APPENDIX

Basic Java Syntax

A.1 Introduction
A.2 The Structure of a Java Program
A.3 Java Data Types
   A.3.1 Primitive Types
   A.3.2 Reference Types
   A.3.3 Identifiers, Variable Declarations, and Scope
A.4 Expressions
A.5 Java Statements
   A.5.1 Assignment Statement
   A.5.2 Conditional Statements
      A.5.2.1 The if/else Statement
      A.5.2.2 The switch Statement
   A.5.3 Simple Input/Output
   A.5.4 Iterative Statements
      A.5.4.1 The while Statement
      A.5.4.2 The do/while Statement
      A.5.4.3 The for Statement
A.1 INTRODUCTION

In this text, we assume that the reader has completed an introductory programming course and is familiar with elementary programming concepts such as declarations, scope, assignment, conditionals, iteration, procedures, and parameters. However, we realize that the language covered in a first course varies from one school to another, and it may not have been Java, the language used throughout this book. To provide support for students with knowledge of an alternative programming language such as C++, Visual Basic, or Scheme, we have included this Appendix to review the basic elements of the Java language. It introduces the overall structure of a Java program and summarizes its basic data types, declarations, and statements. This material is expanded in the main body of the text with discussions of advanced features of the language. This includes the object-oriented capabilities of Java (Chapters 2 and 3), the Java Collection Framework (Chapter 9), exceptions (Chapter 10), streams (Chapter 10), threads (Chapter 11), the AWT and Swing graphical user interface libraries (Chapter 12), and networking (Chapter 13).

This Appendix is not a replacement for the in-depth coverage of the language that would be found in a typical introductory Java text. For students who want more detailed information and more guidance and assistance in learning the language, we suggest that you purchase and read one of the many good introductory Java programming books available in the marketplace to supplement the information presented in the following pages.

A.2 THE STRUCTURE OF A JAVA PROGRAM

In its most basic form, a Java program consists of a class declaration that defines a single method named main(). The method main() contains the code that is executed when the program is run. A Java class declaration is placed in a source file. The name of the source file and the name of the class that it contains must be the same. By convention, files that contain Java source code use the suffix .java. The general structure of a Java program is shown in Figure A.1.

A typical Java program will use several different classes as it executes. Some of these classes will come from prewritten classes libraries, and other classes will be ones that you write yourself. In this Appendix, we only discuss how to write a single top-level class that can be used to build simple Java programs. However, this single class is permitted to use all of the classes found in the Java class library that is part of every standard Java development kit. Chapters 2 and 3 explain in detail how to write complex object-oriented programs that are composed of many classes.

The Java class library that is included with Java contains hundreds of classes that can perform many useful operations. These classes are organized into about 135 packages. A package is nothing more than a collection of related classes. Table A.1 lists a few of the packages that are part of the Java API.
/* Here is an example of the general structure 
of a Java program */
import packageName;
public class className {
    public static void main( String args[] ) {
        // Your program goes here
    }
}

FIGURE A.1 General structure of a Java program

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang</td>
<td>Contains classes that implement many of the fundamental concepts of the Java language. Almost every Java program will use a class from this package.</td>
</tr>
<tr>
<td>java.math</td>
<td>Contains classes that provide infinite precision (i.e., big) numbers and other useful numerical routines.</td>
</tr>
<tr>
<td>java.util</td>
<td>Contains several utility classes that are useful in most Java programs. For example, this package contains the Date class that represents dates and time in a Java program.</td>
</tr>
</tbody>
</table>

To use a class that has been placed into a package, you use an import statement, as shown in Figure A.1. The import statements must appear at the very top of the program. (Note: The package java.lang is considered so essential that it is automatically imported into every Java program. It is the one package that does not need an explicit import statement.) The import statement allows you to access either a single class or a set of classes within a given package. For example, to use the Date class located in the package java.util, we would write:

    import java.util.Date;     // allow access to class Date in
    // package java.util

To access all of the classes in the java.util package, we use the following shorthand notation:

    import java.util.*;  // allow access to every class in java.util

Note, however, when using the * notation, you gain access to every class inside this package but not to the contents of any other packages that are nested inside it. Although the * notation is a convenient way to import all of the classes in a package, it is considered good practice to explicitly import each class that will be used in a program. This way it is possible, just by looking at the import statements, to determine which classes in a package are actually being used by the program.
In addition to providing you with access to the public classes inside a package, the `import` statement also allows you to refer to a class by its class name alone rather than by its fully qualified name. In general, when referring to a class, you must use its **fully qualified name**, which is:

```
packageName.ClassName
```

So, for example, the fully qualified name of the `Date` class in the `java.util` package is `java.util.Date`. However, the `import` statement allows you to refer to the class solely by its class name, which is `Date`.

Another part of the Java programming language that can be seen in the program of Figure A.1 is the format of a comment. Java supports two types of comments. The first, a **multiline comment**, begins with the characters `/*` and ends with `*/`. Anything between the starting `/*` and the terminating `*/` is ignored by the compiler. The second type of comment, a **single-line comment**, starts with the characters `//` and terminates at the end of the line. Everything on the line following the `//` is ignored by the compiler. Comments do not nest, which means that the characters `/*` and `//` have no special meaning inside a comment. The program in Figure A.2 illustrates comments in a Java program.

```
/**
 * This program will print the message "Enjoy reading this book!!"
 * when executed by the JVM.
 */

public class MyFirstProgram {

    public static void main(String args[]) {

        // This statement will print the message
        System.out.println("Enjoy reading this book!!");
    }

} // MyFirstProgram
```

**Figure A.2 Java program with comments**

To "execute" a class from the command line, the class must define a method called `main()` that is defined as:

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

The **Java Virtual Machine (JVM)** starts the execution of a Java program by invoking the `main()` method contained in the class specified on the command line. The program begins execution with the first executable statement in `main()`. The program terminates when the last statement in `main()` is executed. If the file named on the command line does not contain a method called `main()`, the JVM will flag this as an error and not run the program.
For additional information about how to compile and run Java programs, you will have to refer to documentation provided by your instructor, as this will depend on exactly which compiler and hardware are in use at your installation.

### A.3 JAVA DATA TYPES

Java, like most programming languages, allows a programmer to define a variable that refers to a memory location within the computer. Once a variable has been defined, the programmer can store and retrieve values to and from this memory location. An **identifier** is a name used to identify a variable, class, method, or symbolic constant in a program. The rules for forming identifiers are the same, regardless of how the identifier will be used:

- It may contain uppercase letters (A-Z), lowercase letters (a-z), digits (0-9), the underscore (_), and the dollar sign ($).
- It may not start with a digit. (You should also avoid starting an identifier with either an underscore or a dollar sign. These names are often used by the system for special purposes, and they can sometimes cause strange problems.)
- It can be any length, although to be useful the name should fit on a single line.
- It is case sensitive, so ABC, Abc, AbC, and abc are all different names.
- It cannot be equal to any of the 59 reserved words listed in Figure A.3.

<table>
<thead>
<tr>
<th>abstract</th>
<th>default</th>
<th>goto</th>
<th>operator</th>
<th>synchronized</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>do</td>
<td>if</td>
<td>outer</td>
<td>this</td>
</tr>
<tr>
<td>break</td>
<td>double</td>
<td>implements</td>
<td>package</td>
<td>throw</td>
</tr>
<tr>
<td>byte</td>
<td>else</td>
<td>import</td>
<td>private</td>
<td>throws</td>
</tr>
<tr>
<td>byvalue</td>
<td>extends</td>
<td>inner</td>
<td>protected</td>
<td>transient</td>
</tr>
<tr>
<td>case</td>
<td>false</td>
<td>instanceof</td>
<td>public</td>
<td>true</td>
</tr>
<tr>
<td>cast</td>
<td>final</td>
<td>int</td>
<td>rest</td>
<td>try</td>
</tr>
<tr>
<td>catch</td>
<td>finally</td>
<td>interface</td>
<td>return</td>
<td>var</td>
</tr>
<tr>
<td>char</td>
<td>float</td>
<td>long</td>
<td>short</td>
<td>void</td>
</tr>
<tr>
<td>class</td>
<td>for</td>
<td>native</td>
<td>static</td>
<td>volatile</td>
</tr>
<tr>
<td>const</td>
<td>future</td>
<td>new</td>
<td>super</td>
<td>while</td>
</tr>
<tr>
<td>continue</td>
<td>generic</td>
<td>null</td>
<td>switch</td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE A.3** Reserved words in Java

In addition to these restrictions, there are long-established stylistic guidelines followed by Java programmers when selecting identifiers. Although not enforced by the compiler, it is considered good practice to follow them to enhance the readability of your code.

