet been given a value by the program is uninitialized. Such a variable might literally have no value, or it might have some default value. For example, an integer variable might have a default value of zero, and a reference variable might have a default value of `null`, which is a predefined constant in Java. However, your program will be clearer if you explicitly give the variable a value, even if you are simply reassigning it the default value. (The exact details on default values have been known to change and should not be counted on.)

One easy way to ensure that you do not have an uninitialized variable is to initialize it within the declaration. Simply combine the declaration and an assignment statement, as in the following examples:

```java
int count = 0;
double taxRate = 0.075;
char grade = 'A';
int balance = 1000, newBalance;
```

Note that a single declaration, such as the last statement, can initialize some variables and not others.

Sometimes the compiler may complain that you have failed to initialize a variable. In most cases, this is indeed true. Occasionally, the compiler is mistaken about this. However, the compiler will not compile your program until you convince it that the variable in question is initialized. To make the compiler happy, initialize the variable when it is declared, even if the variable will be given a different value before you use it for anything. In such cases, you cannot argue with the compiler.

**Note:** A reference variable that contains `null` does not reference any object.

**Assignment Compatibilities**

**B.20** You cannot put a square peg in a round hole, and similarly you cannot put a value of one type in a variable of another type. You cannot put an `int` value like 42 in a variable of type `char`. You cannot put a `double` value like 3.5 in a variable of type `int`. You cannot even put the `double` value 3.0 in a variable of type `int`. You cannot store a value of one type in a variable of another type unless the value is somehow converted to match the type of the variable.
When dealing with numbers, however, this conversion will sometimes—but not always—be performed for you automatically. For example, you can always assign a value of an integer type to a variable of a floating-point type, such as when you write either

```java
double interestRate = 7;
```
or

```java
int wholeRate = 7;
double interestRate = wholeRate;
```

More generally, you can assign a value of any type on the following list to a variable of any type that appears further down on the list:

- `byte` → `short` → `int` → `long` → `float` → `double`

For example, you can assign a value of type `long` to a variable whose type is either `long`, `float`, or `double`. Notice that you can assign a value of any integer type to a variable of any floating-point type. This is not an arbitrary ordering of the types. As you move down the list from left to right, the types become more complex, or wider, either because they allow larger values or because they allow decimal points in the numbers. Thus, you can assign a value of one type to a variable of either the same type or a wider type.

In addition, you can assign a value of type `char` to a variable of type `int` or to any of the numeric types that follow `int` in the previous list of types. However, we do not advise doing so, because the result could be confusing.¹

If you want to assign a value of type `double` to a variable of type `int`, you must change the type of the value explicitly by using a type cast, as we explain next.

### Type Casting

A **type cast** is the changing of the type of a value to some other type, such as changing the type of 2.0 from `double` to `int`. The previous segment described when a change in type is done for you automatically. In all other cases, if you want to assign a value of one type to a variable of another type, you must perform a type cast. For example, you cannot simply assign a value of type `double` to a variable of type `int`, even if the value of type `double` happens to have all zeros after the decimal point and so is conceptually a whole number. Thus, the second of the following statements is illegal:

```java
double distance = 9.0;
int points = distance; // ILLEGAL
```

To cast the type of `distance` to `int`, you enclose `int` within parentheses and place it in front of `distance`. For example, we would replace the preceding illegal assignment with

```java
int points = (int)distance; // Casting from double to int
```

Note that when you type-cast from any floating-point type to any integer type, the value is not rounded. The part after the decimal point is simply discarded, or **truncated**. For example, if the variable `distance` contains 25.86, `(int)distance` has an `int` value of 25. A type cast does not really change the value of a variable; `distance` is still 25.86, but `points` is 25.

Recall that when you assign an integer value to a variable of a floating-point type, the type cast is done automatically for you.

---

¹ Readers who have used certain other languages, such as C or C++, may be surprised to learn that we cannot assign a value of type `char` to a variable of type `byte`. This is because Java uses the Unicode character set rather than the ASCII character set, and so Java reserves two bytes of memory for each value of type `char`, but naturally reserves only one byte of memory for values of type `byte`. This is one of the few cases where we might notice that Java uses the Unicode character set. Indeed, if we convert from an `int` to a `char` or vice versa, we can expect to get the usual correspondence of ASCII numbers and characters.
Note: When casting, some programmers place a space before the variable, as in (int) sum. We prefer to treat casting much like a minus sign. Just as we write minus five as –5, we cast sum to an integer by writing (int)sum.

Arithmetic Operators and Expressions

B.22 In Java, you perform arithmetic by using the arithmetic operators +, −, *, /, and %. You combine variables and constants with these operators and parentheses to form an arithmetic expression. The variables and constants in an expression are called operands. Spaces around the operators, operands, and parentheses within an expression are ignored.

A unary operator is one that has only one operand, like the operator – in the assignment statement

```java
bankBalance = -cost;
```

A binary operator has two operands, like the operators + and * in

```java
total = cost + (tax * discount);
```

Note that the operators − and + can be used as both unary and binary operators.

The meaning of an expression is basically what you expect it to be, but there are some subtleties about the type of the result and occasionally even about the value of the result. The type of the value produced when an expression is evaluated depends on the types of the values being combined. Consider an expression with only two operands, such as

```java
amount – adjustment
```

If both amount and adjustment are of type int, the result of the subtraction has type int. If either amount or adjustment, or both, is of type double, the result is of type double. If we replace the operator − with any of the operators +, *, /, or %, the type of the result is determined in the same way. However, the operator % is typically used with integers, as you will see soon.

Larger expressions using more than two operands are viewed as a series of steps, each of which involves only two operands. For example, to evaluate the expression

```java
balance + (balance * rate)
```

we evaluate balance * rate and obtain a number, and then we add that number to balance. Thus, if balance is int and rate is double, balance * rate is double and so is the entire expression.

Note: If all the items in an arithmetic expression have the same type, the result has that type. If at least one of the items has a floating-point type, the result has a floating-point type.

Knowing whether the value produced has an integer type or a floating-point type is typically all that you need. However, if you need to know the exact type of the value produced by an arithmetic expression, you can use the following rule:

Note: The data type of an arithmetic expression’s value matches the most complex, or widest, data type among the operands in the expression. In other words, the data type matches the type that appears rightmost in the following list:

```
byte → short → int → long → float → double
```

For example, if sum is float and more is int, sum + more is float.
B.23 The division operator / deserves special attention, because the type of its operands can affect the value produced in a dramatic way. When you combine two numbers with the division operator / and at least one of the numbers has a floating-point type, the result has a floating-point type. For example, 9.0 / 2 has one operand of type double, namely 9.0. Hence, the result is the type double number 4.5. However, when both operands have an integer type, the result can be surprising. For example 9 / 2 has two operands of type int, and so it yields the result 4 of type int, not 4.5. The fraction after the decimal point is simply lost. When you divide two integers, the result is truncated, not rounded. The part after the decimal point is discarded no matter how large it is. So, 11 / 3 is 3, not 3.6666... If nothing but a zero is after the decimal point, that decimal point and zero are still lost. Even this seemingly trivial difference can be of some significance. For example, 8.0 / 2 has the value 4.0 of type double, which technically is only an approximate quantity. However, 8 / 2 has the int value 4, which is an exact quantity. The approximate nature of 4.0 can affect the accuracy of any further calculation that is performed with this result.

Often, the % operator has operands only of integer types. You use it to recover the equivalent of the fraction after the decimal point. When you divide one integer by another, you get a result (which some call a quotient) and a remainder. For example, 14 divided by 4 yields 3 with a remainder of 2 (or with 2 left over). The % operation gives the remainder—that is, the amount left over after doing the division. So 14 / 4 is 3 and 14 % 4 is 2, because 14 divided by 4 is 3 with 2 left over. The % operator is called the remainder operator.

The % operator has more applications than you might at first suspect. It allows your program to count by 2s, 3s, or any other number. For example, if you want to do something to every other integer, you need to know whether the integer is even or odd. An integer n is even if n % 2 is zero, and it is odd if n % 2 is not zero. Similarly, if you want your program to do something to every third integer, you test whether the integer n is divisible by 3. It will be if n % 3 is zero.

![Programming Tip:](image)

To make your arithmetic expressions more readable, place a space on both sides of each binary operator.

Parentheses and Precedence Rules

B.24 You can use parentheses to group portions of an arithmetic expression in the same way that you use parentheses in algebra and arithmetic. With the aid of parentheses, you can indicate which operations are performed first, second, and so forth. For example, consider the following two expressions that differ only in the positioning of their parentheses:

\[
\begin{align*}
\text{(cost + tax)} & \times \text{discount} \\
\text{cost} & + (\text{tax} \times \text{discount})
\end{align*}
\]

To evaluate the first expression, the application first adds cost and tax and then multiplies the result by discount. To evaluate the second expression, it multiplies tax by discount and then adds the result to cost. If you use some numbers for the values of the variables and carry out the two evaluations, you will see that they produce different results.

If you omit parentheses, as in the assignment statement

\[
\text{total} = \text{cost} + \text{tax} \times \text{discount};
\]

multiplication occurs before addition. Thus, the previous statement is equivalent to

\[
\text{total} = \text{cost} + (\text{tax} \times \text{discount});
\]
More generally, when the order of operations is not determined by parentheses, the operations occur in an order determined by the following precedence rule:

**Note: Precedence of arithmetic operators**

Arithmetic operators in an expression execute in the order of their precedence, as follows:

- The unary operators +, –
- The binary operators *, /, %
- The binary operators +, –

Parentheses can override this order.

Operators that are listed higher on the list are said to have higher precedence. For example, unary operators have a higher precedence than binary operators, and binary operators have a lower precedence than unary operators. Operators of higher precedence execute before operators of lower precedence, unless parentheses override this order. Operators at the same level have the same precedence. When two operators have equal precedence, the operations are performed using the following convention:

**Note: Binary operators of equal precedence in an expression are performed in left-to-right order.**

Increment and Decrement Operators

**B.25** The increment and decrement operators increase or decrease the value of a variable by 1. The increment operator is written as two plus signs, ++. For example, the following Java statement will increase the value of the variable count by 1:

```java
count++;  
```

If the variable count has the value 5 before this statement is executed, it will have the value 6 after the statement is executed. Thus, this statement is equivalent to

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

You can use the increment operator with variables of any numeric type, but it is used most often with variables of an integer type such as int.

The decrement operator is similar, except that it subtracts 1 rather than adds 1 to the value of the variable. The decrement operator is written as two minus signs, --. For example, the following will decrease the value of the variable count by 1:

```java
count--;  
```

If the variable count has the value 5 before this statement is executed, it will have the value 4 after the statement is executed. This statement is equivalent to

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

**B.26** You can use the increment and decrement operators within expressions, but when you do, the increment operator or the decrement operator changes the value of the variable it is applied to and returns a value. Although we do not recommend using the increment and decrement operators in expressions, you might see them used this way in code written by other developers.
In expressions, you can place the ++ or -- either before or after a variable, but your choice affects the result. For example, consider the code

```java
int n = 3;
int m = 4;
int result = n * (++m);
```

After this code executes, the value of \( n \) is unchanged at 3, the value of \( m \) is 5, and the value of \( \text{result} \) is 15. Thus, \( \text{++m} \) changes the value of \( m \) and returns that changed value to the multiply operator.

In the previous example, we placed the increment operator before the variable \( m \). If we place it after the variable \( m \), something slightly different happens. Consider the code

```java
int n = 3;
int m = 4;
int result = n * (m++);
```

Now, after the code executes, the value of \( n \) is 3 and the value of \( m \) is 5, just as in the previous example. However, the value of \( \text{result} \) is 12, not 15, because \( m++ \) returns the value of \( m \) to the multiply operator and then changes the value of \( m \).

The two expressions \( n * (\text{++m}) \) and \( n * (\text{m++}) \) both increase the value of \( m \) by 1, but the first expression increases the value of \( m \) before it does the multiplication, whereas the second expression increases the value of \( m \) after it does the multiplication. Both \( \text{++m} \) and \( \text{m++} \) have the same effect on the final value of \( m \), but when we use them as part of an arithmetic expression, they give a different value to the expression.

