

Creating Java Programs

In this chapter, you will:

- ⦿ Define basic programming terminology
- ⦿ Compare procedural and object-oriented programming
- ⦿ Describe the features of the Java programming language
- ⦿ Analyze a Java application that produces console output
- ⦿ Compile a Java class and correct syntax errors
- ⦿ Run a Java application and correct logical errors
- ⦿ Add comments to a Java class
- ⦿ Create a Java application that produces GUI output
- ⦿ Find help

Learning Programming Terminology

A **computer program** is a set of instructions that you write to tell a computer what to do. Computer equipment, such as a monitor or keyboard, is **hardware**, and programs are **software**. A program that performs a task for a user (such as calculating and producing paychecks, word processing, or playing a game) is **application software**; a program that manages the computer itself (such as Windows or Linux) is **system software**. The **logic** behind any computer program, whether it is an application or system program, determines the exact order of instructions needed to produce desired results. Much of this book describes how to develop the logic to create application software.

All computer programs ultimately are converted to machine language. **Machine language**, or **machine code**, is the most basic set of instructions that a computer can execute. Each type of processor has its own set of machine language instructions. Programmers often describe machine language using 1s and 0s to represent the on-and-off circuitry of computer systems.

Machine language is a **low-level programming language**, or one that corresponds closely to a computer processor's circuitry. Low-level languages require you to use memory addresses for specific machines when you create commands. This means that low-level languages are difficult to use and must be customized for every type of machine on which a program runs.

Fortunately, programming has evolved into an easier task because of the development of high-level programming languages. A **high-level programming language** allows you to use a vocabulary of reasonable terms, such as *read*, *write*, or *add*, instead of the sequences of 1s and 0s that perform these tasks. High-level languages also allow you to assign single-word, intuitive names to areas of computer memory, such as `hoursWorked` or `rateOfPay`, rather than having to remember the memory locations. Java is a high-level programming language.

Each high-level language has its own **syntax**, or rules of the language. For example, depending on the specific high-level language, you might use the verb *print* or *write* to produce output. All languages have a specific, limited vocabulary (the language's **keywords**) and a specific set of rules for using that vocabulary. When you are learning a computer programming language, such as Java, C++, or Visual Basic, you really are learning the vocabulary and syntax rules for that language.

Using a programming language, programmers write a series of **program statements**, similar to English sentences, to carry out the tasks they want the program to perform. Program statements are also known as **commands** because they are orders to the computer, such as "output this word" or "add these two numbers."

After the program statements are written, high-level language programmers use a computer program called a **compiler** or **interpreter** to translate their language statements into machine language. A compiler translates an entire program before carrying out the statement, or **executing** it, whereas an interpreter translates one program statement at a time, executing a statement as soon as it is translated.



Whether you use a compiler or interpreter often depends on the programming language you use. For example, C++ is a compiled language, and Visual Basic is an interpreted language. Each type of translator has its supporters; programs written in compiled languages execute more quickly, whereas programs written in interpreted languages are easier to develop and debug. Java uses the best of both technologies: a compiler to translate your programming statements and an interpreter to read the compiled code line by line when the program executes (also called **at run time**).

Compilers and interpreters issue one or more error messages each time they encounter an invalid program statement—that is, a statement containing a **syntax error**, or misuse of the language. Subsequently, the programmer can correct the error and attempt another translation by compiling or interpreting the program again. Locating and repairing all syntax errors is the first part of the process of **debugging** a program—freeing the program of all errors. Figure 1-1 illustrates the steps a programmer takes while developing an executable program. You will learn more about debugging Java programs later in this chapter.