- Class names begin with a capital letter, whereas variable and method names begin with lowercase letters. So, *Mathematics* is fine for a class name, but *Sort* and *Compare* would be used for the names of methods in this class.
- When names are composed of two or more separate words strung together, you capitalize the first letter of each of the words, except possibly for the first word
to adhere to the previous rule. So, the name MyFirstExample would be fine for a class name, and sortByKey would be a good method name.

* Choose a name that is highly indicative of the function of this entity, such as setRate or getAccelerationValue. Meaningless names like x32 or zzz should be avoided.
* Use all capital letters for constants and separate the words in an identifier with an underscore, such as PI and FIXED_RATE.

Every variable in a Java program is identified by a unique identifier and has a data type associated with it. The data type of a variable determines the kind of values that can be stored in the variable. A declaration statement defines a variable and specifies the identifier and data type of the variable. Every variable in a Java program must be declared and initialized before it can be used. The two formats for a variable declaration follow (the symbols [ ] indicate that a particular construct is optional and may be omitted).

```
[ final ] data-type identifier [ = expression ];
[ final ] data-type identifier-list;
```

where identifier-list is a list of valid identifiers separated by commas. Here are some valid variable declarations:

```java
int count = 0;
float root, xValue, yValue;
double temperature, humidity;
boolean done = false;
```

You may add the optional reserved word final before any declaration. This means that the identifier can be initialized once but never changed. Essentially, this makes it a constant, and any attempt to assign it a new value will be flagged as a fatal error. The word final is an example of a Java modifier that is used to set the characteristics of a variable. There are a number of other modifiers that can be added to a Java declaration, including public, private, protected, and static. These modifiers are discussed and explained in detail in Chapter 3. The use of the final modifier to define symbolic constants in a Java program is shown in the following examples:

```java
final double PI = 3.1415927;
final double FREEZING_POINT = 32.0;
final int MONTHS_PER_YEAR = 12;
```

There are two quite different categories of data types in Java: primitive types and reference types. We describe each of these two types in the following sections.

**A.3.1 Primitive Types**

The primitive types have their value stored directly in a variable. For example, given the declaration

```java
int x = 5;
```
the primitive value will be stored in memory as shown:

```
  x
\^ 5
```

In Java, there are eight primitive types, and they are summarized in Table A.2. Each of these primitive types has a fixed size (in bits), regardless of the platform on which the program is being run. Note that primitive types in Java are used for only the simplest and most basic of data values: integers, reals, characters, and Boolean. These are the types that are often implemented directly in hardware by most processors.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Has two values, true and false.</td>
</tr>
<tr>
<td>byte</td>
<td>8-bit signed two's complement integers, range: −128 to 127</td>
</tr>
<tr>
<td>short</td>
<td>16-bit signed two's complement integers, range: −32768 to 32767</td>
</tr>
<tr>
<td>int</td>
<td>32-bit signed two's complement integers, range: −2147483648 to 2147483647</td>
</tr>
<tr>
<td>long</td>
<td>64-bit signed two's complement integers, range: −9223372036854775808 to 9223372036854775807</td>
</tr>
<tr>
<td>char</td>
<td>16-bit unsigned values from 0 to 65535, representing Unicode characters</td>
</tr>
<tr>
<td>float</td>
<td>Single-precision, 32-bit format IEEE 754 floating-point values, range: 1.40239846e−45 to 3.40282347e +38</td>
</tr>
<tr>
<td>double</td>
<td>Double-precision, 64-bit format IEEE 754 floating-point values, range: 4.9406564581246544e−324 to 1.79769313486231570e+308</td>
</tr>
</tbody>
</table>

There are special floating-point values: positive infinity, negative infinity, and not a number (NaN).

Primitive literals refer to specific primitive values in a Java program such as +4 or −7.3. Although we present a number of rules in this section that describe how to construct valid literals, the basic rule of thumb you should follow is that literals in Java are written in the same way that you would write numbers on a piece of paper. For example, a series of digits with no decimal point is an integer, whereas a series of digits with a decimal point is of type double. Character literals consist of a single character surrounded by single quotation marks. The Boolean literals are the words true and false. Table A.3 lists all of the literal values recognized in the Java language.

Note that decimal integer literals use the standard 32-bit format, unless they end with the letter L or l, in which case they are stored in the long, 64-bit format. Alternatively, integers can be expressed in hexadecimal (base-16) and octal (base-8) notation. An integer that begins with the symbols 0x is interpreted as a hexadecimal number, and an integer that begins with the character 0 is interpreted as octal.

Floating-point constants must have at least one digit to the right of the decimal point, but it is optional whether or not to include digits to the left. All floating-point literals are stored as 64-bit double values by default. To represent a number as a
<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte, short, int</td>
<td>0, 123, -456, 55665, ...</td>
</tr>
<tr>
<td>long</td>
<td>0L, 123L, -456L, ...</td>
</tr>
<tr>
<td>char</td>
<td>'a', 'A', 'b', 'B', ... 'z', 'Z'</td>
</tr>
<tr>
<td></td>
<td>'0', '1', '2', ... '9'</td>
</tr>
<tr>
<td></td>
<td>'l', '@', '#', ...</td>
</tr>
<tr>
<td></td>
<td>'\b', '\f', '\n', '\r', '\t', '\v', ''</td>
</tr>
<tr>
<td>float</td>
<td>1.2345f, 1234.423f, 0.1f, -1.23f, ...</td>
</tr>
<tr>
<td></td>
<td>1e10f, 1.234e-10f, 3.456e2f, -1.2345e12f, ...</td>
</tr>
<tr>
<td>double</td>
<td>1.2345, 1234.423, 0.1, -1.23, ...</td>
</tr>
<tr>
<td></td>
<td>1e10, 1.234e-10, 3.456e2, -1.2345e12, ...</td>
</tr>
<tr>
<td>boolean</td>
<td>true, false</td>
</tr>
<tr>
<td>String</td>
<td>&quot;This is a String&quot;, &quot;Hello World\n&quot;</td>
</tr>
<tr>
<td>Object</td>
<td>null</td>
</tr>
</tbody>
</table>

32-bit single-precision float value, you follow the literal by the character F or f. Finally, real numbers can be written in scientific notation using the letter E or e to represent the phrase “times 10 to the power of ...”.

Java will automatically convert one primitive type into another whenever it can be sure that no information will be lost in the conversion. This would be done, for example, to convert an int to a double or a byte to an int. This is called a widening conversion. However, Java will not do this when the conversion is to a smaller data type and information might be lost. This could happen, for example, if we were to convert a float to an int or a short to a byte. This is called a narrowing conversion, and the Java compiler will flag this as a fatal error. However, if you still wish to force Java to carry out this conversion, you can do so using a casting operation. This is done by writing in parentheses, before the expression to be converted, the name of the type into which you wish the value to be cast. For example, if x is an int and we wish to convert it to a byte, we can write:

```
(byte)x
```

Java will do the cast, but you should realize that if \( x > +127 \) or \( x < -128 \), the maximum and minimum values that can be represented in a byte, then this conversion will produce an incorrect value. Narrowing conversions should be done with the utmost of caution.