Similarly, both \( --m \) and \( m-- \) have the same effect on the final value of \( m \), but when we use them as part of an arithmetic expression, they give a different value to the expression. If the \( -- \) is before the \( m \), the value of \( m \) is decreased before its value is used in the expression. If the \( -- \) is after the \( m \), the value of \( m \) is decreased after its value is used in the expression.

The increment and decrement operators can be applied only to variables. They cannot be applied to constants or to entire, more complicated arithmetic expressions.

**Programming Tip:** To avoid errors and confusing code, use the operator ++ or -- only in a statement that involves one variable and no other operators.

### Special Assignment Operators

**B.27** You can combine the simple assignment operator (=) with an arithmetic operator, such as +, to produce a kind of special-purpose assignment operator. For example, the following will increase the value of the variable \( \text{amount} \) by 5:

```java
amount += 5;
```

This is really just a shorthand for

```java
amount = amount + 5;
```

You can do the same thing with any of the other arithmetic operators -, *, /, and %. For example, the statement

```java
amount *= 25
```

is equivalent to

```java
amount = amount * 25;
```
Named Constants

You probably recognize the number 3.14159 as the approximate value of \( \pi \), the number that is used in many circle calculations and that is often written as \( \pi \). However, when you see 3.14159, you might not be sure that it is \( \pi \) and not some other number; somebody other than you might have no idea of where the number 3.14159 came from. To avoid such confusion, you should always give a name to constants, such as 3.14159, and use the name instead of writing out the number. For example, we might give the number 3.14159 the name \( \text{PI} \). Then the assignment statement

\[
\text{area} = 3.14159 \times \text{radius} \times \text{radius};
\]

could be written more clearly as

\[
\text{area} = \text{PI} \times \text{radius} \times \text{radius};
\]

How do you give a number, or other constant, a name like \( \text{PI} \)? You could use a variable named \( \text{PI} \) and initialize it to the desired value 3.14159. But then you might inadvertently change the value of this variable. However, Java provides a mechanism that allows you to define and initialize a variable and moreover fix the variable’s value so it cannot be changed. The syntax is

\[
\text{public static final type name} = \text{constant};
\]

For example, the statement

\[
\text{public static final double PI} = 3.14159;
\]
gives the name \( \text{PI} \) to the constant 3.14159. The part

\[
\text{double PI} = 3.14159;
\]

simply declares \( \text{PI} \) as a variable and initializes it to 3.14159. The word \text{public} says that there are no restrictions on where we can use the name \( \text{PI} \). The word \text{static} defines one copy of \( \text{PI} \) that every object of the class can access instead of having its own copy of \( \text{PI} \). The word \text{final} means that the value 3.14159 is the \text{final} value assigned to \( \text{PI} \) or, to phrase it another way, it means that the program cannot change the value of \( \text{PI} \). Appendix C provides more details about \text{static} and \text{final}.

It is a good practice to place named constants near the beginning of a class and outside of any method definitions. That way, your named constants are handy in case you need to modify them. You might, for example, want to change the number of digits you provide for a constant.

**Note:** The class \text{Math} in the Java Class Library defines a static constant named \( \text{PI} \) just like the one we defined in this segment, but with more decimal places. The following segment describes this class and shows how to access \( \text{PI} \). You should use \text{Math}'s \( \text{PI} \) instead of defining your own.

**Programming Tip:** Programmers typically use all uppercase letters when naming constants to distinguish constants from ordinary variables. They use an underscore as a separator in multi-word names. For example, \text{FEET_PER_MILE} follows this convention.
The Class Math

B.29 The class Math in the package java.lang of the Java Class Library provides a number of standard mathematical methods. These methods are static methods. (Segment C.28 of Appendix C discusses static methods in more detail.) When you invoke a static method, you write the class name—Math, in this case—a dot, the name of the method, and a pair of parentheses. Using the name of a class to invoke a method is not typical. Ordinarily, you use the name of an object to invoke a method.

Most Math methods require that you specify items within the pair of parentheses. As we noted earlier in this appendix, these items are called arguments to the method. Thus, a typical invocation of a method in this class has the form Math.method_name(arguments).

You can invoke the method in an assignment statement, such as

```
variable = Math.method_name(arguments);
```

or embed it within an arithmetic expression. That is, you can use Math.method_name(arguments) anyplace that you can use a variable of a primitive data type. Figure B-2 describes some of the available methods in this class.

The class Math also has two predefined named constants. \( \mathbb{E} \) is the base of the natural logarithm system—often written \( e \) in mathematical formulas—and is approximately 2.72. \( \pi \) is used in calculations involving circular geometric figures—often written \( \pi \) in mathematical formulas—and is approximately 3.14159. Because these constants are defined in the class Math, you use them by writing Math.E and Math.PI.

Simple Input and Output Using the Keyboard and Screen

The input and output of data is usually referred to as I/O. A Java program can perform I/O in many different ways. In this section, we present some ways to handle simple text input that we type at the keyboard and simple text output displayed on the screen.

Screen Output

B.30 Statements like

```
System.out.println("Enter a whole number from 1 to 99.");
```

and

```
System.out.println(quarters + " quarters");
```

send output to the display screen. As we mentioned earlier in this appendix, System.out is an object within the class System, which is a class in the Java Class Library. This object has println as one of its methods. So the preceding output statements are calls to the method println of the object System.out. You simply follow System.out.println with a pair of parentheses that contain what you want to display. You end the statement with a semicolon.

Within the parentheses can be strings of text in double quotes, like "Enter a whole number from 1 to 99.", variables like quarters, numbers like 5 or 7.3, and almost any other object or value. To display more than one thing, simply place a + operator between them. For example,

```
System.out.println("Lucky number = " + 13 +
"Secret number = " + number);
```
If the value of number is 7, the output will be

Lucky number = 13
Secret number = 7

Notice also that no spaces are added. If we want a space between the 13 and the word “Secret” in the preceding output—and we probably do—we should add a space to the beginning of the string "Secret number = " so that it becomes " Secret number = ".

Notice that you use double quotes, not single quotes, and that the opening and closing quotes are the same symbol. Finally, notice that you can place the statement on two lines if it is too long.

FIGURE B-2 Some methods in the class Math

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.cbrt(x)</td>
<td>Returns the cube root of x.</td>
</tr>
<tr>
<td>Math.ceil(x)</td>
<td>Returns the nearest whole number that is ( \geq x ).</td>
</tr>
<tr>
<td>Math.cos(x)</td>
<td>Returns the trigonometric cosine of the angle ( x ) in radians.</td>
</tr>
<tr>
<td>Math.exp(x)</td>
<td>Returns ( e^x ).</td>
</tr>
<tr>
<td>Math.floor(x)</td>
<td>Returns the nearest whole number that is ( \leq x ).</td>
</tr>
<tr>
<td>Math.hypot(x, y)</td>
<td>Returns the square root of the sum ( x^2 + y^2 ).</td>
</tr>
<tr>
<td>Math.log(x)</td>
<td>Returns the natural (base e) logarithm of x.</td>
</tr>
<tr>
<td>Math.log10(x)</td>
<td>Returns the base 10 logarithm of x.</td>
</tr>
<tr>
<td>Math.pow(x, y)</td>
<td>Returns ( x^y ).</td>
</tr>
<tr>
<td>Math.random()</td>
<td>Returns a random number that is ( \geq 0 ) but (&lt; 1 ).</td>
</tr>
<tr>
<td>Math.sin(x)</td>
<td>Returns the trigonometric sine of the angle ( x ) in radians.</td>
</tr>
<tr>
<td>Math.sqrt(x)</td>
<td>Returns the square root of x, assuming that ( x \geq 0 ).</td>
</tr>
<tr>
<td>Math.tan(x)</td>
<td>Returns the trigonometric tangent of the angle ( x ) in radians.</td>
</tr>
<tr>
<td>Math.toDegrees(x)</td>
<td>Returns an angle in degrees equivalent to the angle ( x ) in radians.</td>
</tr>
<tr>
<td>Math.toRadians(x)</td>
<td>Returns an angle in radians equivalent to the angle ( x ) in degrees.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.abs(x)</td>
<td>Returns the absolute value of x.</td>
</tr>
<tr>
<td>Math.max(x, y)</td>
<td>Returns the larger of x and y.</td>
</tr>
<tr>
<td>Math.min(x, y)</td>
<td>Returns the smaller of x and y.</td>
</tr>
<tr>
<td>Math.round(x)</td>
<td>Returns the nearest whole number to ( x ). If ( x ) is float, returns an int; if ( x ) is double, returns a long.</td>
</tr>
</tbody>
</table>
However, you should break the line before or after a + operator, not in the middle of a quoted string or a variable name. You also should indent the second line to make the entire statement easier to read.

Later, in the section about the class String, you will see that the + operator joins, or concatenates, two strings. In the preceding System.out.println statement, Java converts the number 13 to the string "13". Likewise, it converts the integer 7 in the variable number to the string "7". Then the + operator joins the strings and the System.out.println statement displays the result. You do need to be a bit careful, however. If you write a + between two numeric values or variables, they will be added rather than concatenated.

You can also use the println method to display the value of a String variable, as illustrated by the following:

```java
String greeting = "Hello Programmers!";
System.out.println(greeting);
```

This will cause the following to be written on the screen.

```
Hello Programmers!
```

**B.31** Every invocation of println ends a line of output. If you want the output from two or more output statements to appear on a single line, use print instead of println. For example,

```java
System.out.print("One, two,");
System.out.print(" buckle my shoe.");
System.out.println(" Three, four,");
System.out.println("shut the door.");
```

will produce the following output:

```
One, two, buckle my shoe. Three, four, shut the door.
```

Notice that a new line is not started until you use println, rather than print. Also notice that the new line starts after displaying the items specified in the println statement. This is the only difference between print and println.

**Keyboard Input Using the Class Scanner**

**B.32** A Java program can read data from either the keyboard or another source such as a disk, and place it into memory. The Java Class Library provides the class Scanner for this purpose. Here, we will use the methods in Scanner to read data typed at the keyboard and place it into variables that we specify.

As we noted earlier, the class Scanner is in the package java.util. To use Scanner in your program, you must import it from this package by writing the following import statement before the rest of your program:

```java
import java.util.Scanner;
```

Before you can use any of the methods in Scanner, you must create a Scanner object by writing a statement such as

```java
Scanner keyboard = new Scanner(System.in);
```

The variable keyboard—which could be any variable of your choosing—is assigned a Scanner object that is associated with the input device that System.in represents. This device by convention is the keyboard. The variable keyboard has a class type.
B.33 Scanner provides several methods that read input data. You can use any of these methods by writing a statement that has the following form, where keyboard is the Scanner object that we defined previously:

```java
variable = keyboard.method_name();
```

The named method reads a value from the keyboard and returns it. That is, the expression `keyboard.method_name()` represents the value that was read. The previous statement then assigns this value to the indicated variable.

You can read integers and real numbers by using the following expressions:

- `keyboard.nextInt()` — Returns the next integer encountered in the input data.
- `keyboard.nextDouble()` — Returns the next real number encountered in the input data.

Each of these expressions ignores any white space that might precede or follow the number typed at the keyboard. White-space characters are the characters that appear as spaces when printed on paper or displayed on the screen. The blank-space character is likely the only white-space character that will concern us at first, but the start of a new line and the tab symbol are also white-space characters.

B.34 Example. To read an integer from the keyboard, you can write a statement such as

```java
size = keyboard.nextInt();
```

where `size` has been declared previously as an `int` variable. The user of your program would type an integer and press the Enter, or Return, key. The value is read by the method `nextInt`, returned, and assigned to the variable `size`.

Typically, you should display a message, or prompt, for the user to enter data. For example, your program might contain the following statements:

```java
System.out.println("What is your age?");
int age = keyboard.nextInt();
```

Whatever the user types appears in the same window as the prompt. Here, the prompt would appear on one line and the user would type his or her age on the next.

B.35 Example. To read a real number from the keyboard, you can write statements such as

```java
System.out.print("Enter the area of your room in square feet: ");
double area = keyboard.nextDouble();
```

After the user types a real number at the keyboard and presses the Enter key, the value of the number is assigned to the variable `area`. Since we have used `print` instead of `println`, both the prompt and the input data appear on the same line on the display.