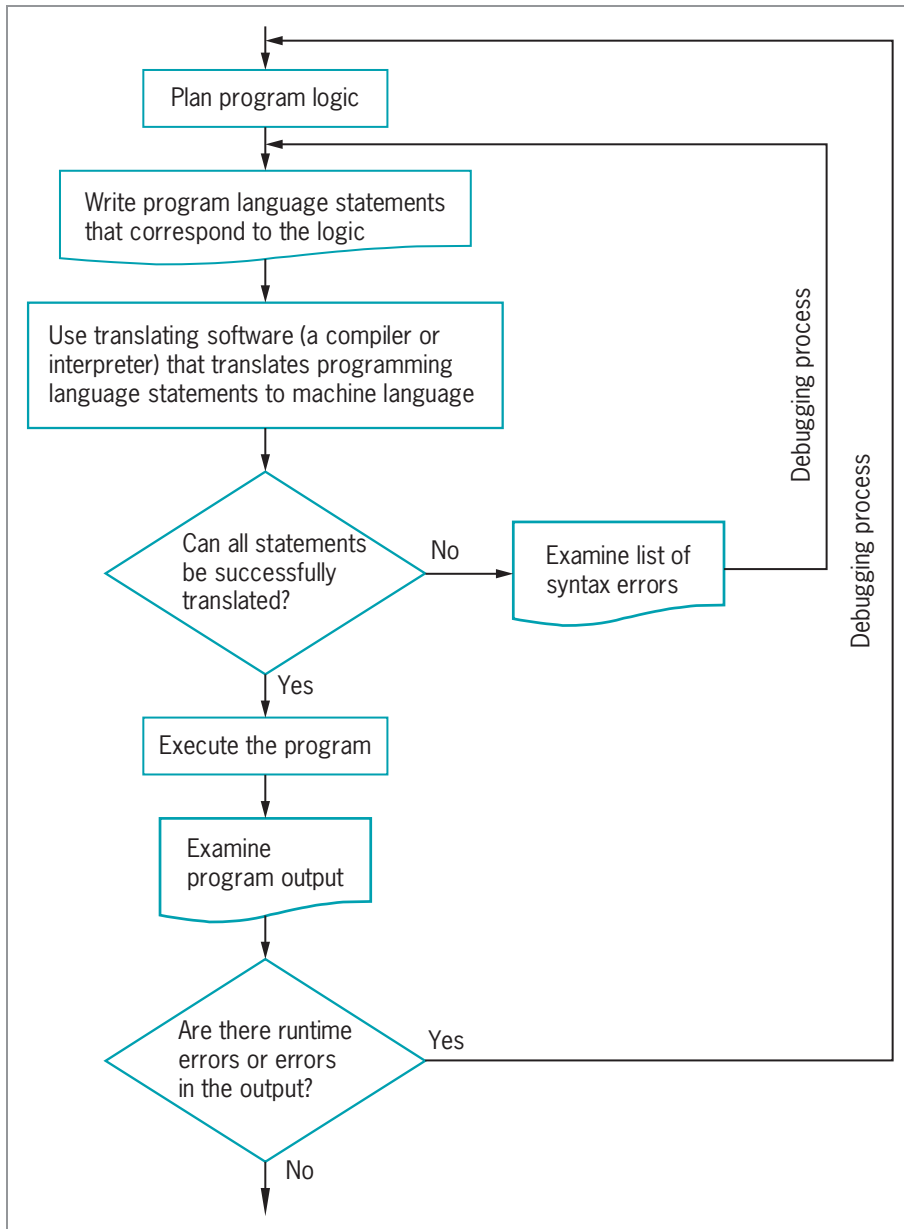


Figure 1-1 The program development process

As Figure 1-1 shows, you might be able to use a computer language's syntax correctly but still have errors to correct. In addition to learning the correct syntax for a particular language, a programmer must also understand computer programming logic. When you develop a program of any significant size, you should plan its logic before you write any program statements. Correct logic requires that all the right commands be issued in the appropriate order. Examples of logical errors include multiplying two values when you meant to divide them or producing output prior to obtaining the appropriate input.

Correcting logical errors is the second part of the debugging process and is much more difficult than correcting syntax errors. Syntax errors are discovered when you compile a program, but often you can identify logical errors only when you examine a program's first output. For example, if you know an employee's paycheck should contain the value \$5,000, but you see that it holds \$50 or \$50,000 after you execute a payroll program, a logical error has occurred. Tools that help you visualize and understand logic are presented in the chapter *Making Decisions*.



Programmers call some logical errors **semantic errors**. For example, if you misspell a programming-language word, you commit a syntax error, but if you use a correct word in the wrong context, you commit a semantic error.

TWO TRUTHS & A LIE

Learning Programming Terminology

In each “Two Truths & a Lie” section, two of the numbered statements are true, and one is false. Identify the false statement and explain why it is false.