In Java, characters are stored using the 16-bit Unicode set. To indicate a character, it is placed inside single quotation marks—for example, 'A'. There are also special escape characters to represent nonprinting control characters, such as:
A.3.2 Reference Types

Reference types are used for every data value in Java other than the eight primitive
types listed in Table A.2. With a reference type, the values associated with object \( x \) are
not stored directly in memory location \( x \). Instead, location \( x \) holds a reference to that
object—that is, a pointer to where in memory this object will be found (Figure A.4).

```java
int p1; // p1 and p2 are primitive types
int p2;
Cat c1; // c1 and c2 are reference types because the type
Cat c2; // Cat is not one of the 8 primitive types
p1 = 123;
p2 = 456;
c1 = new Cat( "Grumpy Cat" );
c2 = new Cat( "Fat Cat" );
```

![Figure A.4 Reference types in Java](image)

The eight primitive types in Table A.2 can be converted into reference types using
wrapper classes that have exactly the same name as the primitive type. These
wrapper classes are all found in the package java.lang. For example, to convert the
primitive integer value \( x \) into a reference type, we use the class Integer. To convert
the double value \( y \) into a reference, we use the class Double. This capability is impor-
tant, as there are a number of places in Java where only reference types are permitted. You
will see how to do this in the following section.

The last data type we discuss in this section is an array. An array is a structure
that holds multiple values of the same type. In Java, arrays are considered reference
types, and they behave slightly differently from array types in languages like C or
C++. A declaration for an array specifies the name that will identify the array and
the type of elements that will be stored in the array as shown next. The square brack-
ets in the declaration indicate that \( x \) refers to an array.
int x[];

This declaration statement allocates storage for a reference to an array; it does not actually create the array itself. An array is created using operator new as follows:

\[
\text{x = new int[10];}
\]

The type specifies the type of the elements that will be stored in the array, and the number inside the square brackets indicates how many elements can be stored. The elements in the array will automatically be initialized to the appropriate initial value for its type: 0 for integers, 0.0 for reals, the null character for characters, and null for references.

The size of an array is fixed and cannot be changed. Given a variable that contains a reference to an array, it is possible to determine the size of an array using the length operator:

\[
\text{int size = x.length;}
\]

Elements within the array are accessed using the following syntax:

\[
\text{x[0] = 10; // Set element x[0] to 10}
\]
\[
\text{int z = x[1]; // Set z to element x[1]}
\]

In Java, array indexes are 0-based, which means they start at 0 and run to 1 less than the length of the array. For example, the index of the first element in the 10-element array \(x\) would be 0 and the last index would be 9. At run time, the JVM checks all array accesses and verifies that the indexes are valid. If an attempt is made to use an invalid index (i.e., a negative index or one greater than the length of the array), an exception is raised, which may cause the program to terminate.

## A.3.3 Identifiers, Variable Declarations, and Scope

Every variable and constant declaration has a scope—the region of the program where it is visible to the programmer and can be used. A variable is said to have class scope if it is declared within a class but outside the methods contained within this class. An identifier with class scope can be accessed by any of the methods in the class that follow its declaration. An example of class scope is shown in Figure A.5. The constant \(\text{PI}\) can be accessed by any method that follows its declaration, specifically \(\text{areaOfACircle()}\) and \(\text{circumferenceOfACircle()}\).

```java
public class ClassScopeExample {
    final double PI = 3.14; // this has class scope
    public double areaOfACircle() {
        // ...
    }
    public double circumferenceOfACircle() {
        // ...
    }
}
```

**Figure A.5** Example of class scope
An identifier is said to have block scope if it is declared within a block. A block is a construct that begins with a ( and ends with a ). The code inside a Java method is an example of a block. The scope of a variable declared inside a block is from the point where the declaration is made to the end of that block. For example, Figure A.6 expands the example in Figure A.5 to show a variable called area with block scope.

```java
public class BlockScopeExample {
    final double PI = 3.14; // this has class scope
    public double areaOfACircle(double radius) {
        double area; // this has block scope
        area = PI * (radius * radius);
        return area;
        // and its scope ends here
    }
    public double circumferenceOfACircle() { ... };
}
```

**FIGURE A.6 Example of block scope**

The declaration `double area` on the second line of `areaOfACircle` is declared inside a block that begins with the immediately preceding ( This variable has a scope that extends from the point of the declaration to the ) character that occurs four lines later.

When a primitive type, such as an int or float, is declared, the JVM allocates enough memory to store the specified value and initializes its value either to a default value or to an initial value provided by the user. However, when a reference variable is declared, something quite different happens. We said that a reference variable points to or refers to an object. However, the object to which it refers has not yet been created. For example, if `Circle` is a class, then the declaration

```java
Circle x;
```

creates a reference variable `x` that can point to objects of type `Circle`, but right now it does not refer to anything. After making this declaration, the variable `x` looks like this:

```
  x  null
```

To create an object for `x` to refer to, we use the `new` operator in Java. This operator allocates memory for a new object of the type that `x` refers to (`Circle`, in this case) and returns a reference (i.e., a pointer or memory address) that we can store into the reference variable. The methods that create new objects are called constructors, and they have exactly the same name as the class to which they belong. So, for example, to create a new `Circle` object, we would use a constructor called `Circle`. A constructor may have zero, one, or more parameters.
Here is an example of how to use a constructor with no parameters, called the default constructor, to initialize \( x \) that was created using the preceding declaration:

\[
x = \text{new Circle();} \quad // \text{assume no parameters}
\]

This creates a new Circle object and has it referenced by the variable \( x \):

![Diagram of a Circle object and variable reference]

It is also possible to combine into a single step the declaration of a reference variable and its initialization as follows:

\[
\text{Circle } x = \text{new Circle();}
\]

This is also the way to convert a primitive type, such as an int or double, into a reference type. We use the constructor from the corresponding wrapper class, passing it the primitive type as its only parameter. What we will get back is the primitive type now represented as an object. For example:

\[
\begin{align*}
\text{int } x; & \quad // x \text{ is a primitive object} \\
\text{double } y; & \quad // \text{so is } y \\
\text{Integer } x\text{AsAnObject;} & \quad // x \text{ as a reference value} \\
\text{Double } y\text{AsAnObject; } & \quad // y \text{ as a reference value}
\end{align*}
\]

\[
\begin{align*}
x\text{AsAnObject } &= \text{new Integer}(x); \\
y\text{AsAnObject } &= \text{new Double}(y);
\end{align*}
\]

When a variable goes "out of scope," the memory allocated to the variable is returned to the heap, the area of memory used for dynamic allocation of space to variables. To keep track of when objects are no longer needed and the memory allocated to them can be returned to the heap, the JVM keeps track of the number of references to every object in memory. Whenever a new reference to an object is created, the reference count of the object is incremented by 1. When a reference to an object is removed, the reference count of the object is decremented by 1. When the reference count of an object reaches 0, the object can no longer be accessed, and it is called "garbage." Periodically, Java invokes a garbage collector that collects all the objects that can no longer be accessed and returns the memory allocated to those objects to the heap.
A.4 EXPRESSIONS

Java expressions are composed of variables, constants, operators, method invocations, and the grouping symbols ( and ). All variables and constants in the expression must have been declared, initialized, and in scope; otherwise, the compiler will flag this as an error. In addition, the user must be careful to use values of consistent data type within the expression. If you use values of different type within a single expression, the compiler must be able to convert them, either automatically or via a user-specified cast, into a single data type so that they can be operated on in a consistent manner. For example, if i is an int and f is a float, then the expression \((i + f)\) mixes values of different types. This is called a mixed mode expression, and Java must deal with it in a reasonable way. In this example, the compiler will automatically perform a widening conversion on the integer \(i\) and convert it to a float so that the addition can be performed using real numbers. We could also write this expression in either of the following two ways:

\[
\text{(float)} \ i + f \quad // \text{behaves the same as not doing a cast} \\
i + (\text{int}) f \quad // \text{this is a narrowing conversion.}
\]

Note that the second example may cause the loss of information because we are discarding the fractional part of \(f\).