B.36 Example. You can read more than one value per line of input. For example,

```java
System.out.println("Please enter your height in feet and inches:");
int feet = keyboard.nextInt();
int inches = keyboard.nextInt();
```

The user could type either

```
6 2
```
on one line or

```
6
2
```
on two lines. In either case, `feet` is 6 and `inches` is 2.
Note: Streams
The characters that a user types at the keyboard are directed into the memory assigned to your program by an object known as an input stream. The name of the input stream associated with the keyboard is System.in. Likewise, an output stream is an object that directs data from your program to an output device. System.out is such an object, directing characters to a display.

B.37 More input methods. The class Scanner includes the following method to read a string:

nextLine()—Returns the string that appears next in the input data.

For example, if keyboard is defined as shown earlier, the statement

    String message = keyboard.nextLine();

reads the entire string that the user types before pressing the Enter key—including any spaces—and assigns it to the variable message.

The method next in the class Scanner reads the next group of contiguous characters that are not white space and returns it as a string. For example, you can use this method to read the next word that appears in the input data, as follows:

    String word = keyboard.next();

We used next in Segment B.3 to read the user’s yes or no response.

The Scanner methods we’ve looked at—nextInt, nextDouble, nextLine, and next—often are invoked in simple assignment statements, although that is not necessary. Since each method returns a value, you can call it within an arithmetic expression, for example.

The if-else Statement
B.38 In programs, as in everyday life, things can sometimes go in one of two different ways. If you have money in your checking account, some banks will pay you a little interest. On the other hand, if you have overdrawn your checking account, you will be charged a penalty. This might be reflected in the bank’s accounting program by the following Java statement, known as an if-else statement:

    if (balance >= 0)
        balance = balance + (INTEREST_RATE * balance) / 12;
    else
        balance = balance - OVERDRAWN_PENALTY;

The two-symbol operator >= means “greater than or equal to” in Java. We use two symbols because the one-character symbol ≥ is not on the keyboard.

The meaning of an if-else statement is really just the meaning it would have if read as an English sentence. When your program executes an if-else statement, it first checks the expression in parentheses after the if. This expression must evaluate to either true or false. If it is true, the statement after the if is executed. If the expression is false, the statement after the else is executed. In the preceding example, if balance is positive or zero, the following action occurs:

    balance = balance + (INTEREST_RATE * balance) / 12;
We divide by 12 because the interest is for only 1 of 12 months.) On the other hand, if the value of balance is negative, the following is executed instead:

\[
\text{balance} = \text{balance} - \text{OVERDRAWN\_PENALTY};
\]

The indentation in the if-else statement is conventional as an aid in reading the statement; it does not affect the statement’s meaning.

B.39 If you want to include more than one statement in either of the two portions of the if-else statement, you simply enclose the statements in braces, as in the following example:

```java
if (balance >= 0)
{
    System.out.println("Good for you. You earned interest.");
    balance = balance + (INTEREST\_RATE * balance) / 12;
}
else
{
    System.out.println("You will be charged a penalty.");
    balance = balance - OVERDRAWN\_PENALTY;
} // end if
```

When you enclose several statements within braces, you get one larger statement called a compound statement. Compound statements are seldom used by themselves but often are used as substatements of a larger statement such as an if-else statement.

**Programming Tip:** Some programmers always use compound statements within other statements such as if-else, even when only a single statement appears between the braces. Doing so makes it easier to add another statement to the compound statement, but more importantly, it avoids the error that would occur if you forgot to add the braces. We encourage you to follow this convention, even though we do not always do so in this book to save space.

B.40 You can omit the else part. If you do, nothing happens when the tested expression is false. For example, if your bank does not charge an overdraft penalty, the statement would be the following, instead of the previous one:

```java
if (balance >= 0)
{
    System.out.println("Good for you. You earned interest.");
    balance = balance + (INTEREST\_RATE * balance) / 12;
} // end if
```

If balance is negative, the statement after the closing brace executes next.

### Boolean Expressions

B.41 A **boolean expression** is an expression that is either true or false. The expression

\[
\text{balance} \geq 0
\]

that we used in the previous if-else statement is an example of a simple boolean expression. Such expressions compare two things, like numbers, variables, or other expressions. Figure B-3 shows the various Java **comparison operators** you can use to compare two expressions.
Logical operators. Often, when you write an if-else statement, you will want to use a boolean expression that is more complicated than a simple comparison. You can form more-complicated boolean expressions from simpler ones by joining expressions with either the Java version of "and," which is &&, or the Java version of "or," which is ||. For example, consider the following:

```java
if ((pressure > min) && (pressure < max))
    System.out.println("Pressure is OK.");
else
    System.out.println("Warning: Pressure is out of range.");
```

If the value of pressure is greater than min, and the value of pressure is less than max, the output will be

- **Pressure is OK.**

Otherwise, the output is

- **Warning: Pressure is out of range.**

Note that you cannot use a string of inequalities in Java, like the following:

```java
min < pressure < max
```

This is illegal! Instead, you must express each inequality separately and connect them with &&, as follows:

```java
(pressure > min) && (pressure < max)
```

The parentheses in the previous expression are not necessary, but we typically include them. The parentheses that surround the entire expression in an if-else statement are required, however.

The binary operators && and || together with the unary operator ! are logical operators. We look at each of them next.

The operator &&. When you form a larger boolean expression by connecting two smaller expressions with &&, the entire larger expression is true provided that both of the smaller expressions are true. Thus, if at least one of pressure > min and pressure < max is false, the larger expression is false. Moreover, if the first part of the larger expression is false, the second part is ignored, since the larger expression must be false regardless of the value of the second part. For example, if pressure is less than min, we know that

```java
(pressure > min) && (pressure < max)
```

is false without looking at pressure < max.
The operator ||. You also can use || to form a larger boolean expression from smaller ones in the same way that you use &&, but with different results. The meaning is essentially the same as the English word “or.” For example, consider

```java
if (((salary > expenses) || (salary + savings > expenses))
    System.out.println("Solvent");
else
    System.out.println("Bankrupt");
```

If the value of salary is greater than the value of expenses or the value of salary + savings is greater than the value of expenses—or both—the output will be Solvent. Otherwise, the output will be Bankrupt.

The entire larger expression is true if either of the smaller expressions is true. Moreover, if the first part of the larger expression is true, the second part is ignored, since the larger expression must be true regardless of the value of the second part. For example, if salary is greater than expenses, we know that

```java
(salary > expenses) || (salary + savings > expenses)
```

is true without looking at salary + savings > expenses.

You use parentheses in expressions containing the || operator in the same way that you use them with &&.

---

**Note:** Short-circuit evaluation

When two boolean expressions are joined by either && or ||, the second expression is not evaluated if the value of the first expression implies the value of the entire expression. Such is the case if the first expression is false when the operator is && or true when the operator is ||. This behavior is known as the short-circuit evaluation of a boolean expression.

Besides saving execution time, short-circuit evaluation can prevent execution errors. For example, the following statement prevents a division by zero:

```java
if ((count != 0) && (sum / count > minimum))
```

If count is zero, the expression count != 0 is false. Thus, the expression sum / count > minimum is not evaluated, thereby avoiding the erroneous division.

The operator !. You can negate a boolean expression by preceding it with the operator !. For example, the expression

```java
!(number >= min)
```

has the same meaning as the expression

```java
number < min
```

In this case, you can and should avoid using !.

Sometimes, however, the use of ! makes perfect sense. For example, if you have two strings that should be the same for normal processing to continue, you would compare them and issue a warning if they are not equal. Later, in the section about the class String, you will see that you use the equals method to compare two strings. For example,

```java
stringOne.equals(stringTwo)
```
is true if the strings stringOne and stringTwo are equal. But if we want to know whether these strings are not equal, we could write

```java
if (!stringOne.equals(stringTwo))
    System.out.println("Warning: The strings are not the same.");
```

The precedence of the boolean operators in relation to each other and to the arithmetic operators follows:

---

**Note: Precedence of a selection of Java operators**

Operators in the same expression execute in the order of their precedence, as follows:

- The unary operators `+`, `-`, `!`
- The binary arithmetic operators `*`, `/`, `%`
- The binary arithmetic operators `+`, `-`
- The comparison operators `<`, `>`, `<=`, `>=`
- The comparison operators `==`, `!=`
- The logical operator `&&`
- The logical operator `||`

Parentheses can override this order.

---

**Nested Statements**

B.46 Notice that an if-else statement contains smaller statements within it. These smaller statements can be any sort of Java statements. In particular, you can use one if-else statement within another if-else statement to get nested if-else statements, as illustrated by the following:

```java
if (balance >= 0)
    if (INTEREST_RATE >= 0)
        balance = balance + (INTEREST_RATE * balance) / 12;
    else
        System.out.println("Cannot have a negative interest.");
else
    balance = balance - OVERDRAWN_PENALTY;
```

If the value of balance is greater than or equal to zero, the entire following if-else statement is executed:

```java
if (INTEREST_RATE >= 0)
    balance = balance + (INTEREST_RATE * balance) / 12;
else
    System.out.println("Cannot have a negative interest.");
```

When writing nested if-else statements, you may sometimes become confused about which if goes with which else. To eliminate this confusion, you can add braces as follows:

```java
if (balance >= 0)
{
    if (INTEREST_RATE >= 0)
        balance = balance + (INTEREST_RATE * balance) / 12;
    else
        System.out.println("Cannot have a negative interest.");
}
else
    balance = balance - OVERDRAWN_PENALTY;
```

Here, the braces are an aid to clarity but are not, strictly speaking, needed. In other cases, they are needed. While you should use indentation to indicate your intentions, remember that it is ignored by the compiler.
B.47 If you omit an else, things get a bit trickier. The following two if-else statements differ only in that one has a pair of braces, but they do not have the same meaning:

```java
// First Version
if (balance >= 0)
{
    if (INTEREST_RATE >= 0)
        balance = balance + (INTEREST_RATE * balance) / 12;
} else
    balance = balance - OVERDRAWN PENALTY;
// Second Version
if (balance >= 0)
    if (INTEREST_RATE >= 0)
        balance = balance + (INTEREST_RATE * balance) / 12;
else
    balance = balance - OVERDRAWN PENALTY;
```

In the second version, without braces, the else is paired with the second if, not the first one, as the indentation leads us to believe. Thus, the meaning is

```java
// Equivalent to Second Version
if (balance >= 0)
{
    if (INTEREST_RATE >= 0)
        balance = balance + (INTEREST_RATE * balance) / 12;
    else
        balance = balance - OVERDRAWN PENALTY;
}
```

To clarify the difference a bit more, consider what happens when balance is less than zero. The first version causes the following action:

```java
balance = balance - OVERDRAWN PENALTY;
```

However, the second version takes no action.

**Note:** In an if-else statement, each else is paired with the nearest previous unmatched if.

**Programming Tip:** Indentation within an if-else statement does not affect the action of the statement. For clarity, you should use indentation that matches the logic of the statement.

### Multiway if-else Statements

B.48 Since an if-else statement has two outcomes, and each of these two outcomes can have an if-else statement with two outcomes, you can use nested if-else statements to produce any number of possible effects. Convention provides a standard way of doing this. Let’s start with an example.

Suppose balance is a variable that holds your checking account balance and you want to know whether your balance is positive, negative (overdrawn), or zero. To avoid any questions
about accuracy, let’s assume that balance is of type int—that is, balance is the number of dollars in your account, with the cents ignored. To find out if your balance is positive, negative, or zero, you could use the following nested if-else statement:

```java
if (balance > 0)
    System.out.println("Positive balance");
else if (balance < 0)
    System.out.println("Negative balance");
else if (balance == 0)
    System.out.println("Zero balance");
```

This is really an ordinary nested if-else statement, but it is indented differently than before. The indentation reflects the logic more clearly and is preferred. Although this is not a separate kind of if-else statement, we call this nested construction a **multiway if-else statement**.