1. Unlike a low-level programming language, a high-level programming language allows you to use a vocabulary of reasonable terms instead of the sequences of on and off switches that perform the corresponding tasks.
2. A compiler executes each program statement as soon as it is translated, whereas an interpreter translates all of a program's statements before executing any.
3. A syntax error occurs when you misuse a language; locating and repairing all syntax errors is part of the process of debugging a program.

The false statement is #2. A compiler translates an entire program before carrying out any statements, whereas an interpreter translates one program statement at a time, executing a statement as soon as it is translated.

Comparing Procedural and Object-Oriented Programming Concepts

Two popular approaches to writing computer programs are procedural programming and object-oriented programming.

Procedural Programming

Procedural programming is a style of programming in which operations are executed one after another in sequence. In procedural applications, you create names for computer memory locations that can hold values—for example, numbers and text—in electronic form. The named computer memory locations are called **variables** because they hold values that might vary. For example, a payroll program written for a company might contain a variable named `rateOfPay`. The memory location referenced by the name `rateOfPay` might contain different values (a different value for every employee of the company) at different times. During the execution of the payroll program, each value stored under the name `rateOfPay` might have many operations performed on it—the value might be read from an input device, the value might be multiplied by another variable representing hours worked, and the value might be printed on paper. For convenience, the individual operations used in a computer program are often grouped into logical units called **procedures**. For example, a series of four or five comparisons and calculations that together determine a person's federal withholding tax value might be grouped as a procedure named `calculateFederalWithholding`. A procedural program defines the variable memory locations and then calls a series of procedures to input, manipulate, and output the values stored in those locations. When a program **calls a procedure**, the current logic is temporarily abandoned so that the procedure's commands can execute. A single procedural program often contains hundreds of variables and procedure calls. Procedures are also called *modules*, *methods*, *functions*, and *subroutines*. Users of different programming languages tend to use different terms. As you will learn later in this chapter, Java programmers most frequently use the term *method*.

Object-Oriented Programming

Object-oriented programming is an extension of procedural programming in which you take a slightly different approach to writing computer programs. Writing **object-oriented programs** involves creating classes, which are blueprints for objects; creating objects from those classes; and creating applications that use those objects. After creation, classes can be reused repeatedly to develop new programs. Thinking in an object-oriented manner involves envisioning program components as objects that belong to classes and that are similar to concrete objects in the real world; then, you can manipulate the objects and have them interrelate with each other to achieve a desired result.



Programmers use *OO* as an abbreviation for *object-oriented*; it is pronounced “oh oh.” Object-oriented programming is abbreviated *OOP*, and pronounced to rhyme with *soup*.

Originally, object-oriented programming was used most frequently for two major types of applications:

- **Computer simulations**, which attempt to mimic real-world activities so that their processes can be improved or so that users can better understand how the real-world processes operate
- **Graphical user interfaces**, or **GUIs** (pronounced “gooeys”), which allow users to interact with a program in a graphical environment

Thinking about objects in these two types of applications makes sense. For example, a city might want to develop a program that simulates traffic patterns to help prevent traffic tie-ups. By creating a model with objects such as cars and pedestrians that contain their own data and rules for behavior, the simulation can be set in motion. For example, each car object has a specific current speed and a procedure for changing that speed. By creating a model of city traffic using objects, a computer can create a simulation of a real city at rush hour.

Creating a GUI environment for users also is a natural use for object orientation. It is easy to think of the components a user manipulates on a computer screen, such as buttons and scroll bars, as similar to real-world objects. Each GUI object contains data—for example, a button on a screen has a specific size and color. Each object also contains behaviors—for example, each button can be clicked and reacts in a specific way when clicked. Some people consider the term *object-oriented programming* to be synonymous with GUI programming, but object-oriented programming means more. Although many GUI programs are object oriented, do not assume that all object-oriented programs use GUI objects. Modern businesses use object-oriented design techniques when developing all sorts of business applications, whether they are GUI applications or not.