Some mixed mode expressions cannot be rectified. For example, if \(b\) is a boolean and \(f\) is a float, then the expression \((b + f)\) makes no sense because Boolean values cannot be cast into real numbers, or vice versa. In this case, the compiler will produce an error message. The bottom line about writing expressions is to be very careful of the data types you use within the expression and use only a single type or a set of types that can be meaningfully cast into a single type.

The arithmetic operators in Java are listed in order by precedence in Table A.4. The operators with the highest precedence are listed first, followed by the operators with lower precedence. All of the operators with the same description have equal precedence.

You should be familiar with most of the operators in Table A.4 because they are, for the most part, quite similar to what is found in a number of other programming languages. The ones that you may not have seen before are:

\[
a \% b \quad \text{remainder} \quad \text{The remainder left after doing an integer division of } \ a \text{ by } b. \text{ For example, } 9 \% 5 \text{ is 4.}
\]

\[
a++ \quad \text{postincrement} \quad \text{Use the value of } a \text{ to evaluate the expression and, when that is completed, increment } a \text{ by 1. If } a \text{ currently has the value 4, then } (a++) * 2 \text{ will evaluate to 8, and } a \text{ will be 5.}
\]

\[
++a \quad \text{preincrement} \quad \text{First increment } a \text{ and then use that new value to evaluate the expression. If } a \text{ currently has the value 4, then } (+a) * 2 \text{ will evaluate to 10, and } a \text{ will be 5.}
\]

(Note: The pre- and postdecrement work in a similar fashion but subtract 1.)
<table>
<thead>
<tr>
<th>Description</th>
<th>Syntax</th>
<th>Meaning</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary postfix</td>
<td>[]</td>
<td>Array access</td>
<td>Left to right</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>Object member access</td>
<td></td>
</tr>
<tr>
<td></td>
<td>()</td>
<td>Method invocation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>++</td>
<td>Postincrement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Postdecrement</td>
<td></td>
</tr>
<tr>
<td>Unary prefix</td>
<td>++</td>
<td>Preincrement</td>
<td>Right to left</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Predecrement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>Unary plus</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Negation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>~</td>
<td>Bitwise complement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>!</td>
<td>Logical NOT</td>
<td></td>
</tr>
<tr>
<td>Object creation</td>
<td>new</td>
<td>Object creation</td>
<td>Right to left</td>
</tr>
<tr>
<td>and Cast</td>
<td>(type)</td>
<td>Cast</td>
<td></td>
</tr>
<tr>
<td>Multiplicative</td>
<td>*</td>
<td>Multiplication</td>
<td>Left to right</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>Division</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>Remainder</td>
<td></td>
</tr>
<tr>
<td>Additive</td>
<td>+</td>
<td>Addition</td>
<td>Left to right</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Subtraction</td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt;</td>
<td>Left shift</td>
<td>Left to right</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;</td>
<td>Right shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;&gt;&gt;</td>
<td>Right shift with zero fill (unsigned shift)</td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&lt;</td>
<td>Less than</td>
<td>Left to right</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>Less than or equal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>Greater than</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>Greater than or equal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>instanceof</td>
<td>Instance of</td>
<td></td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>Equal</td>
<td>Left to right</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>Not equal</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>Left to right</td>
</tr>
<tr>
<td>XOR</td>
<td>^</td>
<td>Bitwise exclusive OR (XOR)</td>
<td>Left to right</td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td></td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>Logical AND</td>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>Left to right</td>
</tr>
<tr>
<td>Logical OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td>?</td>
<td>Ternary conditional operator</td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>=</td>
<td>Assignment</td>
<td>Right to left</td>
</tr>
<tr>
<td></td>
<td>+=</td>
<td>Assignment combined with an operator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-=</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*=</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/=</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>%=</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;=</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;=</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;&gt;=</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;=</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>^=</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=</td>
<td></td>
</tr>
</tbody>
</table>
a \&\& b \quad \text{Boolean AND} \quad \&\& \text{ is the standard Boolean AND with which you are familiar. The operator } \&\& \text{ is a bitwise AND, where each bit of } a \text{ and } b \text{ is separately ANDeD together.}

a \mid\mid b \quad \text{Boolean OR} \quad \mid\mid \text{ The standard Boolean OR. The operator } \mid \text{ is a bitwise OR, where each bit of } a \text{ and } b \text{ is ORed together.}

Since Java includes a Boolean data type, the Boolean operators like && and || will only work with Boolean types. This is different from programming languages like C or C++ that use integer values to represent Boolean values.

The Boolean operators perform what is called short-circuit evaluation. For example, when evaluating an expression such as \((a \&\& b \&\& c \&\& d)\), the JVM will evaluate each of the Boolean variables one at a time starting from the left. As soon as one of the Boolean expressions evaluates to \text{false}, the evaluation stops since the overall value of the entire expression must be \text{false}. This means that some of the variables in the expression may never be evaluated (Boolean OR works in a similar fashion). It is also possible to use the bitwise AND and bitwise OR operators with Boolean types. These operators never short circuit, which means all of the variables in the expression will be evaluated.

The precedence level of an operator determines the order in which operations are carried out in the absence of grouping symbols that change the order. The rule is that, in the absence of parentheses, an operator with a higher precedence is always done before an operation of lower precedence. The symbol pair ( ) changes precedence because anything inside parentheses is done first. For example, the expression

\[ a++ + b > --c \]

uses the following four arithmetic operators:

\[ + \quad \text{postincrement} \]
\[ + \quad \text{addition} \]
\[ > \quad \text{greater than} \]
\[ -- \quad \text{predecrement} \]

Since we do operations in strict order of their precedence level, we can determine from Table A.4 that this expression will be evaluated as if it had been written as:

\[ ((a++) + b) > (--c) \]

One of the characteristics of a well-written program is that it is easy to read. Since it is unrealistic to expect a reader of your program to be familiar with all of the precedence levels in Java, you should use the grouping symbols \{ and \} whenever required to make your intentions clear. Not only will your program be easier to read, but you are likely to prevent yourself from making errors. Therefore, when writing a complex expression, be sure to use parentheses whether or not they change the actual order of computation. This will make your program far more readable.

The associativity value given in Table A.4 specifies how operators of the same precedence level will group. For example, the subtraction operator groups left to
right. Therefore, the expression $a - b - c$ is interpreted as $(a - b) - c$. The assignment operator groups right to left, so the expression $a = b = c$ is evaluated as $a = (b = c)$.

Here are some examples of arithmetic expressions and their Java equivalents. The examples assume that all variables have been correctly declared and properly initialized (all expressions are assumed to use real values):

<table>
<thead>
<tr>
<th>Arithmetic Expression</th>
<th>Java Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{\text{sum}}{\text{number} + 1}$</td>
<td><code>sum / (number + 1.0)</code></td>
</tr>
</tbody>
</table>
| $\frac{a + b c^4}{2}$ | $(a / 2.0) + (b * Math.pow(c, 4.0));$  
(Investigating `Math.pow()` is in class `java.lang.Math`) |
| $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$ | $(-b + Math.sqrt(b * b) - (4.0 * a * c)) / (2.0 * a);$  
(Investigating `Math.sqrt()` is in class `java.lang.Math`) |

### A.5 Java Statements

#### A.5.1 Assignment Statement

The Java assignment behaves in much the same fashion as the assignment statement in other imperative languages such as C, C++, and BASIC. The syntax of the statement is

```java
variable = expression;
```

where `variable` is the name of a declared variable currently in scope, `=` is the assignment operator, and `expression` is any valid Java expression formed according to the rules described in Section A.4. The behavior of this statement is exactly what you would expect: The expression on the right side of the assignment operator is evaluated, and the resulting value is stored in the variable on the left side. The previous value stored in this variable is lost.