When a multiway if-else statement is executed, the application tests the boolean expressions one after the other, starting from the top. When it finds the first true boolean expression, it executes the statement after the expression. The rest of the if-else statement is ignored. For example, if balance is greater than zero, the preceding statements will display

**Positive balance**

Exactly one of the three possible messages will be displayed, depending on the value of the variable balance.

**B.49** The previous example has three possibilities, but you can have any number of possibilities by adding more else-if parts. In this example, the possibilities are mutually exclusive. That is, only one of the three possibilities can actually occur for any given value of balance. However, you can use any boolean expressions, even if they are not mutually exclusive. If more than one boolean expression is true, only the action associated with the first true boolean expression is executed. A multiway if-else statement never performs more than one action.

If none of the boolean expressions is true, nothing happens. However, it is a good practice to add an else clause—without any if—at the end, to be executed in case none of the boolean expressions is true. In fact, we can rewrite our previous example in this way. We know that if balance is neither positive nor negative, it must be zero. So we do not need the test

```java
if (balance == 0)
```

Thus, we can and should write the previous if-else statement as

```java
if (balance > 0)
    System.out.println("Positive balance");
else if (balance < 0)
    System.out.println("Negative balance");
else
    System.out.println("Zero balance");
```

The Conditional Operator *(Optional)*

**B.50** To allow compatibility with older programming styles, Java includes an operator that is a notational variant on certain forms of the if-else statement. A **conditional operator expression** consists of a boolean expression followed by a question mark and two expressions separated by a colon. For example, the expression on the right side of the assignment operator in the following statement is a conditional operator expression:

```java
max = (n1 > n2) ? n1 : n2;
```

The ? and : together form a **ternary operator** that has three operands and is known as the **conditional operator**. If the boolean expression is true, the value of the first of the two expressions is returned; otherwise, the value of the second of the two expression is returned. Thus, the logic of this example is equivalent to
This book will not use conditional operator expressions, as they are less clear than equivalent if-else statements. If you decide to use them in your program, realize that not everyone will know their meaning.

### The switch Statement

**B.51** Multiway if-else statements can become unwieldy when you must choose from among many possible courses of action. If the choice is based on the value of an integer or character expression, the switch statement can make your code easier to read.

The switch statement begins with the word switch followed by an expression in parentheses. This expression is called the **controlling expression**. Its value must be of type int, char, byte, short, or String. The switch statement in the following example determines the price of a ticket according to the location of the seat in a theater. An integer code that indicates the seat location is the controlling expression:

```java
int seatLocationCode;
< Code here assigns a value to seatLocationCode >

double price = -0.01;
switch (seatLocationCode)
{
    case 1:
        System.out.println("Balcony.");
        price = 15.00;
        break;
    case 2:
        System.out.println("Mezzanine.");
        price = 30.00;
        break;
    case 3:
        System.out.println("Orchestra.");
        price = 40.00;
        break;
    default:
        System.out.println("Unknown ticket code.");
        break;
} // end switch
```

The switch statement contains a list of cases, each consisting of the reserved word case, a constant, a colon, and a list of statements that are the actions for the case. The constant after the word case is called a **case label**. When the switch statement executes, the controlling expression—in this example, seatLocationCode—is evaluated. The list of alternative cases is searched until a case label that matches the current value of the controlling expression is found. Then the action associated with that label is executed. You are not allowed to have duplicate case labels, as that would be ambiguous.

If no match is found, the case labeled default is executed. The default case is optional. If there is no default case, and no match is found to any of the cases, no action takes place. Although the default case is optional, we encourage you to always use it. If you think your cases cover all the possibilities without a default case, you can insert an error message or an assertion as the default case. You never know when you might have missed some obscure case.
Notice that the action for each case in the previous example ends with a break statement. If you omit the break statement, the action just continues with the statements in the next case until it reaches either a break statement or the end of the switch statement. Sometimes this feature is desirable, but sometimes omitting the break statement causes unwanted results.

Note: The controlling expression in a switch statement provides an entry point to a case within the statement. Execution continues from this point until it reaches either a break statement or the end of the switch.

B.52 At times, you will want to take the same action in more than one case. You can list cases one after the other so that they all apply to the same action. In the following example, we have changed the seat location code to a character instead of an integer. A code of B or b, for example, indicates a balcony seat:

```java
char seatLocationCode;
< Code here assigns a value to seatLocationCode >
.
.
double price = -0.01;
switch (seatLocationCode)
{
    case 'B':
    case 'b':
        System.out.println("Balcony.");
        price = 15.00;
        break;
    case 'M':
    case 'm':
        System.out.println("Mezzanine.");
        price = 30.00;
        break;
    case 'O':
    case 'o':
        System.out.println("Orchestra.");
        price = 40.00;
        break;
    default:
        System.out.println("Unknown ticket code.");
        break;
} // end switch
```

The first case, B, has no break statement; in fact, the case has no action statements at all. Execution continues with the case for b, as desired. Note that we have written the cases in two ways to show two common programming styles.

The controlling expression in a switch statement need not be a single variable. It can be a more complicated expression, but it must evaluate to a single value. The expression cannot indicate a range of values. That is, the expression cannot be a boolean expression like the ones you use in an if-else statement. Thus, if you want to take one action when the controlling expression has values from 1 to 10 and a second action for values from 11 to 20, you would need a case label for each value. In situations like this, a switch statement would be harder to write than an if-else statement.

Programming Tip: Omitting a break statement
If you test a program that contains a switch statement and it executes two cases when you expect it to execute only one case, you probably have forgotten to include a break statement where one is needed.
B.53 Our last example of a switch statement revises the previous ones to use strings as the case labels:

```java
String seatLocationCode;
< Code here assigns a value to seatLocationCode >
...
double price = -0.01;
if (seatLocationCode != null)
{
    switch (seatLocationCode)
    {
    case "balcony":
        System.out.println("Balcony.");
        price = 15.00;
        break;
    case "mezzanine":
        System.out.println("Mezzanine.");
        price = 30.00;
        break;
    case "orchestra":
        System.out.println("Orchestra.");
        price = 40.00;
        break;
    default:
        System.out.println("Unknown ticket code.");
        break;
    } // end switch
} // end if
```

Because strings are objects, you must be careful that the value of the controlling expression is not null, as we have done here. You should also take care that the possible strings assigned to seatLocationCode match the case labels exactly. In this example, the labels use lowercase letters. To ensure that the value of the controlling expression uses lowercase letters, we can replace the beginning of the switch statement with

```java
switch (seatLocationCode.toLowerCase())
```

Segment B.76 later in this appendix will describe the String method toLowerCase further.

### Enumerations

B.54 To compute a student’s quality-point average—also known as a grade-point average—a program could assign the number of quality points for a given letter grade to the double variable qualityPoints. You could use a char variable grade for the letter grade, but then it could contain any character, not just the letters A, B, C, D, and F. Instead, to restrict the contents of grade to the values you specify, you could declare it as an enumerated data type, or enumeration. An enumeration itemizes the values that a variable can have.

For example, the following statement defines LetterGrade as an enumeration:

```java
eenum LetterGrade {A, B, C, D, F}
```

LetterGrade behaves as a class type, so we can declare grade to have this type, as follows:

```java
LetterGrade grade;
```

The items listed between the braces in the definition of LetterGrade are objects that grade can reference. These values behave as static constants. For example, you can write

```java
grade = LetterGrade.A;
```

to assign A to grade. You qualify each of them with the name of the enumeration just as you qualify the constant PI with the name of its class Math. Assigning a value other than A, B, C, D, or F to grade will cause a syntax error.
Note: An enumeration is actually a class. Therefore, you cannot define an enumeration within a method. Instead, define enumerations outside of any method definitions, preferably near the beginning of your class. Also, note that no semicolon follows an enumeration’s definition. Writing one, however, will not cause a syntax error; the semicolon will simply be ignored.

Example. You can use a switch statement with a variable whose data type is an enumeration. For example, if we define LetterGrade and grade as in the previous segment, the following switch statement assigns the correct number of quality points to the double variable qualityPoints:

```java
switch (grade)
{
    case A:
        qualityPoints = 4.0;
        break;
    case B:
        qualityPoints = 3.0;
        break;
    case C:
        qualityPoints = 2.0;
        break;
    case D:
        qualityPoints = 1.0;
        break;
    case F:
        qualityPoints = 0.0;
        break;
    default:
        qualityPoints = -9.0;
} // end switch
```

Since the data type of the expression in the switch statement is an enumeration, the case labels are assumed to belong to that enumeration without qualification. In fact, writing

```java
case LetterGrade.A: // Syntax error
```

for example, is a syntax error. However, if you need to reference one of the enumerated values elsewhere within the switch statement, you must qualify it.

Since we know that grade cannot have values other than those in the enumeration, a default case is unnecessary. However, if you choose to omit the default case, you must assign a value to qualityPoints prior to the switch statement to avoid a syntax error. Without this initialization, the compiler would think it possible for qualityPoints to remain uninitialized after the switch statement.

When the compiler encounters an enumeration, it creates a class that has several methods. Among them is the method ordinal, which you can use to access the ordinal value of an object within an enumeration. These values begin at zero. Thus, in LetterGrade, the ordinal values of A and F are 0 and 4, respectively. For example, if you have the following assignment:

```java
LetterGrade yourGrade = LetterGrade.A;
```

the expression

```java
yourGrade.ordinal()
```

returns 0. Likewise, the expression

```java
LetterGrade.B.ordinal()
```

returns 1.
The method `equals` tests whether yourGrade is equal to a given object within the enumeration. For example, you might write

```java
if (yourGrade.equals(LetterGrade.A))
    System.out.println("Congratulations, your grade is A!");
```

Finally, the static method `valueOf` takes a string and returns a matching object in a given enumeration. For example, the expression

```java
LetterGrade.valueOf("A")
```
returns `LetterGrade.A`. The string passed to `valueOf` must match the name of the constant exactly.

Appendix C discusses enumerations further, beginning at Segment C.30.

---

**Scope**

**B.57** The scope of a variable (or a named constant) is the portion of a program in which the variable is available. That is, a variable does not exist outside of its scope. A variable’s scope begins at its declaration and ends at the closing brace of the pair of braces that enclose the variable’s declaration.

For example, consider the following statements that involve two variables, `counter` and `greeting`:

```java
{
    // counter and greeting are not available here
    . . .
    int counter = 1;
    // counter is available here
    . . .
    {
        String greeting = "Hello!";
        // Both greeting and counter are available here
        . . .
    } // end scope of greeting
    . . .
    // Only counter is available here
    . . .
} // end scope of counter
```

The variable `counter` is available anywhere after its declaration. The variable `greeting` is available only within the inner pair of braces.

The concept of scope applies to every pair of braces within a Java program, regardless of whether they delineate the definition of a class or a method, appear within an `if-else` statement or `switch` statement, or appear within the loops described in the next section.

---

**Loops**

**B.58** Programs often need to repeat some action. For example, a grading program would contain some code that assigns a letter grade to a student based on the student’s scores on assignments and exams. To assign grades to the entire class, the program would repeat this action for each student in the class. A portion of a program that repeats a statement or group of statements is called a loop. The statement (or group of statements) to be repeated in a loop is called the body of the loop. Each repetition of the loop body is called an iteration of the loop.

When you design a loop, you need to decide what action the body of the loop should take and when the loop should stop repeating this action. Once you have made these choices, you can pick one of three Java statements to implement the loop: the `while` statement, the `for` statement, or the `do-while` statement.
The while Statement

B.59 One way to construct a loop in Java is with a `while` statement, which is also known as a `while` loop. A `while` statement repeats its action again and again until a controlling boolean expression becomes false. That is, the loop is repeated `while` the controlling boolean expression is true. The general form of a `while` statement is

```java
while (expression)
    statement;
```

The `while` loop starts with the reserved word `while` followed by a boolean expression in parentheses. The loop body is a statement, typically a compound statement enclosed in braces `{}`. The loop body is repeated while the boolean expression is true. The loop body normally contains some action that can change the value of the boolean expression from true to false and so end the loop.