Understanding object-oriented programming requires grasping three basic concepts:

- Encapsulation as it applies to classes as objects
- Inheritance
- Polymorphism

Understanding Classes, Objects, and Encapsulation

In object-oriented terminology, a **class** is a term that describes a group or collection of objects with common properties. In the same way that a blueprint exists before any houses are built from it, and a recipe exists before any cookies are baked from it, so does a class definition exist before any objects are created from it. A **class definition** describes what attributes its objects will have and what those objects will be able to do. **Attributes** are the characteristics that define an object; they are **properties** of the object. When you learn a programming language such as Java, you learn to work with two types of classes: those that have already been developed by the language’s creators and your own new, customized classes.

An **object** is a specific, concrete **instance** of a class. When you create an object, you **instantiate** it. You can create objects from classes that you write and from classes written by other programmers, including Java’s creators. The values contained in an object’s properties

often differentiate instances of the same class from one another. For example, the class `Automobile` describes what `Automobile` objects are like. Some properties of the `Automobile` class are make, model, year, and color. Each `Automobile` object possesses the same attributes but not, of course, the same values for those attributes. One `Automobile` might be a 2009 white Ford Taurus and another might be a 2014 red Chevrolet Camaro. Similarly, your dog has the properties of all `Dogs`, including a breed, name, age, and whether his shots are current. The values of the properties of an object are also referred to as the object's **state**. In other words, you can think of objects as roughly equivalent to nouns, and of their attributes as similar to adjectives that describe the nouns.

When you understand an object's class, you understand the characteristics of the object. If your friend purchases an `Automobile`, you know it has a model name, and if your friend gets a `Dog`, you know the dog has a breed. Knowing what attributes exist for classes allows you to ask appropriate questions about the states or values of those attributes. For example, you might ask how many miles the car gets per gallon, but you would not ask whether the car has had shots. Similarly, in a GUI operating environment, you expect each component to have specific, consistent attributes and methods, such as a window having a title bar and a close button, because each component gains these properties as a member of the general class of GUI components. Figure 1-2 shows the relationship of some `Dog` objects to the `Dog` class.



By convention, programmers using Java begin their class names with an uppercase letter. Thus, the class that defines the attributes and methods of an automobile would probably be named `Automobile`, and the class for dogs would probably be named `Dog`. However, following this convention is not required to produce a workable program.

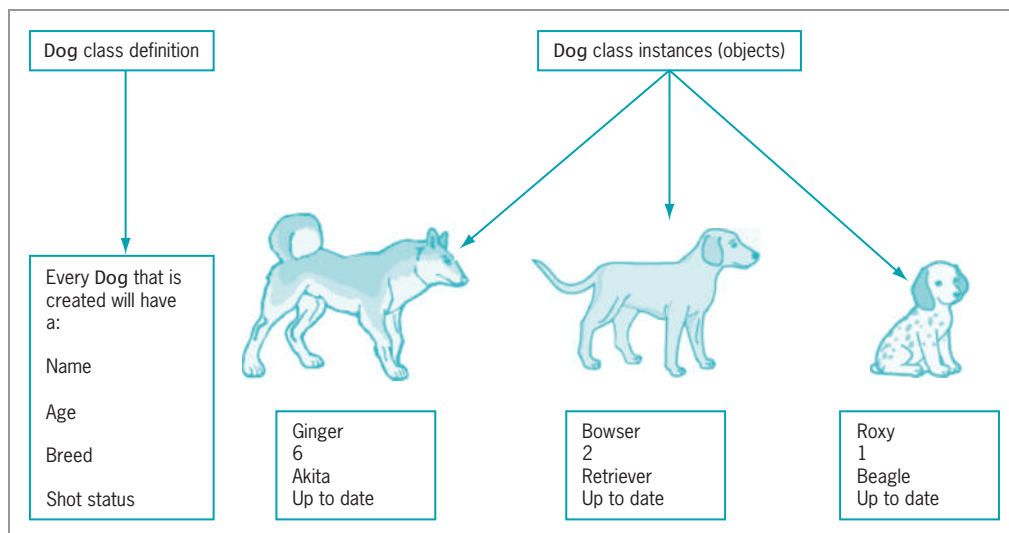


Figure 1-2 A class definition and some objects created from it

Besides defining properties, classes define methods their objects can use. A **method** is a self-contained block of program code that carries out some action, similar to a procedure in a procedural program. An `Automobile`, for example, might have methods for moving forward, moving backward, being filled with gasoline, and being washed. Some methods can ascertain certain attributes, such as the current speed of an `Automobile` and the status of its gas tank. Similarly, a `Dog` can walk or run, eat food, and get a bath, and there are methods to determine how hungry the `Dog` is or what its name is. GUI operating system components can be maximized, minimized, and dragged. In other words, if objects are similar to nouns, then methods are similar to verbs.