Assignment statements themselves have a value. That is, the statement $a = b + c$ will not only compute the sum of $b$ and $c$ and store that value in $a$, but it also evaluates to the value that is stored into $a$. Since an assignment statement produces a result, you can treat it as if it were an expression and use the result of that expression in other computations. For example:

```java
a = b = c + d;
```

You can see in Table A.4 that the assignment operator groups right to left. So, the computer will begin by evaluating $b = c + d$, which computes the sum $c + d$ and stores the result in $b$. Now the next assignment expression is evaluated, which causes the variable $a$ to be assigned the value of the assignment statement $b = c + d$ or the value that was stored in $b$. The end result is that the sum $c + d$ is placed into both a
and b. This is a good way to initialize a group of variables to the same value with a single statement:

\[ a = b = c = 0; \]  // initialize a, b, and c to 0.

Here are two examples of Java assignment statements:

\[
\text{Algebraic Representation} \quad \text{As a Java Assignment Statement}
\]

a. \[
C = \frac{5(F-32)}{9}
\]
\[
\text{double centigrade, fahrenheit;}
\]
\[
\text{centigrade} = (5.0*(\text{fahrenheit}-32.0)) / 9.0;
\]

b. \[
\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]
\[
\text{double a, b, c; // the three coefficients}
\]
\[
\text{double root1, root2; // the two real roots}
\]
\[
\text{root1} =
\]
\[
(-b + \text{Math.sqrt}((b*b)-(4.0*a*c)))/(2.0*a);
\]
\[
\text{root2} =
\]
\[
(-b - \text{Math.sqrt}((b*b)-(4.0*a*c)))/(2.0*a);
\]

In example b, we used the Java method Math.sqrt(), which is found in class Math in package java.lang. However, as mentioned earlier, it is not necessary to import any classes in the java.lang package because they are automatically imported into every program you write. It would not be wrong, though, to include the following:

```java
import java.lang.Math;
```

Finally, a common statement in Java is an assignment that looks like the following:

\[ a = a + b; \]

in which you take a value a, perform some arithmetic operation on it (in this case, adding b), and put the result back into a. Java provides variations of the assignment operator to make it easier to write this type of assignment statement. These variants all have the format \( \text{op} = \), where \( \text{op} \) is one of the following 11 operators: + - * / % & | ^ << >> >>>. The meaning of this construct is exactly the same as if you had written \( \text{var} = \text{var} \text{ op expr} \). Examples of these variations are:

<table>
<thead>
<tr>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a += b</td>
<td>a = a + b</td>
</tr>
<tr>
<td>a -= b</td>
<td>a = a - b</td>
</tr>
<tr>
<td>a *= b</td>
<td>a = a * b</td>
</tr>
</tbody>
</table>

\[ \text{A.5.2 Conditional Statements} \]

The conditional statements are the decision-making statements of Java. The two conditional statements in the language are the if/else and the switch. There is also a ternary conditional operator, ?:, that allows you to imbed a conditional test
inside a Java expression. This conditional operator is written:

```java
boolean expression ? expression1 : expression2
```

The meaning of this is to first evaluate the boolean expression. If it is true, then evaluate expression1, and that is the result of the overall expression; if the boolean expression is false, then evaluate expression2 and that becomes the result of the overall expression. So, to find the bigger of a and b and store that value into c, we could write:

```java
c = a > b ? a : b
```

This operator is not used very often, and we will not say any more about it.

### A.5.2.1 The if/else Statement

This is the most important conditional statement in Java, and it is the one that is used the overwhelming majority of the time. The syntax of the `if/else` (where the `[]` mean that something is optional) is shown in Figure A.7.

```java
if (boolean expression)
    statement1;
[else
    statement2;
]
```

**Figure A.7** Syntax of the `if/else` statement

The meaning of this statement is to first evaluate the boolean expression. If it evaluates to true, then execute statement1 and skip statement2 (if it is present) and continue execution with the next statement. If boolean expression evaluates to false, then skip statement1 and execute statement2 if it is present. If the else clause has been omitted, then do nothing at all and continue with the next statement. Pictorially, we can view the `if/else` statement as shown in Figure A.8.

**Figure A.8** Behavior of the `if/else` statement
Note that the syntax of the $if/else$ explicitly allows only a single statement in either case, but we often want to do much more. To accomplish this, we must use the concept of a **compound statement**, also called a block. This is a group of declarations and/or statements that begins with a {, includes any number of declarations and/or statements in a row, and ends with a }. For example,

```java
{    int b = 53;  // example of a compound statement
    b = x * 45;
    int c = b - 101;
}
```

The rule is that anywhere in Java where only a single statement is allowed, you may place a compound statement. Thus, if you need to do more than a single thing in an $if/else$ (or anywhere else for that matter), this will not be a problem. Just surround the statements with { and }, and they will now function as if they were one statement.

In the last section, we showed how to write two assignment statements to compute the real roots of a quadratic equation. However, it is not possible to compute these roots if either $a = 0$, because the denominator of the formula will be 0, or if $b^2 - 4ac < 0$, because the roots will then be imaginary. We can handle this situation as follows:

```java
boolean errorFlag = false;
if ((a == 0) || ((b*b) - (4.0 * a * c)) < 0.0 ) {
    errorFlag = true;
}
else {
    root1=((-b + Math.sqrt((b*b)-(4.0*a*c)))/(2.0*a));
    root2=((-b - Math.sqrt((b*b)-(4.0*a*c)))/(2.0*a));
}
```

Note that we did not need to put curly braces around the assignment of the value true to `errorFlag` because it is only a single statement. The curly braces are needed, however, in the else clause to allow us to include the two assignment statements. If you accidentally omitted the curly braces in the else clause, then only the first assignment statement (root1 = ...) will be inside the else clause, and the second one (root2 = ...) will become the statement that follows the entire if/else. That is, leaving out the () will produce the following (we are formatting it to more clearly indicate the structure):

```java
    * boolean errorFlag = false;
    if ((a == 0) || ((b*b) - (4.0 * a * c)) < 0.0 ) {
        errorFlag = true;
    }
    else
        root1=(-b + Math.sqrt((b*b)-(4.0*a*c)))/(2.0*a);
    root2=(-b - Math.sqrt((b*b)-(4.0*a*c)))/(2.0*a);
```
If \( a == 0 \), you will correctly set the \texttt{errorFlag} to \texttt{true}, but then you will try to compute the value of \texttt{root2}, producing a fatal divide by zero error. Always remember to bracket \texttt{if} or \texttt{else} clauses with curly braces even if the clause contains a single statement. In fact, many coding standards encourage you always to place curly braces around the statements in an \texttt{if} or \texttt{else} clause regardless of the number of statements in the clause.

Since the \texttt{else} clause is optional in Java, you may get yourself into the following ambiguous situation, called the 	exttt{dangling else} problem:

```java
if (condition1)
    if (condition2)
        statement1;
    else
        statement2;
```

The ambiguity comes from not knowing if the \texttt{else} \texttt{statement2} clause goes with the \texttt{if (condition1)} test or the \texttt{if (condition2)} test. You cannot tell from the indentation alone because we could also have formatted these statements as:

```java
if (condition1)
    if (condition2)
        statement1;
else
    statement2;
```

The rule in Java is that an \texttt{else} clause is always associated with the nearest \texttt{if} statement. Thus, in the preceding situation, the statements will be interpreted as shown in the first example, not the second. That is, \texttt{statement2} is associated with the test \texttt{if (condition2)}.

Java is a free format language, and you are explicitly allowed to format a cluster of statements in whatever way you want, including the second example. Since indentation is transparent to the compiler and does not change the flow of control, you should always use indentation to show how statements are nested and to make your programs easier to read. (Many Java editors will do this for you automatically.) Grouping statements as shown in the second example would be highly confusing to a reader of your program. You are strongly encouraged to avoid this type of misleading indentation.