For example, the following `while` statement displays the integers from 1 to a given integer `number`:

```java
int number;
    // Assign a value to number here
int count = 1;
while (count <= number)
{
    System.out.println(count);
    count++;
} // end while
```

Let's suppose that `number` is 2. Since the variable `count` begins at 1, the boolean expression `count <= number` is true at this point, and so the body of the loop executes. Thus, 1 is displayed and then `count` becomes 2. The expression `count <= number` is still true, so the loop’s body executes a second time, displaying 2 and incrementing `count` to 3. Now `count <= number` is false, so the `while` loop ends. Execution continues with the statement, if any, that follows the loop.

Notice that if `number` is zero or negative in the previous example, nothing is displayed. The body of the loop would not execute at all, since `count`, which is 1, would be greater than `number`.

Programming Tip: A `while` loop can perform zero iterations

The body of a `while` loop can execute zero times. When a `while` loop executes, its first action is to check the value of the boolean expression. If the boolean expression is false, the loop body is not executed even one time. Perhaps the loop adds up the sum of all your expenses for the day. If you did not go shopping on a given day, you do not want the loop body to execute at all.

B.60 Infinite loops. A common program bug is a loop that does not end but simply repeats its loop body again and again. A loop that iterates its body repeatedly without ever ending is called an infinite loop. Normally, a statement in the body of the loop will change some variables so that the controlling boolean expression becomes false. If this variable does not change in the right way, you can get an infinite loop.

For instance, let’s consider a slight variation to the previous example of a `while` loop. If we forget to increment `count`, the boolean expression will never change and the loop will be infinite:

```java
int count = 1;
while (count <= number)
{
    System.out.println(count);
} // end while
```
Some infinite loops will not really run forever but will instead end your program abnormally when a system resource is exhausted. However, some infinite loops will run forever if left alone. To end a program that is in an infinite loop, you should learn how to force a program to stop running. The way to do this depends on your particular operating system. For example, in a Unix operating system, you would press the key combination Control-C.

Sometimes a programmer might intentionally write an infinite loop. For example, an ATM machine would typically be controlled by a program with an infinite loop that handles deposits and withdrawals indefinitely. However, at this point in your programming, an infinite loop is likely to be an error.

The for Statement

B.61 When a counter controls the number of iterations in a while loop, you can replace the while statement with a for statement, or for loop. The for statement has the following general form:

\[
\text{for (initialize; test; update) statement;}
\]

Here initialize is an optional assignment of a value to a variable, test is a boolean expression, and update is an optional assignment that can change the value of test.

For example, the following for statement is exactly equivalent to the while statement in Segment B.59:

```java
int count, number;
    . . . // Assign a value to number here
for (count = 1; count <= number; count++)
    System.out.println(count);
```

The first of the three expressions in parentheses, count = 1, initializes the counter before the loop body is executed for the first time. The second expression, count <= number, is a boolean expression that determines whether the loop should continue execution. This boolean expression is tested immediately after the first expression executes and again after each execution of the third expression. The third expression, count++, executes after each iteration of the loop body. Thus, the loop body is executed while count <= number is true.

In the previous example, we declared count before the for statement. After the loop completes its execution, count is still available as a variable. We could instead declare count within the for statement, as follows:

```java
int number;
    . . . // Assign a value to number here
for (int count = 1; count <= number; count++)
    System.out.println(count);
```

In this case, count is defined only within the for loop and is not available after the loop completes its execution.

Programming Tip: Although declaring a variable within a for statement is convenient, realize that the variable’s scope is then the for loop. The variable is not available after the loop completes its execution.

The counter in a for statement is not restricted to an integer type. It can have any primitive type. You can omit any of the expressions initialize, test, and update from a for statement, but you cannot omit their semicolons. Sometimes it is more convenient to write the initialize part before the for statement or to place the update part within the body of the loop. This is especially true when these parts are lengthy. Although you technically can omit the test from a for loop, you will get an infinite loop if you do.
Some Java programmers tend to favor the `for` statement over the `while` statement because in the `for` statement the initialization, testing, and incrementing of the counter all appear at the beginning of the loop. This tendency is especially true for loops that simply count. For more complex logic, a `while` loop is often more appropriate.

**Note:** A `for` statement is basically another notation for a kind of `while` loop. Thus, like a `while` loop, a `for` statement might not execute its loop body at all.

### B.62 The comma in `for` statements

A `for` loop can perform more than one initialization. To use a list of initialization actions, separate the actions with commas, as in the following example:

```java
int n, product;
for (n = 1, product = 1; n <= 10; n++)
    product = product * n;
```

This `for` loop initializes `n` to 1 and `product` to 1. Note that you use a comma, not a semicolon, to separate the initialization actions.

You can have multiple update actions that are separated by commas. This can sometimes lead to a situation in which the `for` statement has an empty body and still does something useful. For example, we can rewrite the previous `for` statement in the following equivalent way:

```java
for (n = 1, product = 1; n <= 10; product = product * n, n++)
```

In effect, we have made the loop body part of the update action. Notice the semicolon at the end of the statement. Since a semicolon at the end of a `for` statement is often the result of a programming error, a clearer way to write this loop makes the empty body explicit:

```java
for (n = 1, product = 1; n <= 10; product = product * n, n++)
{
}
```

However, the most readable style uses the update action only for variables that control the loop, as in the original version of this `for` loop.

Finally, you cannot have multiple boolean expressions to test for ending a `for` loop. However, you can string together multiple tests by using the `&&` and `||` operators to form one larger boolean expression.

**Programming Tip:** If you have used other programming languages that have a general-purpose comma operator, be warned that the comma operator in Java can appear only in `for` statements.

### B.63 Using an enumeration with a `for` statement

The `for` statement has another form when you want to repeat statements for each object in an enumeration. For example, if you define

```java
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}
```

the `for` loop

```java
for (Suit nextSuit : Suit.values())
    System.out.println(nextSuit);
```

displays

```
CLUBS
DIAMONDS
HEARTS
SPADES
```
You declare a variable to the left of a colon in the for statement. To the right of the colon, you represent the values that the variable will have. For the enumeration Suit, the expression Suit.values() represents the four possible values CLUBS, DIAMONDS, HEARTS, and SPADES. As the loop executes, nextSuit takes on each of these values.

This kind of loop—called a for-each loop—can be used with other collections of data, as you will see.

The do-while Statement

B.64 The do-while statement, or do-while loop, is similar to the while statement, but the body of a do-while statement always executes at least once. As you saw earlier, the body of a while loop might not execute at all.

The general form of a do-while statement is

```
do
  statement;
while (expression);
```

The do-while loop starts with the reserved word do. The loop body is a statement, typically a compound statement enclosed in braces {}. The loop ends with the reserved word while followed by a boolean expression in parentheses and a semicolon.

Programming Tip: Be sure to include a semicolon at the end of a do-while statement.

The loop body executes and is repeated while the boolean expression is true. The loop body normally contains some action that can change the value of the boolean expression from true to false and so end the loop. The boolean expression is tested at the end of the loop, not at its beginning, as it is in a while statement. Thus, the loop body executes at least once even if the boolean expression starts out false.

The following do-while statement displays the integers from 1 to a given integer number:

```java
int number;  
  . . . // Assign a value to number here
int count = 1;
do
  {  
    System.out.println(count);
    count++;
  } while (count <= number);
```

Again, let's suppose that number is 2. The variable count begins at 1 and is displayed. Next, count is incremented to 2. Since the expression count <= number is true at this point, the body of the loop executes again. The value of count (2) is displayed, and then count becomes 3. The expression count <= number is now false, so the do-while loop ends. Execution continues with the statement that follows the loop.

If number is zero or negative in the previous example, 1 is displayed, since the body of the loop executes at least once. If number can possibly be zero or negative, we should use either a while loop or a for loop here instead of a do-while loop.

Notice that we placed the ending brace and the while on the same line. Some programmers prefer to place them on different lines. Either form is fine, but be consistent.
To better understand a do-while loop, let’s rewrite the previous example in the following way:

```java
int number;
    // Assign a value to number here
int count = 1;
{
    System.out.println(count);
    count++;
}
while (count <= number)
{
    System.out.println(count);
    count++;
}
```

When you compare the two versions, it is obvious that a do-while loop differs from a while loop in only one detail. With a do-while loop, the loop body is always executed at least once. With a while loop, the loop body might not execute at all.

**Additional Loop Information**

B.65 **Choosing a loop statement.** Suppose you decide that your program needs a loop. How do you decide whether to use a while statement, a for statement, or a do-while statement? You cannot use a do-while statement unless you are certain that the loop body should execute at least one time. If you are certain of this, a do-while statement is likely to be a good choice. However, more often than you might think, a loop requires the possibility that the body will not execute at all. In those cases, you must use either a while statement or a for statement. If it is a computation that changes some numeric quantity by some equal amount on each iteration, consider a for statement. If the for statement does not work well, use a while statement. The while statement is always a safe choice, since you can use it for any sort of loop. But sometimes one of the other alternatives is easier or clearer.

---

**Programming Tip:** A while loop can do anything that another loop can do.

B.66 **The break and continue statements in loops.** You can use the break statement in a switch statement or in any kind of loop statement. When the break statement executes in a loop, the immediately enclosing loop ends, and the remainder of the loop body is not executed. Execution continues with the statement after the loop.

Adding a break statement to a loop can make the loop more difficult to understand. Without a break statement, a loop has a simple, easy-to-understand structure. There is a test for ending the loop at the top (or bottom) of the loop, and every iteration will go to the end of the loop body. When you add a break statement, the loop might end because either the condition given at the top (or bottom) of the loop is false or the break statement has executed. Some loop iterations may go to the end of the loop body, but one loop iteration might end prematurely. Because of the complications they introduce, you should avoid break statements in loops. Some authorities contend that a break statement should never be used to end a loop, but virtually all programming authorities agree that they should be used at most sparingly.

The continue statement ends the current iteration of a loop. The loop continues with the next iteration. Using a continue statement in this way has the same problems as using a break
Programming Tip: In general, do not use break or continue statements within the body of a loop.

The Class String

B.67 Strings of characters, such as "Enter the amount:“, do not have a primitive type in Java. However, Java does provide a class, called String, that you use to create and process strings of characters. The string constant "Enter the amount:“, in fact, is a value of type String. The class String is part of the package java.lang in the Java Class Library.

A variable of type String can name one of these string values. The statement

```java
String greeting;
```

declares greeting to be the name of a String variable, and the following statement sets the value of greeting to the String value "Hello!":

```java
greeting = "Hello!";
```

These two statements are often combined into one, as follows:

```java
String greeting = "Hello!";
```

We now can display greeting on the screen by writing

```java
System.out.println(greeting);
```

The screen will show

```
Hello!
```

Characters Within Strings

B.68 Most programming languages use the ASCII character set, which assigns a standard number to each of the characters normally used on an English-language keyboard. Java, however, uses the Unicode character set instead. The Unicode character set includes all the ASCII characters plus many of the characters used in languages that have an alphabet different from English. As it turns out, this is not likely to be a big issue if you are using an English-language keyboard. Normally, you can just program as if Java were using the ASCII character set, since the codes for the ASCII characters are the same in Unicode. The advantage of the Unicode character set is that it allows you to easily handle languages other than English. The disadvantage of the Unicode character set is that it requires two bytes to store each character, whereas the ASCII character set requires one.

B.69 Escape characters. Suppose we want to display the following line on the screen:

```
The word “Java” names a language and a drink!
```

This string contains quotes, so the statement

```java
System.out.println("The word "Java" names a language and a drink!");
```

will not work: It produces a compiler error message. The problem is that the compiler sees

"The word "

as a perfectly valid quoted string. Then the compiler sees

```
Java"
```
which is not valid in the Java language. The compiler does not know that we mean to include the quote character as part of the string unless we tell it that we want to do so. We tell the compiler this by placing a backslash \ before the troublesome character, like so:

```java
System.out.println("The word "Java" names a language and a drink!");
```

Some other special characters also need a backslash in order to be included in strings. They are listed in Figure B-4. These are often called escape characters because they escape from the usual meaning of a character in Java, such as the usual meaning of the double-quote character.