In object-oriented classes, attributes and methods are encapsulated into objects that are then used much like real-world objects. **Encapsulation** refers to two closely related object-oriented notions:

- Encapsulation is the enclosure of data and methods within an object. Encapsulation allows you to treat all of an object's methods and data as a single entity. Just as an actual dog contains all of its attributes and abilities, so would a program's `Dog` object.
- Encapsulation also refers to the concealment of an object's data and methods from outside sources. Concealing data is sometimes called *information hiding*, and concealing how methods work is *implementation hiding*; you will learn more about both terms in the chapter *Using Methods, Classes, and Objects*. Encapsulation lets you hide specific object attributes and methods from outside sources and provides the security that keeps data and methods safe from inadvertent changes.

If an object's methods are well written, the user is unaware of the low-level details of how the methods are executed, and the user must simply understand the interface or interaction between the method and the object. For example, if you can fill your `Automobile` with gasoline, it is because you understand the interface between the gas pump nozzle and the vehicle's gas tank opening. You don't need to understand how the pump works mechanically or where the gas tank is located inside your vehicle. If you can read your speedometer, it does not matter how the displayed figure is calculated. As a matter of fact, if someone produces a superior, more accurate speed-determining device and inserts it in your `Automobile`, you don't have to know or care how it operates, as long as your interface remains the same. The same principles apply to well-constructed classes used in object-oriented programs—programs that use classes only need to work with interfaces.

Understanding Inheritance and Polymorphism

An important feature of object-oriented program design is **inheritance**—the ability to create classes that share the attributes and methods of existing classes but with more specific features. For example, `Automobile` is a class, and all `Automobile` objects share many traits and abilities. `Convertible` is a class that inherits from the `Automobile` class; a `Convertible` is a type of `Automobile` that has and can do everything a “plain” `Automobile` does—but with an added mechanism for and an added ability to lower its top. (In turn, `Automobile` inherits from the `Vehicle` class.) `Convertible` is not an object—it is a class. A specific `Convertible` is an object—for example, `my1967BlueMustangConvertible`.

Inheritance helps you understand real-world objects. For example, the first time you encounter a `Convertible`, you already understand how the ignition, brakes, door locks, and other `Automobile` systems work. You need to be concerned only with the attributes and methods that are “new” with a `Convertible`. The advantages in programming are the same—you can build new classes based on existing classes and concentrate on the specialized features you are adding.

A final important concept in object-oriented terminology is **polymorphism**. Literally, polymorphism means “many forms”—it describes the feature of languages that allows the same word or symbol to be interpreted correctly in different situations based on the context. For example, although the classes `Automobile`, `Sailboat`, and `Airplane` all inherit from `Vehicle`, `turn` and `stop` methods work differently for instances of those classes. The advantages of polymorphism will become more apparent when you begin to create GUI applications containing features such as windows, buttons, and menu bars. In a GUI application, it is convenient to remember one method name, such as `setColor` or `setHeight` and have it work correctly no matter what type of object you are modifying.

When you see a plus sign (+) between two numbers, you understand they are being added. When you see it carved in a tree between two names, you understand that the names are linked romantically. Because the symbol has diverse meanings based on context, it is polymorphic. Chapters 10 and 11 provide more information about inheritance and polymorphism and how they are implemented in Java.



Watch the video *Object-Oriented Programming*.

TWO TRUTHS & A LIE

Comparing Procedural and Object-Oriented Programming Concepts

1. An instance of a class is a created object that possesses the attributes and methods described in the class definition.
2. Encapsulation protects data by hiding it within an object.
3. Polymorphism is the ability to create classes that share the attributes and methods of existing classes, but with more specific features.

The false statement is #3. Inheritance is the ability to create classes that share the attributes and methods of existing classes but with more specific features; polymorphism describes the ability to use one term to cause multiple actions.