A conditional construct that is quite common is a series of nested \texttt{if/else} clauses that select exactly one of a blocks of code. For example, assume that \texttt{yearInSchool} is a variable that specifies a student's year in school. We could handle all the different possibilities as shown in Figure A.9.

There is nothing special or different about the block of statements in Figure A.9, as it is simply a series of nested \texttt{if/else} statements of the type just discussed. However, this pattern occurs quite often, and a familiarity with this particular statement sequence can be very useful.
if (yearInSchool == 1) {
    // handle freshman students
} else if (yearInSchool == 2) {
    // handle sophomore students
} else if (yearInSchool == 3) {
    // handle junior students
} else if (yearInSchool == 4) {
    // handle senior students
} else {
    // handle all other cases here
}

FIGURE A.9 Nested if/else statements

A.5.2.2 The switch Statement
There is another way to handle the sequence of decisions shown in Figure A.9. We can use the second type of Java conditional statement called the switch statement. The syntax of this statement is given in Figure A.10.

switch (expression) {
    case c1: // code block 1
        break;
    case c2: // code block 2
        break;
    case c3: // code block 3
        break;
    ...  // default: // default code block
}

FIGURE A.10 Syntax of the switch statement

* The switch statement is a multiway conditional. We first determine the value of the expression on the top line, which must evaluate to a value of type byte, char, short, or int. We then determine if this value is equal to any of the constant values c1, which are of the same type as the expression on the first line. If the value equals the constant c1, we execute code block 1. If the value equals constant c2, we execute code block 2 and so on. Finally, if the value does not equal any of these constant values, then we execute the default code block.

After finishing the indicated code block, the switch statement code continues with the next code block; that is, after doing code block 1, it would start executing
block 2, block 3, and so on. This is rarely what we want to happen. To indicate that after completing execution of a code block we wish to terminate the entire switch statement, we must use a special Java statement called break. Executing this statement will cause you to immediately exit the block in which the break statement is contained and continue execution with the first executable statement after the block. In this case, it will cause you to immediately exit the switch statement and continue with the statement that follows it.

Figure A.11 shows how we could use the switch to implement the year in school computations that were done in Figure A.9 using nested if/else statements.

```java
switch (yearInSchool) {
    case 1:  // handle freshman students
        break;
    case 2:  // handle sophomore students
        break;
    case 3:  // handle junior students
        break;
    case 4:  // handle senior students
        break;
    default: // handle all other conditions
}
```

**Figure A.11 Example of the switch statement**

A switch statement can have more than one case clause labeling a single code block. For example, let's say that juniors and seniors are handled using the same block of code. Then we could rewrite these two cases from Figure A.11 in the following way:

case 3: case 4:  // handle juniors and seniors
    break;

Every constant value in a switch statement must be unique. The following structure

case 1:  // code block 1
    break;

```
case 1:  // code block 2
```

is ambiguous. If the switch expression evaluates to a 1, we will not know whether to execute code block 1 or code block 2. The compiler does not allow this, and it will flag the condition as an error.

**A.5.3 Simple Input/Output**

The easiest way to output text in Java is with the print() and println() methods, both of which can be found in the class PrintWriter in the package java.io. These
two routines produce a character stream that is usually directed to the standard output stream. The syntax of these two methods is as follows:

```
System.out.print(expression1, expression2, ..., expression_n);
System.out.println(expression1, expression2, ..., expression_n);
```

These two routines behave in very similar ways: Each of the expressions, which can be of any type, is evaluated and converted to a character string. Then the individual strings are concatenated together, using the concatenation operator +, and the string that results is displayed on the standard output device, usually the screen. The only difference between the two methods is that, after the output has been displayed, the `println()` variant generates a carriage return. Thus, the following output operation will begin on the next line. With the `print()`, the next output operation begins from wherever this one left off.

For example, here is how we could display the two real roots of a quadratic equation, called `root1` and `root2`:

```
System.out.println("The two roots of the equation are
  +
  root1 + " and "+ root2");
```

If the equation has no real roots, here is how we might display an error message on the screen rather than setting an error flag to true as was done earlier:

```
System.out.println("*** ERROR. No real roots. ****");
System.out.println();
```

A `println()` command with no arguments generates a blank line, and it can be used to control the horizontal spacing of the output.

Basic character output is rather easy in Java using the `print()` and `println()` commands, but the same cannot be said about character-based keyboard input. Java was intended for use in a visual environment that supports a graphical user interface. Therefore, it does not have simple character-oriented keyboard input commands analogous to the `print()` and `println()` output methods just described.

Obviously, input is a fundamental part of most programs. Therefore, to solve the problem of providing basic input operations, most introductory Java textbooks include their own Java classes for reading primitive numerical and character data types from the keyboard. These classes typically include methods with names such as `readInt()`, `readDouble()`, and `readChar()` that allow a user to enter these primitive data types via the keyboard. Unfortunately, these classes are nonstandard, and a method used by students with one textbook will almost certainly not operate in exactly the same way as an input method provided by another text.

A second solution is to immediately learn how to build graphical user interfaces so that you can design and use GUIs for all program input. Although this is a wonderful solution, the time required to adequately cover this huge and complex topic could take significant amounts of time away from other critically important concepts. In this text, we discuss in detail how to design and build graphical user interfaces, but not until Chapter 12, after we have covered other important ideas, such as software design, object-oriented programming, and data structures.
A third way to support keyboard input is to write your own input methods using the stream classes included in the package java.io. We explain at length how to do this in the main body of the text, but again, not until much later (see Chapter 10).

Therefore, if you are currently unfamiliar with how to build a GUI and are not comfortable using the InputStream, InputStreamReader, and BufferedReader classes, then none of these solutions will work for you right now. In this case, you will either need to use a nonstandard keyboard input class provided by your instructor, or you will need to use the fourth method that we will mention: command-line arguments.

To start the execution of a Java program, you must provide the name of a public class that contains a method called main(). For example, given the program in Figure A.2, we could begin its execution with a command line that looks like the following, although the exact format will depend on which Java environment you are using:

```
java MyFirstProgram
```

In addition to the file name, this command line may also contain a series of command-line arguments, which are entered immediately following the file name. For example, here is how we might include three command-line arguments, an integer, a real, and a string:

```
java MyFirstProgram 1 2.3 file.data
```

These command-line arguments are passed into the program using the String parameter args that is the standard parameter for every main() method.

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

The first parameter on the command line (the 1 in this example) is passed in as args[0], the 2.3 is stored in args[1], and the string “file.data” will be located in args[2]. You can determine exactly how many arguments were entered on the command line using the length operation args.length. Once you have these parameters stored in a String variable, you can use the methods contained in the wrapper classes Integer, Float, Double, and so on to convert these string representations to data values of the proper type. For example, the method parseDouble() found in class Double will convert the string representation of a double into a numeric value of type double.

Suppose that we wanted to input the three coefficients of a quadratic equation on the command line in the following way:

```
java Quadratic 3.1 -1.2 4.0
```

This corresponds to the equation $3.1x^2 - 1.2x + 4.0 = 0$. Our program could use these inputs in the following way:

```
if (args.length != 3)
    System.out.println("Wrong number of parameters");
else {
```
double a, b, c;  // the 3 coefficients
a = Double.parseDouble(args[0]);
b = Double.parseDouble(args[1]);
c = Double.parseDouble(args[2]);

// now go ahead and solve the equation
// ax^2 + bx + c = 0

If needed, this technique can serve as a temporary way to do keyboard input until you become more comfortable with one of the more general and more flexible techniques—either GUIs or streams.