### FIGURE B-4 Escape characters

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td>Double quote.</td>
</tr>
<tr>
<td>'</td>
<td>Single quote (apostrophe).</td>
</tr>
<tr>
<td>\</td>
<td>Backslash.</td>
</tr>
<tr>
<td>\n</td>
<td>New line. (Go to the beginning of the next line.)</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return. (Go to the beginning of the current line.)</td>
</tr>
<tr>
<td>\t</td>
<td>Tab. (Insert whitespace up to the next tab stop.)</td>
</tr>
</tbody>
</table>

It is important to note that each escape sequence is a single character, even though it is spelled with two symbols. So the string "Say "Hi"!" contains 9 characters, not 11.

To include a backslash in a string, you must write two backslashes. Displaying the string "abc\def" on the screen would produce

```java
abc
def
```

Writing the string with only one backslash, as in "abc\def", is likely to produce the error message “Invalid escape character,” because \d is invalid.

The escape sequence \n indicates that the string starts a new line at the \n. For example, the statement

```java
System.out.println("The motto is\nGo for it!");
```

will write the following to the screen

```
The motto is
Go for it!
```

You can include a single quote (apostrophe) inside a quoted string, such as "How's this?", but you cannot write a single quote within single quotes. Thus, to define a single-quote character, you use the escape sequence \', as follows:

```java
char singleQuote = '\';
```

### Concatenation of Strings

B.70 You can join two strings by using the + operator. Joining two strings together, end to end, to obtain a larger string is called concatenation. When + is used with strings, it is sometimes called the concatenation operator. For example, the statements

```java
String greeting = "Hello";
String sentence = greeting + "my friend.";
System.out.println(sentence);
```

set the variable sentence to "Hellomy friend." and will write the following on the screen:

```
Hellomy friend.
```
No space separates the first two words, because no spaces are added when you concatenate two strings. If we want sentence to contain "Hello my friend.", we could change the assignment statement to

```
sentence = greeting + " my friend."
```

Notice the space before the word “my.”

You can concatenate any number of `String` objects by using the `+` operator. You can even concatenate a `String` object to any other type of object and get a `String` object as a result. Java can express any object as a string when you concatenate it to a string. For primitives like numbers, Java does the obvious thing. For example,

```
String solution = "The answer is " + 42;
```

will set the `String` variable `solution` to "The answer is 42". This is so natural that it may seem as though nothing special is happening, but it does require a real conversion from one type to another. The Java literal `42` is an integer, whereas "42" is a string consisting of the two characters 4 and 2. Java converts the integer constant `42` to the string constant "42" and then concatenates the two strings "The answer is " and "42" to obtain the longer string

"The answer is 42".

---

**Note:** Every class has a method `toString` that Java uses to get a string representation of any object. If you do not define `toString` for a class that you write, the default `toString` will return a representation of an object’s location in memory. Thus, you generally should provide your own `toString` method when you define a class. Appendixes C and D discuss this method in more detail.

You can also concatenate a single character to a string by using `. For example,

```
String label = "mile";
String pluralLabel = label + 's';
```

sets `pluralLabel` to the string "miles".

Segment B.73 will show you another way to concatenate strings.

### String Methods

Readers who need to review Java methods should consult Appendix C before reading this section.

**B.71** A `String` object has methods as well as a value. You use these methods to manipulate string values. A few of these `String` methods are described here. You invoke, or call, a method for a `String` object by writing the object name, a dot, and the name of the method, followed by a pair of parentheses. Some methods require nothing within the parentheses, while others require that you specify arguments. Let’s look at some examples.

**B.72** The method `length`. The method `length` gets the number of characters in a string. For example, suppose we declare two `String` variables as follows:

```
String command = "Sit Fido!";  // 9 characters
String answer = "bow-wow";    // 7 characters
```

Now `command.length()` has the value 9 (it returns 9), and `answer.length()` returns 7. Notice that you must include a pair of parentheses, even though there are no arguments to the method `length`. Also notice that spaces, special symbols, and repeated characters are all counted in computing the length of a string. All characters except the double quotes that enclose the string are counted.
The Class String

You can use a call to the method `length` anywhere that you can use a value of type `int`. For example, all of the following are legal Java statements:

```java
int count = command.length();
System.out.println("Length is " + command.length());
count = answer.length() + 3;
```

B.73 The method `concat`. You can use the method `concat` instead of the `+` operator to concatenate two strings. For example, if we declare the `String` variables

```java
String one = "sail";
String two = "boat";
```

the expressions

```java
one + two
```

and

```java
one.concat(two);
```

are the same string, `sailboat`.

B.74 Indices. Some of the methods in the class `String` refer to the positions of the characters in the string. Positions in a string start with 0, not with 1. Thus, in the string "Hi Mom", `H` is in position 0, `i` is in position 1, the blank character is in position 2, and so forth. A position is usually referred to as an index. So it would be more normal to say that `H` is at index 0, `i` is at index 1, and so on. Figure B-5 illustrates how index positions are numbered in a string.

FIGURE B-5 Indices 0 through 11 for the string "Java is fun."

```
0 1 2 3 4 5 6 7 8 9 10 11
J a v a  i s  f u n .
```

B.75 The methods `charAt` and `indexOf`. The method `charAt` returns the character at the index given as its one argument. For example, the statements

```java
String phrase = "Time flies like an arrow."
char sixthCharacter = phrase.charAt(5);
```

assign the character `f` to the variable `sixthCharacter`, since the `f` in `flies` is at index 5. (Remember, the first index is 0, not 1.)

The method `indexOf` tests whether a string contains a given substring and, if it does, returns the index at which the substring begins. Thus, `phrase.indexOf("flies")` will return 5 because the substring `flies` begins at index 5 within `phrase`.

Note: Out-of-bounds index

String methods such as `charAt` that take an index as an argument will cause an error during execution if the index is negative or too large. Such an index is said to be out of bounds. The error causes a `StringIndexOutOfBoundsException`. Java Interludes 2 and 4 discuss exceptions.

B.76 Changing case. The method `toLowerCase` returns a string obtained from its argument string by replacing any uppercase letters with their lowercase counterparts. Thus, if `greeting` is defined by

```java
String greeting = "Hi Mary!";
```
the expression

greeting.toLowerCase()
returns the string "hi mary!". An analogous method, toUpperCase, converts any lowercase letters in a string to uppercase. Note the capital C in toLowerCase and toUpperCase.

B.77 The method trim. The method trim trims off any leading and trailing white space, such as blanks. So the statements

```java
String command = "   Sit Fido!   ";
String trimmedCommand = command.trim();
```
set trimmedCommand to the string "Sit Fido!". The blanks between words are not affected.

B.78 Comparing strings. You use the method compareTo to compare two strings. Strings are ordered according to the Unicode values of their characters. This ordering—called lexicographic ordering—is analogous to alphabetic ordering. The expression

```java
stringOne.compareTo(stringTwo)
```
returns a negative integer or a positive integer, depending on whether stringOne occurs before or after stringTwo. The expression returns zero if the two strings are equal.

The method compareToIgnoreCase behaves similarly to compareTo, except that the uppercase and lowercase versions of the same letter are considered to be equal. For example, the method compareTo places the string "Hello" before the string "hello", but the method compareToIgnoreCase finds these strings to be equal.

Programming Tip: Do not use the operators ==, !=, <, <=, >, or >= to compare the contents of two strings.

If you want only to see whether two strings are equal—that is, contain the same values—you can use the method equals. Thus,

```java
stringOne.equals(stringTwo)
```
is true if stringOne equals stringTwo and is false if they are not equal. The method equalsIgnoreCase behaves similarly to equals, except that the uppercase and lowercase versions of the same letter are equal. For example, the method equals finds the strings "Hello" and "hello" unequal, but the method equalsIgnoreCase finds them equal.

Programming Tip: When applied to two strings (or to any two objects), the operator == tests whether they are stored in the same memory location. Sometimes that is sufficient, but if you want to know whether two strings that are in different memory locations contain the same sequence of characters, use the method equals.

The Class StringBuilder

B.79 Once you create a string object of the class String, you cannot alter it. But sometimes you would like to. For example, we might define the string

```java
String name = "rover";
```
and then decide that we want to capitalize its first letter. We cannot. We could, of course, write

```java
name = "Rover";
```
but this statement creates a new string Rover and discards rover.
The class String has no method that modifies a String object. However, the class StringBuilder in the package java.lang has methods such as the following:

```java
public StringBuilder append(String s)
Concatenates the string s to the end of this string and returns a reference to the result.
```

```java
public StringBuilder delete(int start, int after)
Removes the substring of this string beginning at the index start and ending at either the index after - 1 or the end of the string, whichever occurs first, and returns a reference to the result. Throws StringIndexOutOfBoundsException if start is invalid.
```

```java
public StringBuilder insert(int index, String s)
Inserts the string s into this string at the given index and returns a reference to the result. Throws StringIndexOutOfBoundsException if the index is invalid.
```

```java
public StringBuilder replace(int start, int after, String s)
Replaces a substring of this string with the string s. The substring to be replaced begins at the index start and ends at either the index after - 1 or the end of the string, whichever occurs first. Returns a reference to the result. Throws StringIndexOutOfBoundsException if start is invalid.
```

```java
public void setCharAt(int index, char character)
Sets the character at the given index of this string to a given character. Throws IndexOutOfBoundsException if the index is invalid.
```

If you are not familiar with exceptions, think of them as error messages for now. Java Interludes 2 and 4 will explain them to you.

StringBuilder has other versions of the methods append and insert that take data of any type as arguments. While the classes StringBuilder and String have some methods in common, several methods in String are not in StringBuilder.

### B.80 Examples

If we have the following instance of StringBuilder

```java
StringBuilder message = new StringBuilder("rover");
```

we can capitalize its first letter by writing

```java
message.setCharAt(0, 'R');
```

Now the statement

```java
message.append("", roll over!");
```
changes message to the string Rover, roll over!, and

```java
message.insert(7, "Rover, ");
```
changes it to Rover, Rover, roll over! Next,

```java
message.delete(0, 7);
```
changes message to Rover, roll over!, and

```java
message.replace(7, 16, "come here");
```
changes message to Rover, come here!

Each of the previous methods except setCharAt returns the result of the operation. So, for example, if we write

```java
newMessage = message.append("", roll over!");
```
both newMessage and message reference the same sequence of characters.
Using Scanner to Extract Pieces of a String

B.81 Segments B.32 through B.37 show how the class Scanner is used to read data that a user types at the keyboard. Methods in Scanner—such as `nextInt`, `nextDouble`, and `next`—read a group of contiguous characters, or token, as an integer, a real number, or a string, respectively. When more than one token appears in the input data, they are separated by one or more characters known as delimiters. By default, Scanner uses white space as the delimiter.

In addition to reading data from the keyboard, you can use Scanner to process a string that you define within your program. You simply use the string instead of `System.in` when creating a Scanner object.

For example, the statements

```java
String phrase = "one potato        two    potato three potato four";
Scanner scan = new Scanner(phrase);
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
```

display the words

one
potato
two
potato

We essentially have read tokens from a string instead of from the keyboard. The tokens here are separated by white space, the default delimiter.

B.82 Specifying delimiters. Whether you read characters from an input device such as a keyboard or from a string defined in your program, you can specify the delimiters that Scanner will use. The Scanner method `useDelimiter` sets the delimiters to those indicated in its string argument. For example, to use a comma as a delimiter, you would write

```java
scan.useDelimiter(",");
```

where `scan` is a Scanner object. Thus, the statements

```java
String data = "one,potato,two,potato";
Scanner scan = new Scanner(data);
scan.useDelimiter(",");
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
```

display the words

one
potato
two
potato

B.83 Example. To extract the words one, two, three, and four from the string phrase that we defined in Segment B.81, you would describe the delimiter as one or more blanks, the word potato, and one or more blanks. You use the notation in Figure B-6 to describe the delimiters. For example, \s+ denotes one or more white-space characters. You must duplicate each backslash character in this
notation when it appears between the quotes of a string literal to distinguish it from escape characters such as \n.