Features of the Java Programming Language

Java was developed by Sun Microsystems as an object-oriented language for general-purpose business applications and for interactive, World Wide Web–based Internet applications.

Some of the advantages that have made Java so popular in recent years are its security features and the fact that it is **architecturally neutral**: Unlike other languages, you can use Java to write a program that runs on any operating system (such as Windows, Mac OS, or Linux) or device (such as PCs, phones, and tablet computers).

Java can be run on a wide variety of computers and devices because it does not execute instructions on a computer directly. Instead, Java runs on a hypothetical computer known as the **Java Virtual Machine (JVM)**. When programmers call the JVM *hypothetical*, they don't mean it doesn't exist. Instead, they mean it is not a physical entity created from hardware but is composed only of software.

Figure 1-3 shows the Java environment. Programming statements written in a high-level programming language are **source code**. When you write a Java program, you first construct the source code using a text editor such as Notepad or a development environment and source code editor such as **jGRASP**, which you can download from the Web for free. A **development environment** is a set of tools that help you write programs by providing such features as displaying a language's keywords in color. The statements are saved in a file; then, the Java compiler converts the source code into a binary program of **bytecode**. A program called the **Java interpreter** then checks the bytecode and communicates with the operating system, executing the bytecode instructions line by line within the Java Virtual Machine. Because the Java program is isolated from the operating system, the Java program also is insulated from the particular hardware on which it is run. Because of this insulation, the JVM provides security against intruders accessing your computer's hardware through the operating system. Therefore, Java is more secure than other languages. Another advantage provided by the JVM means less work for programmers—when using other programming languages, software vendors usually have to produce multiple versions of the same product (a Windows version, Macintosh version, UNIX version, Linux version, and so on) so all users can run the program. With Java, one program version runs on all these platforms. **“Write once, run anywhere” (WORA)** is a slogan developed by Sun Microsystems to describe the ability of one Java program version to work correctly on multiple platforms.

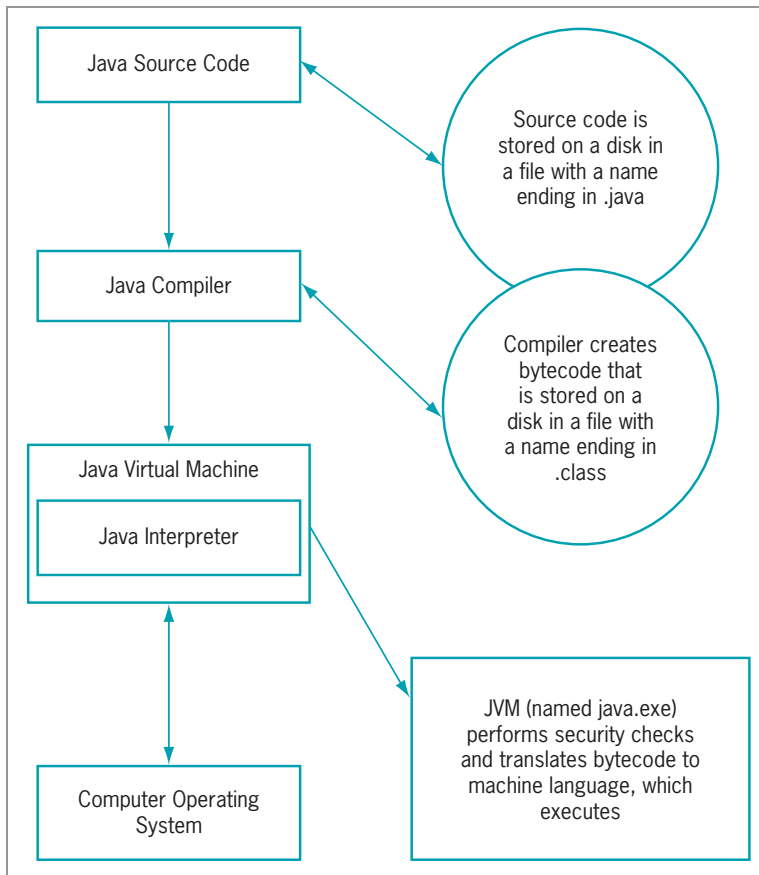


Figure 1-3 The Java environment

Java is also simpler to use than many other object-oriented languages. Java is modeled after C++. Although neither language is easy to read or understand on first exposure, Java does eliminate some of the most difficult-to-understand features in C++, such as pointers and multiple inheritance.