\* A.5.4 Iterative Statements

The most basic method of looping in Java is iteration: the repetition of a block of statements either a fixed number of times or until a specific condition occurs. There is another method, called recursion, to execute a block of statements repeatedly. Recursion is the invocation of a method by the method itself, and there are some languages (e.g., LISP and Scheme) in which recursion is the only method of implementing a loop. Recursion is explicitly allowed in Java, and if you are familiar and comfortable with it, go ahead and use it. In fact, we show numerous examples of recursive algorithms in the body of the text, such as the merge sort and quicksort examples in Chapter 5.

However, by far the most common method of looping in Java is iteration using one of the three iterative control constructs: while, do, and for. We will discuss these statements in the upcoming sections.

A.5.4.1 The while Statement

The while statement is the most common and widely used iterative statement in Java. The while statement repeats a block of statements, called the loop body, until a given Boolean condition become false. The syntax of the while statement is shown in Figure A.12(a), and its behavior is diagramed in Figure A.12(b).

The while statement is a type of looping construct called a pretest loop. In this type of loop, we initially test the Boolean condition to see if it is false. If so, we immediately exit the loop, and the statement is never executed. Thus, the loop body can be executed zero, one, or more times. Note that the syntax of the while permits only a single statement. However, as we mentioned in the previous section, that is not a problem. If you want the loop body to be longer than a single statement, then make it a compound statement by encasing the loop body within the characters {}.

As with any loop based on iterating until a condition becomes true (or false), it is essential that at least one statement in the loop body changes the value of a variable such that the Boolean expression will eventually become false. Otherwise, you have the unfortunately all-too-common programming error called an infinite loop. (Note: If you want to write a program that runs forever, or at least until the user does something to stop it, then it is acceptable to write while (true)... However, be very careful when using this construct.)
There are a number of other things to be careful of when using the while construct that can lead to errors. The first concern is that, because the Boolean expression is immediately evaluated, you must be sure that all variables in the expression have a value when the loop is first entered. For example, let's say you want to read input values until the occurrence of the special sentinel value 999, at which time you exit the loop. (Assume that `getInput()` returns the next input value.) The following code has errors:

```java
while (num != 999) {
    num = getInput();
    // now process the input value num
}
```

The problem is that `num` is given a value by the `getInput()` method on the first line of the loop. However, we initially test `num` to see if it is 999 when we start the loop, before it is ever assigned an input value. (We could rely on the fact that all integer variables are automatically initialized to 0, but we should write the program correctly in the first place rather than rely on Java to fix our mistakes.) Another problem is that, once the while loop is entered, it will continue executing to the end of the loop body, even if the boolean expression becomes true somewhere in the middle of the loop. So, when we input the sentinel value 999 on the first line, we will still continue with the rest of the loop body and will process the value 999 as if it were “real” data, which it is not.

The correct way to write the loop is as follows:

```java
num = getInput(); // read the first data value
while (num != 999) {
```
// process this input value.
// and then get the next
num = getInput();
}

This version of the loop will read a number before we ever start the loop so that the variable num has an initial value. If this value is 999, we will skip the entire loop. If it is not 999, then we enter the loop, process this data value, and get the next input to set up for the next pass through the loop.

Another common programming error when using iterative statements is the off-by-one error. It is caused by writing a loop that executes either one time too many or one time too few. The program in Figure A.13, for example, is supposed to input \( N \) real numbers and compute the average of all these numbers.

```java
float sum; // the sum of all numbers
float number; // the next input value
final int N = 100 // the number of input values
int count; // a count of how many values // have been read in so far

count = 0;
sum = 0.0;
while (count <= N) {
    number = getInput();
    sum = sum + number;
    count++;
}

float average = sum / (float) count;
```

**Figure A.13** Program that should read in \( N \) real numbers

There is an off-by-one error in this code, although it is certainly not obvious. The problem is that we start counting the number of times we have done the loop at 0 and continue up to and including the value \( N \). This represents a total of \( N + 1 \) iterations, one more than desired. The correct test to use in the Boolean expression is \((\text{count} < N)\) rather than \((\text{count} <= N)\). Our choice of the wrong relational operator has led to an off-by-one error and an incorrect program. The moral of this example is that every time you write a loop, be sure to check that it starts at the place you want and ends at exactly the place you want—not one time too many or one time too few.

**A.5.4.2 The do/while Statement**

The do/while statement is similar to the while described in the previous section in that it repeats a block of statements as long as a Boolean expression remains true. The main difference is that the do/while is a posttest looping construct. The body of the loop is executed, and then the Boolean condition is tested at the end of the iteration. This means that the body of a do/while loop must be executed at least once. The syn-
tax of the do/while statement is shown in Figure A.14(a), and its behavior is diagramed in Figure A.14(b).

![Diagram of do/while statement behavior](image)

**FIGURE A.14** do/while statement in Java

The other difference between the do/while statement of Figure A.14 and the while statement of Figure A.12 is that the entire do/while construct is terminated with a semicolon.

The while is used more often than the do/while because it is somewhat unusual to construct a loop that must always be executed at least once. With most loops, there is the possibility that the loop will be skipped entirely. However, if a loop does have the characteristic that it must execute at least once, then the do/while can be an appropriate control structure.

The code fragment in Figure A.15 shows the outline for a block of code that inputs a value from a user, computes an answer (not shown), prints that answer, and then asks the user if they want to repeat this operation. The do/while is quite appropriate because the computation must be done at least once.

```java
// This is a loop that solves a problem and asks the
// user if they want to do it again. The computation
// of the answer is not shown
do
  value = getInput();    // get the data

  // Compute the answer
  answer = ...;

[continued]
```

**FIGURE A.15** Obtaining user input using a do/while loop
/\ Print the results
System.out.println(" Answer = ' + answer);

/\ Find out what the user wants to do
System.out.println(" Do it again, Y or N? ");
response = getAnswer(); // get the user's answer
while ((response == 'Y') || (response == 'y'));

**Figure A.15** Obtaining user input using a do/while loop (continued)

### A.5.4.3 The for Statement

The for statement is an extremely powerful iterative construct, and for certain situations, it is by far the most convenient looping statement. It works best when there are four specific components in the loop: (a) an *initialization* expression executed exactly once before the loop begins, (b) a *test* to see if we are done before the loop body is executed (i.e., a pretest looping condition), (c) the *loop body*, and (d) an *update* operation that is executed at the completion of each iteration. The general model for this type of loop structure is shown using a *while* statement to construct it:

```
initialization
while (test) {
    loop body
    update
}
```

While sounding somewhat specialized, this type of loop structure is rather common and occurs quite often in computer science problems. We have just shown that it is possible to implement this loop using a *while* statement. However, note that in addition to the loop body, you need three separate statements to construct this loop: the initialization, test, and update commands. The *Java* for statement simplifies writing this type of loop by combining all three of these operations into a single construct. The resulting for behaves in an identical fashion to the *while* loop shown but with less work and fewer statements. The syntax of the for is given in Figure A.16(a). and its behavior is summarized in Figure A.16(b).

Although the initialization expression can be any statement in Java, it is almost always an assignment statement that initializes either a loop counter or another variable that controls how many times the loop is executed. The test is a Boolean expression that determines when the loop has completed. Finally, the update expression indicates that the loop body has been done one more time. It is often implemented as an increment (or decrement) of a counter that tallies the total number of times the loop has been done.

For example, the following for loop evaluates the series $1^3 + 2^3 + 3^3 + \ldots + N^3$. (It assumes that N has already been assigned a value.)
double sum = 0.0;
for (int i = 1; i <= N; i++) {
    sum += (double) (i * i * i);
}

Notice how easy it is to understand the behavior of this loop because all its parameters appear in a single place: at the very top of the loop. In this case, we can determine that the loop counter \( i \) starts at 1, the loop continues as long as \( i \) is less than or equal to \( N \), and the loop counter \( i \) is incremented by 1 after each pass. Thus, it is simple to determine that this loop executes exactly \( N \) times. Notice also that the loop counter \( i \) has been declared inside the for statement itself. The scope of \( i \) is the body of the for loop, which means the final value of \( i \) will not be available after the loop terminates. While not required, this is quite common since we no longer need this counter once the loop has exited.