**FIGURE B-6** Some notation used to define the delimiters that Scanner uses

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>Any digit 0 through 9</td>
</tr>
<tr>
<td>\D</td>
<td>Any character other than a digit</td>
</tr>
<tr>
<td>\s</td>
<td>Any white-space character</td>
</tr>
<tr>
<td>\S</td>
<td>Any character other than white space</td>
</tr>
<tr>
<td>\w</td>
<td>Any letter, digit, or underscore</td>
</tr>
<tr>
<td>\W</td>
<td>Any character other than a letter,</td>
</tr>
<tr>
<td></td>
<td>digit, or underscore</td>
</tr>
<tr>
<td>-</td>
<td>Any character</td>
</tr>
<tr>
<td>X</td>
<td>One occurrence of X</td>
</tr>
<tr>
<td>X?</td>
<td>Zero or one occurrence of X</td>
</tr>
<tr>
<td>X*</td>
<td>Zero or more occurrences of X</td>
</tr>
<tr>
<td>X+</td>
<td>One or more occurrences of X</td>
</tr>
<tr>
<td>X{n}</td>
<td>Exactly n occurrences of X</td>
</tr>
<tr>
<td>X{n,}</td>
<td>At least n occurrences of X</td>
</tr>
</tbody>
</table>

The following statement sets the delimiter of the Scanner object scan to the word potato with leading and trailing white-space characters:

```java
scan.useDelimiter("\s+potato\s+");
```

Thus, the statements

```java
String phrase = "one potato two potato three potato four";
Scanner scan = new Scanner(phrase);
scan.useDelimiter("\s+potato\s+");
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
```

display the words

one
two
three
four

**Example.** The statements

```java
String phrase = "5 potato 6 potato 7 potato more";
Scanner scan = new Scanner(phrase);
scan.useDelimiter("\s*\d\s*");
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
```

display the words

potato
potato
potato more
The delimiter is a digit with optional leading and trailing white space.

If you now write

```java
    scan = new Scanner(phrase);
    scan.useDelimiter("\\s*potato\\s*");
    System.out.println(scan.nextInt());
    System.out.println(scan.nextInt());
    System.out.println(scan.next());
    System.out.println(scan.next());
```

you will get the following output

```
5
6
7
more
```

Here the delimiter is the word *potato* with optional leading and trailing white space.

**Example.** The statements

```java
    String phrase = "one - two   -    three four";
    Scanner scan = new Scanner(phrase);
    scan.useDelimiter("\\s+-?\\s*");
    System.out.println(scan.next());
    System.out.println(scan.next());
    System.out.println(scan.next());
    System.out.println(scan.next());
```

display the words

```
one
two
three
four
```

The delimiter is at least one white space character optionally followed by a dash and more white space.

### Arrays

**B.86** In Java, an array is a special kind of object that stores a finite collection of items having the same data type. For example, you can create an array of seven numbers of type `double` as follows:

```java
    double[] temperature = new double[7];
```

The left side of the assignment operator declares `temperature` as an array whose contents are of type `double`. The right side uses the `new` operator to request seven memory locations for the array. This is like declaring the following strangely named variables to have type `double`:

```java
    temperature[0], temperature[1], temperature[2], temperature[3],
    temperature[4], temperature[5], temperature[6]
```

*Note that the numbering starts with 0, not 1.* Each of these seven variables can be used just like any other variable of type `double`. For example, we can write

```java
    temperature[3] = 32;
    System.out.println(temperature[6]);
```

But these seven variables are more than just seven plain old variables of type `double`. The number in square brackets—called an *index*, or *subscript*—can be any arithmetic expression whose value is an integer. In this example, the index value must be between 0 and 6, because we declared
the array temperature to have seven memory locations. A variable such as temperature[3] is called either an **indexed variable**, a **subscripted variable**, or simply an **element** of the array. The value in an indexed variable is called an **entry**, and its data type is the array's **entry type**.

For example, the following statements read seven temperatures into an array and compare them with their average:

```java
public static final int DAYS_PER_WEEK = 7;
Scanner keyboard = new Scanner(System.in);

double[] temperature = new double[DAYS_PER_WEEK];
System.out.println("Enter " + DAYS_PER_WEEK + " temperatures: ");
double sum = 0;
for (int index = 0; index < DAYS_PER_WEEK; index++)
    { 
        temperature[index] = keyboard.nextDouble();
        sum = sum + temperature[index];
    } // end for

double average = sum / DAYS_PER_WEEK;
System.out.println("The average temperature is "+ average);
System.out.println("The temperatures are");
for (int index = 0; index < DAYS_PER_WEEK; index++)
    { 
        if (temperature[index] < average)
            System.out.println("temperature["+index+]" below average.");
        else if (temperature[index] > average)
            System.out.println("temperature["+index+]" above average.");
        else // temperature[index] == average
            System.out.println("temperature["+index+]" average.");
    } // end for
```

Figure B-7 illustrates the array temperature after seven values have been read into it.

**FIGURE B-7**  An array of seven temperatures

<table>
<thead>
<tr>
<th>Indices</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
<td>30</td>
<td>25.7</td>
<td>26</td>
<td>34</td>
<td>31.5</td>
<td>29</td>
</tr>
</tbody>
</table>

Each location in the array temperature contains a temperature. That is, the array is full. But arrays are not always full. You need to distinguish between the number of locations in an array—its **length**—and the number of items currently stored in the array.

An array has a data field **length** that contains the declared number of elements in the array. For example, if we create an array by writing

```java
int[] age = new int[50];
```

then age.length is 50. Notice that **length** is not a method, so no parentheses follow it. If we place only 10 values into the first 10 locations of this array, age.length is still 50. If we need to know how many values we place into an array, we will need to keep track of that ourselves.
**Note:** An array has a data type and an entry type
The data type of the previously defined array age is `int[]`, but its entry type is `int`.

### Array Parameters and Returned Values

Readers who need to review Java methods should consult Appendix C before reading this section.

**B.87 Array parameters.** You can pass an indexed variable as an argument to a method anyplace that you can pass an ordinary variable of the array’s entry type. For example, if a method has the header

```java
public double compute(double value)
```

and temperature is the array of `double` values that we defined earlier, we can invoke the method by writing

```java
double result = compute(temperature[3])
```

An entire array can also be a single argument to a method. For example, the following method

```java
public static void incrementArrayBy2(double[] array)
{
    for (int index = 0; index < array.length; index++)
        array[index] = array[index] + 2;
} // end incrementArrayBy2
```

will accept any array of `double` values as its single argument. We declare the parameter array in the method’s header just as we would declare any other array: by specifying the type of the array entries followed by square brackets. We do not specify the length of the array.

The following statement is an example of how we would invoke this method:

```java
incrementArrayBy2(temperature);
```

You use no square brackets when you pass an entire array as an argument to a method. Notice that the method can take any length array as an argument. The method `incrementArrayBy2` adds 2 to each entry in the argument array `temperature`. That is, the method actually changes the values in the argument array.

**Note:** A method can change the values in an argument array. However, a method cannot replace an argument array with another array.

**B.88 Arrays as return values.** In Java, a method can return an array. For example, rather than modifying its array argument, the previous method `incrementArrayBy2` could return an array whose values are 2 more than the corresponding values in the array argument. You specify the method’s return type in the same way that you specify a type for an array parameter. The method would then look like this:

```java
public static double[] incrementArrayBy2(double[] array)
{
    double[] result = new double[array.length];
    for (int index = 0; index < array.length; index++)
        result[index] = array[index] + 2;
    return result;
} // end incrementArrayBy2
```

The following statements invoke this method:

```java
double[] originalArray = new double[10];
< Statements that place values into originalArray >
```
Arrays

double[] revisedArray = incrementArrayBy2(originalArray);
< At this point, originalArray is unchanged. >

Programming Tip: When a method returns an array that you want to assign to an array variable, you should declare the variable but not allocate memory for an array. For example, to invoke the method in the previous example, you do not write

double[] revisedArray = new double[10]; // WRONG!
revisedArray = incrementArrayBy2(originalArray);

The first statement allocates 10 locations for a new array, but these locations are discarded when the second statement executes.

Initializing Arrays

You can provide initial values for the elements in an array when you declare it. To do this, you enclose the values for the array in braces and place them after the assignment operator, as in the following example:

double[] reading = {3.3, 15.8, 9.7};

You do not explicitly state the array’s length. Instead the length is the minimum number of locations that will hold the given values. This initializing declaration is equivalent to the following statements:

double[] reading = new double[3];
reading[0] = 3.3;
reading[1] = 15.8;
reading[2] = 9.7;

If you do not initialize the elements of an array, they are given initial default values according to their type. For example, if you do not initialize an array of integers, each element of the array will be initialized to zero. In an array of objects, each element is initialized to null. However, it is usually clearer to do your own explicit initialization, either when you declare the array or later by using a loop and assignment statements.

Array Index Out of Bounds

When programming with arrays, making a mistake with an index is easy. This is especially true if that index is an expression. If the array temperature has seven elements, but an index is some integer other than 0 through 6, the index is said to be out of bounds. An out-of-bounds index expression will compile without any error message, but will cause an error when you run your program. In particular, you will get an IndexOutOfBoundsException. As we mentioned in Segment B.75, a similar situation can occur when you work with strings. (Java Interludes 2 and 4 discuss exceptions.)

Use of = and == with Arrays

The operator =. Recall that a variable for an object really contains the memory address of the object. The same is true of arrays. All array locations are together in one section of memory so that one memory address can specify the location of the entire array. The assignment operator copies this memory address. For example, if a and b are arrays of the same size, the assignment b = a gives the array variable b the same memory address as the array variable a. In other words, a and b are two different names for the same array. These variables are aliases. Thus, when you change the value of a[2], you are also changing the value of b[2]. Appendix C talks more about aliases and references.
If you want the array b to have the same values as the array a, but in separate memory locations, then instead of one assignment statement you must use a loop like the following one:

```java
for (int index = 0; index < a.length; index++)
    b[index] = a[index];
```

**B.92 The operator ==.** The equality operator == tests two arrays to see if they are stored in the same place in the computer’s memory. It does not test whether the arrays contain the same values. To do so, you must compare the two arrays entry by entry. For example, if the arrays a and b contain primitive values and have the same length, the following code could be used:

```java
boolean match = true;
int index = 0;
while (match && (index < a.length))
{
    if (a[index] != b[index])
        match = false;
    else
        index++;
} // end while
if (match)
    System.out.println("Arrays have the same contents");
else
    System.out.println("Arrays have different contents");
```

If the arrays contained objects instead of primitive values, we would use the boolean expression !a[index].equals(b[index]) instead of a[index] != b[index].

---

**Note: Are arrays really objects?**

Arrays behave very much like objects. On the other hand, arrays do not belong to any class. Because arrays were used by programmers for many years before classes and objects (as we have used them) were invented, arrays use a special notation of their own. Other features of objects do not apply to arrays, such as inheritance (which we discuss in Appendix D). So whether or not arrays should be considered objects is primarily an academic debate. Whenever Java documentation says that something applies to all objects, it also applies to arrays.

---

**Note: Array types are reference types**

A variable of an array type holds only the address where the array is stored in memory. This memory address is often called a reference to the array object in memory. For this reason, an array type is a reference type. A reference type is any type whose variables hold references—that is, memory addresses—as opposed to the actual item named by the variable. Array types and class types are both reference types. Primitive types are not reference types.

---

**Arrays and the For-Each Loop**

**B.93 Earlier in Segment B.63 we used a for-each loop to process all the values in an enumeration. We can use a similar for-each loop to process all the values in an array. For example, the following statements compute the sum of the integers in an array:**

```java
int[] anArray = {1, 2, 3, 4, 5};
int sum = 0;
```
for (int integer : anArray)
    sum = sum + integer;
System.out.println(sum);

Similarly, the following statements display all the strings in an array:

String[] friends = {"Gavin", "Gail", "Jared", "Jessie"};
for (String name : friends)
    System.out.println(name);

Multidimensional Arrays

B.94 You can have an array with more than one index. For example, suppose we wanted to store the dollar amounts shown in Figure B-8 in some sort of array. The bold items are just labeling. There are 60 entries. If we use an array with one index, the array will have a length of 60, and keeping track of which entry goes with which index would be almost impossible. On the other hand, if we allow ourselves two indices, we can use one index for the row and one index for the column. This arrangement is illustrated in Figure B-9.