Java Program Types

You can write two kinds of programs using Java:

- **Applets** are programs that are embedded in a Web page. You will create applets in the chapter *Applets, Images, and Sound*.
- **Java applications** are stand-alone programs. Java applications can be further subdivided into **console applications**, which support character output to a computer screen in a DOS window, for example, and **windowed applications**, which create a GUI

with elements such as menus, toolbars, and dialog boxes. Console applications are the easiest applications to create; you start using them in the next section.

TWO TRUTHS & A LIE

Features of the Java Programming Language

1. Java was developed to be architecturally neutral, which means that anyone can build an application without extensive study.
2. After you write a Java program, the compiler converts the source code into a binary program of bytecode.
3. Java programs that are embedded in a Web page are called applets, while stand-alone programs are called Java applications.

The false statement is #1. Java was developed to be architecturally neutral, which means that you can use Java to write a program that will run on any platform.

Analyzing a Java Application that Produces Console Output

At first glance, even the simplest Java application involves a fair amount of confusing syntax. Consider the application in Figure 1-4. This program is written on seven lines, and its only task is to display “First Java application” on the screen.

```
public class First
{
    public static void main(String[] args)
    {
        System.out.println("First Java application");
    }
}
```

Figure 1-4 The First class



When you see program code in figures in this book, Java keywords as well as `true`, `false`, and `null` are blue, and all other program elements are black. A complete list of Java keywords is shown later in this chapter.



The code for every complete program shown in this book is available in a set of student files you can download so that you can execute the programs on your own computer.

Understanding the Statement that Produces the Output

Although the program in Figure 1-4 occupies several lines, it contains only one Java programming statement. The statement `System.out.println("First Java application");` does the actual work in this program. Like all Java statements, this one ends with a semicolon. Most Java programming statements can be spread across as many lines as you choose, as long as you place line breaks in appropriate places. For example, in the program in Figure 1-4, you could place a line break before or after the opening parenthesis, or before or after the closing parenthesis. However, you usually want to place a short statement on a single line.

The text “First Java application” is a **literal string** of characters—a series of characters that will appear in output exactly as entered. Any literal string in Java is written between double quotation marks. In Java, a literal string cannot be broken and placed on multiple lines. Figure 1-5 labels this string and the other parts of the statement.

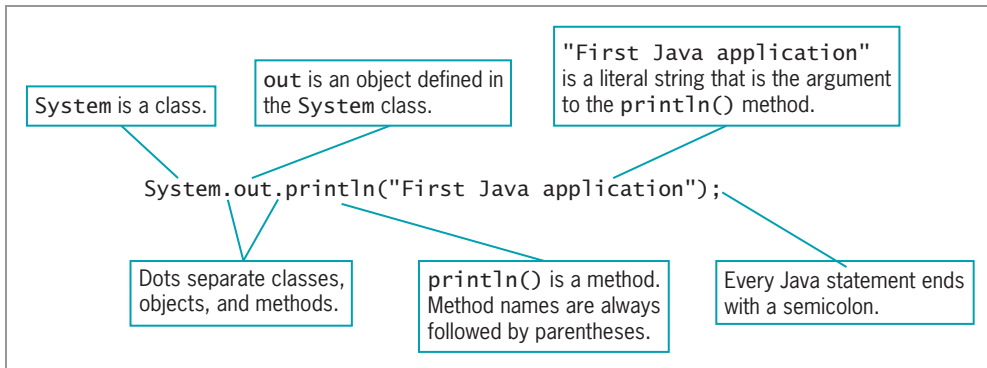


Figure 1-5 Anatomy of a Java statement

The string “First Java application” appears within parentheses because the string is an argument to a method, and arguments to methods always appear within parentheses. **Arguments** are pieces of information that are sent into a method, usually because the method requires the information to perform its task or carry out its purpose. The act of sending arguments to a method is called **passing arguments** to the method. As an example, consider placing a catalog order with a company that sells sporting goods. Processing a catalog order is a method that consists of a set of standard procedures—recording the order, checking the availability of the item, pulling the item from the warehouse, and so on. Each catalog order also requires a set of data items, such as which item number you are ordering and the quantity of the item desired; these data items can be considered the order’s arguments. If you order two of item 5432 from a catalog, you expect different results than if you order 1,000 of item 9008. Likewise, if you pass the argument “Happy Holidays” to a Java method, you expect different results than if you pass the argument “First Java application”.