The for loop in Figure A.17 iterates through an array \( x[] \) of integer values, counting how many elements in the array are negative, how many are zero, and how many are positive. The length of the array determines the total number of times the loop body is executed.

The increment and update expressions can contain a comma-separated list of expressions. This makes it possible for a programmer to specify that more than one action occur during the initialization or update phases of the for loop. The update expression can consist of any valid expressions; however, the initialization expression can be either a comma-separated list of declaration statements or expression state-
negatives = zeros = positives = 0;

for (int k = 0; k < x.length; k++) {
    if (x[k] < 0) {
        negatives++;
    } else {
        if (x[k] == 0) {
            zeros++;
        } else {
            positives++;
        }
    }
}

**Figure A.17** Iterating through an array using a for loop

In other words, you cannot mix declarations and expressions in the initialization portion of the for loop. So, for example, we could incorporate the initialization of the three counters—negatives, zeros, and positives—into the initialization expression of the for statement itself (Figure A.18).

Note in Figure A.18 that we had to declare the variables k, negatives, zeros, and positives outside the for loop because we cannot mix declarations with expressions in the initialization expression of a for loop.

```java
int k;
int negatives;
int zeros;
int positives;

for (k=0, negatives=0, zeros=0, positives=0; k < x.length; k++) {
    if (x[k] < 0) {
        negatives++;
    } else {
        if (x[k] == 0) {
            zeros++;
        } else {
            positives++;
        }
    }
}
```

**Figure A.18** Using the comma operator in a for loop
The program fragment in Figure A.19 contains a for loop that reverses the contents of an array \( x \) and uses multiple expressions in both the initialization and update portions in the for loop. Since \( i \) and \( j \) are only needed within the loop itself, they are declared in the initialization expression of the for loop, which means the scope of these variables is limited to the body of the loop.

```java
// The array that will be reversed
int x[] = new int[10];

// The variables i and j identify the elements to reverse.
// i starts out at the front of the array and j starts at
// the back. The loop continues as long as i and j have
// not passed each other.
for (int i = 0, j = x.length-1; i < j; i++, j--) {
    // Swap the elements
    int temp = x[i];
    x[i] = x[j];
    x[j] = temp;
}
```

**Figure A.19** Reversing an array

### A.6 METHODS

The last topic that will be introduced in this Appendix is how to write methods. The general structure of a Java method is given in Figure A.20, where [] indicates an optional construct:

```java
[modifiers] return-type method-name( parameter list).
{
    local declarations
    method body
}
```

**Figure A.20** General structure of a Java method

The optional *modifiers* are zero, one, or more keywords that describe characteristics of how this method can be used. These modifiers, such as public, private, protected, and static, are introduced and described in detail in Chapter 3, and they will not be dealt with any further in this Appendix.

The *return-type* specifies the data type of the value that is returned by this method. If the method does not return an explicit value, then the reserved word void is used as the return type.

Finally, the *parameter list* is a list of zero, one, or more parameters separated by commas. The syntax of this list is:
(type name, type name, . . . , type name)

If the method has no parameters, then we use the notation (). In Java, all parameters are passed by value. This means that a copy of the value is passed to the method rather than the value itself.

Here is the first line of a method called `myMethod` that contains three parameters—an integer and two floats—and returns a `String`:

```java
String myMethod(int x, float y, double z)
```

The **signature** of a method is a listing of the data types of all of the method's parameters. Thus, the signature of method `myMethod` just shown is:

```java
int, float, double
```

The signature of a method is an extremely important characteristic for the following reason: It is permissible to have two or more methods with the same name as long as they have different signatures. This is called **method overloading**. Thus, it would be acceptable to declare the following two methods, with the same name `myMethod`, within your program:

```java
String myMethod(byte b, double d);
void myMethod(float f)
```

This is because the signatures of these methods

```java
byte, double
float
```

are all unique and can be distinguished from each other. A Java compiler would not have a problem determining which of these three versions of `myMethod()` you wish to execute. It would simply examine the data types of the parameters contained in the parameter list.

The body of a method is a single compound statement that can contain any of the Java declarations and statements that have been introduced and discussed in the previous sections. The one new statement that will appear within a method is the **return statement**, which specifies the return value of the method. The syntax of this statement is

```java
return expression;
```

where `expression` evaluates to a value of the same data type as the return type of the method. In addition to providing a return value, the return statement also terminates execution of the current method and returns control to the method that invoked this one. If the method has a `void` return value, then the return statement is written as:

```java
return;
```
Figure A.21 shows a method that computes the discriminant of the quadratic equation $ax^2 + bx + c$. The discriminant is the value $b^2 - 4ac$.

```java
// preconditions: three double values a, b, c
// postconditions: returns the value b^2 - 4ac
public double discriminant(double a, double b, double c) {
    double disc;
    disc = (b * b) - 4.0 * a * c;
    return disc;
}
```

**FIGURE A.21** Example Java method

Note that we started out with some helpful comments that describe the overall behavior of the method, namely, its pre- and postconditions. This can be extremely useful to anyone reading your program, and it is a habit that you should immediately adopt. In Chapter 1, we talk more about the use of comments in Java, especially a special form of comment called the Javadoc comment.

Note that the method in Figure A.21 includes the declaration of a double variable called disc. As we mentioned earlier, the scope of this declaration is the block in which it is declared, which in this case is the body of this method. Variables declared within a method are called local variables, and they only exist for the life of the method. When the method terminates, these variables disappear, and they are no longer available. Finally, note that the method returns the value disc using a return statement. This is because the declaration of the method indicated that it returns a double value. Failure to return a value of the correct type will cause an error.

To use a method, you must invoke it. If the method has a nonvoid return value, then you can place a call to the method anywhere that you could legally place a data value of the specified return type. In the case of the method `discriminant()` in Figure A.21, which returns a double, you can place a call to this method anywhere in the program that you could legally place a variable of type double.

For example, here is how we might invoke the method `discriminant()` in a program that evaluates quadratic equations:

```java
double x = getInput(); // assume getInput reads a
// single input value
double y = getInput();
double z = getInput();

double d = discriminant(x, y, z); // compute the discriminant

if (d < 0.0)
    System.out.println("*** Complex roots ***");
else
    // compute the real roots here
```
If a method has a void return value, then you invoke it by writing the method name with the appropriate parameters. For example, if you have written the following void method

```java
void exampleMethod(int j)
```

then it could be invoked in the following way:

```java
int a;
...
exampleMethod(a); // invoking a void method
```

It is also possible to invoke a method with a nonvoid return value in the same way. All that happens is that the return value of the method is discarded and not used. For example, the method `discriminant()` in Figure A.21 returns a double value. However, it would be permissible to write

```java
discriminant(x, y, z);
```

which invokes `discriminant()` but does not keep the value it returns. This is not very useful because in this case we invoked the method specifically to compute a result. However, there are methods that return an "optional" value in the sense that you may or may not wish to keep it.

### A.7 SUMMARY

This Appendix has been a brief overview of the basic procedural constructs of Java. This treatment has included coverage of primitive data types, reference types, declarations, expressions, assignment, control statements, simple input/output methods, and parameters. As was mentioned earlier, this coverage is not meant to replace the comprehensive treatment of the language that would be found in a beginning Java text. However, if you come into the course with a good knowledge of a language similar to Java (e.g., C++, Visual Basic), then the material in this Appendix should be adequate to allow you to read and understand all the programming examples that are presented.

In addition to the material in this Appendix, a number of important advanced features of Java are introduced within the main body of the text. These concepts include support for object-oriented programming, exceptions, streams, threads, GUIs, and networks. Together with the Appendix, they provide a relatively thorough overview of the Java language.