**FIGURE B-8** A table of values

<table>
<thead>
<tr>
<th>Year</th>
<th>5.00%</th>
<th>5.50%</th>
<th>6.00%</th>
<th>6.50%</th>
<th>7.00%</th>
<th>7.50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1050</td>
<td>$1055</td>
<td>$1060</td>
<td>$1065</td>
<td>$1070</td>
<td>$1075</td>
</tr>
<tr>
<td>2</td>
<td>$1103</td>
<td>$1113</td>
<td>$1124</td>
<td>$1134</td>
<td>$1145</td>
<td>$1156</td>
</tr>
<tr>
<td>3</td>
<td>$1158</td>
<td>$1174</td>
<td>$1191</td>
<td>$1208</td>
<td>$1225</td>
<td>$1242</td>
</tr>
<tr>
<td>4</td>
<td>$1216</td>
<td>$1239</td>
<td>$1262</td>
<td>$1286</td>
<td>$1311</td>
<td>$1335</td>
</tr>
<tr>
<td>5</td>
<td>$1276</td>
<td>$1307</td>
<td>$1338</td>
<td>$1370</td>
<td>$1403</td>
<td>$1436</td>
</tr>
<tr>
<td>6</td>
<td>$1340</td>
<td>$1379</td>
<td>$1419</td>
<td>$1459</td>
<td>$1501</td>
<td>$1543</td>
</tr>
<tr>
<td>7</td>
<td>$1407</td>
<td>$1455</td>
<td>$1504</td>
<td>$1554</td>
<td>$1606</td>
<td>$1659</td>
</tr>
<tr>
<td>8</td>
<td>$1477</td>
<td>$1535</td>
<td>$1594</td>
<td>$1655</td>
<td>$1718</td>
<td>$1783</td>
</tr>
<tr>
<td>9</td>
<td>$1551</td>
<td>$1619</td>
<td>$1689</td>
<td>$1763</td>
<td>$1838</td>
<td>$1917</td>
</tr>
<tr>
<td>10</td>
<td>$1629</td>
<td>$1708</td>
<td>$1791</td>
<td>$1877</td>
<td>$1967</td>
<td>$2061</td>
</tr>
</tbody>
</table>

Note that, as was true for the simple arrays you have already seen, you begin numbering indices with 0 rather than 1. If the array is named `table` and it has two indices, the Java notation `table[3][2]` specifies the entry in the fourth row and third column. By convention, we think of the first index as denoting the row and the second as denoting the column. Arrays that have exactly two indices can be displayed on paper as a two-dimensional table and are called **two-dimensional arrays**. More generally, an array is said to be an **n-dimensional array** if it has `n` indices. Thus, the ordinary one-index arrays that we have used up to now are **one-dimensional arrays**.
Arrays with multiple indices are handled much like arrays with one index. To declare and create the array \emph{table} with 10 rows and 6 columns, we write

```
int[][] table = new int[10][6];
```

You can have arrays with any number of indices. To get more indices, you just use more square brackets in the declaration.

Indexed variables for multidimensional arrays are just like indexed variables for one-dimensional arrays, except that they have multiple indices, each enclosed in a pair of square brackets. For example, the following statements set all the elements in \emph{table} to zero:

```
for (int row = 0; row < 10; row++)
    for (int column = 0; column < 6; column++)
        table[row][column] = 0;
```

Note that we used two \emph{for} loops, one nested within the other. This is a common way of stepping through the indexed variables in a two-dimensional array. If we had three indices, we would use three nested \emph{for} loops, and so forth for higher numbers of indices.

As was true of the indexed variables for one-dimensional arrays, indexed variables for multidimensional arrays are variables of the array's entry type and can be used anywhere that a variable of the entry type is allowed. For example, for the two-dimensional array \emph{table}, an indexed variable such as \emph{table[3][2]} is a variable of type \emph{int} and can be used anywhere that an ordinary \emph{int} variable can be used.

A multidimensional array can be a parameter of a method. For example, the following method header has a two-dimensional array as a parameter:

```
public static void clearArray(double[][] array)
```

Java implements multidimensional arrays as one-dimensional arrays. For example, consider the array

```
int[][] table = new int[10][6];
```

The array \emph{table} is in fact a one-dimensional array of length 10, and its entry type is \emph{int[6]}. Thus, each entry in the array \emph{table} is a one-dimensional array of length 6. In other words, a multidimensional array is an array of arrays.

Normally, you do not need to be concerned with this fact, since this detail is handled automatically by the compiler. However, sometimes you can profit from this knowledge. For example, consider the previous nested \emph{for} loops that filled the two-dimensional array \emph{table} with zeros. We used the constants 6 and 10 to control the \emph{for} loops, but it would be better style to use the data
Wrapper Classes

B.97 Java makes a distinction between the primitive types, such as int, double, and char, and the class types, such as String and the classes that you write. Java sometimes treats primitive types and class types differently. For example, an argument to a method and the assignment operator = behave differently for primitive types and class types. To make things uniform, Java provides a wrapper class for each of the primitive types that enables us to convert a value of a primitive type to an object of a corresponding class type.

For example, the wrapper class for the primitive type int is the predefined class Integer. If we want to convert an int value, such as 10, to an object of type Integer, we can do so in one of three ways, as the following statements demonstrate:

```java
Integer ten = new Integer(10);
Integer fiftyTwo = new Integer("52");
Integer eighty = 80;
```

In the first way, you supply an int value as a literal, a variable, or an expression. In the second, you provide a string that contains an int value. The third way allows you to simply assign an int value without using the new operator.

B.98 Once you have defined Integer objects, you can compare them by using the methods compareTo and equals, much as you compare strings or the other objects we have discussed in this appendix. Do not use operators such as ==. Just as for strings and other objects, operators like == compare the memory addresses of objects, not their values.

If you need the value of an Integer object as a primitive, you can use methods such as intValue or doubleValue. For example, if ten is defined as in the previous segment, the expression

```java
ten.intValue()
```

returns the int value 10, whereas

```java
ten.doubleValue()
```

returns the double value 10.0. You also can simply assign the Integer object to an int variable, as in

```java
int primitive10 = ten;
```

No type cast is necessary; however, using one is not an error.

B.99 Boxing and unboxing. When performing arithmetic with Integer objects, for example, you can use the same operators that you use for arithmetic with primitives. You can also intermix primitive integers with Integer objects. Thus, you can write statements such as the following:

```java
Scanner keyboard = new Scanner(System.in);
System.out.print("What is his age? ");
int hisAge = keyboard.nextInt();
System.out.print("What is her age? ");
Integer herAge = keyboard.nextInt();
Integer ageDifference = Math.abs(hisAge - herAge);
System.out.println("He is " + hisAge + ", she is " + herAge + ": a difference of " + ageDifference + ").");
```
Java converts between int and Integer as necessary. The process of converting from a primitive type to a corresponding wrapper class is called **boxing**. **Unboxing** is the process used to convert in the other direction. Since these conversions happen automatically, they are often called **auto-boxing** and **auto-unboxing**. Realize that the previous statements are just a demonstration of what is possible. Using only primitive integers for this simple computation is certainly adequate.

**B.100** The wrapper classes for the primitive types double, float, long, and char are Double, Float, Long, and Character, respectively. You create objects of these classes in a way analogous to how you create Integer objects. Except for Character, each wrapper class has methods that return a value in a variety of types. Integer has the methods doubleValue, floatValue, intValue, and longValue. Double, Float, and Long also have these same methods. The class Character has only the analogous method charValue.

Many of the classes that we study in this book represent collections of objects. If your data has a primitive type, an appropriate wrapper class enables you to represent the data as objects so that you can use these classes.

**B.101** Wrapper classes also contain some useful static constants. For example, you can find the largest and smallest values of any of the primitive number types by using the associated wrapper class. The largest and smallest values of type int are

```
Integer.MAX_VALUE and Integer.MIN_VALUE
```

The largest and smallest values of type double are

```
Double.MAX_VALUE and Double.MIN_VALUE
```

**B.102** Wrapper classes have static methods that can be used to convert a string to the corresponding number of type int, double, long, or float. For example, suppose your program needs to convert the string "199.98" to a double value (which will turn out to be 199.98, of course). The static method parseDouble of the wrapper class Double will convert a string to a value of type double. So if theString is a variable of type String whose value is "199.98",

```
Double.parseDouble(theString)
```

returns the double value 199.98. The other wrapper classes Integer, Long, and Float have the analogous methods parseInt, parseLong, and parseFloat.

If there is any possibility that the string named by theString has extra leading or trailing blanks, you should instead use

```
Double.parseDouble(theString.trim())
```

As we discussed in Segment B.77, the method trim, included in the class String, trims off leading or trailing white space, such as blanks. If the string is not a correctly formed number, the invocation of Double.parseDouble will cause an exception. The use of trim helps some in avoiding this problem.

**B.103** Each of the numeric wrapper classes also has a static method called toString that will convert in the other direction—that is, it will convert from a primitive numeric value to a string representation of the numeric value. For example,

```
Integer.toString(42)
```

returns the string value "42", and

```
Double.toString(199.98)
```

returns the string value "199.98". Additionally, each wrapper class, like all other classes, has a nonstatic version of toString. For example, if we define n as follows:

```
Integer n = new Integer(198);
```

then n.toString() returns the string "198".
**Note: Wrapper classes**

Every primitive type has a wrapper class. Wrapper classes allow you to represent values of a primitive type as a class type. They also contain a number of useful predefined constants and methods.

**B.104** Character is the wrapper class for the primitive type char. The following piece of code illustrates some of the basic methods for this class:

```java
Character c1 = new Character('a');
Character c2 = new Character('A');
if (c1.equals(c2))
    System.out.println(c1.charValue() + " is the same as " + c2.charValue());
else
    System.out.println(c1.charValue() + " is not the same as " + c2.charValue());
```

This code displays

```
a is not the same as A
```

The `equals` method checks for equality of characters, so uppercase and lowercase letters are considered different.

Some of the static methods in the class `Character` follow:

```java
public static char toLowerCase(char ch)
Returns the lowercase equivalent of ch, if ch is a letter; otherwise returns ch.
Examples:
Character.toLowerCase('a') returns 'a'
Character.toLowerCase('A') returns 'a'
Character.toLowerCase('5') returns '5'
```

```java
public static char toUpperCase(char ch)
Returns the uppercase equivalent of ch, if ch is a letter; otherwise returns ch.
Examples:
Character.toUpperCase('a') returns 'A'
Character.toUpperCase('A') returns 'A'
Character.toUpperCase('5') returns '5'
```

```java
public static boolean isLowerCase(char ch)
Returns true if ch is a lowercase letter.
Examples:
Character.isLowerCase('a') returns true
Character.isLowerCase('A') returns false
Character.isLowerCase('5') returns false
```

```java
public static boolean isUpperCase(char ch)
Returns true if ch is an uppercase letter.
Examples:
Character.isUpperCase('a') returns false
Character.isUpperCase('A') returns true
Character.isUpperCase('5') returns false
```
```java
public static boolean isLetter(char ch)
    Returns true if ch is a letter.

Examples:
    Character.isLetter('a') returns true
    Character.isLetter('A') returns true
    Character.isLetter('5') returns false

public static boolean isDigit(char ch)
    Returns true if ch is a digit.

Examples:
    Character.isDigit('a') returns false
    Character.isDigit('A') returns false
    Character.isDigit('5') returns true

public static boolean isLetterOrDigit(char ch)
    Returns true if ch is either a letter or a digit.

Examples:
    Character.isLetterOrDigit('a') returns true
    Character.isLetterOrDigit('A') returns true
    Character.isLetterOrDigit('5') returns true
    Character.isLetterOrDigit('%') returns false

public static boolean isWhitespace(char ch)
    Returns true if ch is a white-space character.

Examples:
    Character.isWhitespace('a') returns false
    Character.isWhitespace(' ') returns true
```

**B.105** Java also has a wrapper class `Boolean`. This class has the two constants `Boolean.TRUE` and `Boolean.FALSE`. However, the Java reserved words `true` and `false` are much easier to use for these constants. So the constants in the class `Boolean` will not be of much help to us. The methods of the class `Boolean` are also not used very often. Although the class `Boolean` is not useless, it will be of little use to us in this text and we will discuss it no further.