Within the statement `System.out.println("First Java application");`, the method to which you are passing "First Java application" is named `println()`. The Java methods `println()` and `print()` both produce output. With `println()`, after the output is displayed, the insertion point moves to the following line. (The insertion point is the point at which the next output appears.) With `print()`, however, the insertion point does not advance to a new line; it remains on the same line as the output.

When you call a method, you always use parentheses following the method name. In this book, you will learn about many methods that require arguments between their parentheses, and many others for which you leave the parentheses empty. The `println()` method can be used with no arguments when you want to output a blank line. Later in this chapter, you will learn about a method named `showMessageDialog()` that requires two arguments. Other methods require more.

Within the statement `System.out.println("First Java application");`, `out` is an object that is a property of the `System` class that refers to the **standard output device** for a system, normally the monitor. The `out` object itself is an instance of the `PrintStream` class, which contains several methods, including `println()`. Technically, you could create the `out` object and write the instructions within the `println()` method yourself, but it would be time consuming, and the creators of Java assumed you frequently would want to display output on a screen. Therefore, the `System` and `PrintStream` classes, the `out` object, and the `println()` method were created as a convenience to the programmer.

Within the statement `System.out.println("First Java application");`, `System` is a class. Therefore, `System` defines attributes for `System` objects, just as the `Dog` class defines the attributes for `Dog` objects. One of the `System` attributes is `out`. (You can probably guess that another attribute is `in` and that it represents an input device.)

The dots (periods) in `System.out.println()` are used to separate the names of the components in the statement. You will use this format repeatedly in your Java programs.

Java is case sensitive; the class named `System` is a completely different class from one named `system`, `SYSTEM`, or even `sYsTeM`, and `out` is a different object from one named `Out` or `OUT`. You must pay close attention to using correct uppercase and lowercase values when you write Java programs.

So, the statement that displays the string "First Java application" contains a class, an object reference, a method call, a method argument, and a statement-ending semicolon, but the statement cannot stand alone; it is embedded within a class, as shown in Figure 1-4.

Understanding the First Class

Everything that you use within a Java program must be part of a class. When you write `public class First`, you are defining a class named `First`. You can define a Java class using any name or **identifier** you need, as long as it meets the following requirements:

- A Java identifier must begin with a letter of the English alphabet, a non-English letter (such as α or π), an underscore, or a dollar sign. A class name cannot begin with a digit.
- A Java identifier can contain only letters, digits, underscores, or dollar signs.

- A Java identifier cannot be a reserved keyword, such as `public` or `class`. (See Table 1-1 for a list of reserved keywords.)
- A Java identifier cannot be one of the following values: `true`, `false`, or `null`. These are not keywords (they are primitive values), but they are reserved and cannot be used.



Java is based on **Unicode**, which is an international system of character representation. The term *letter* indicates English-language letters as well as characters from Arabic, Greek, and other alphabets. You can learn more about Unicode in Appendix B.

| | | | | |
|-----------------------|-----------------------|-------------------------|------------------------|---------------------------|
| <code>abstract</code> | <code>continue</code> | <code>for</code> | <code>new</code> | <code>switch</code> |
| <code>assert</code> | <code>default</code> | <code>goto</code> | <code>package</code> | <code>synchronized</code> |
| <code>boolean</code> | <code>do</code> | <code>if</code> | <code>private</code> | <code>this</code> |
| <code>break</code> | <code>double</code> | <code>implements</code> | <code>protected</code> | <code>throw</code> |
| <code>byte</code> | <code>else</code> | <code>import</code> | <code>public</code> | <code>throws</code> |
| <code>case</code> | <code>enum</code> | <code>instanceof</code> | <code>return</code> | <code>transient</code> |
| <code>catch</code> | <code>extends</code> | <code>int</code> | <code>short</code> | <code>try</code> |
| <code>char</code> | <code>final</code> | <code>interface</code> | <code>static</code> | <code>void</code> |
| <code>class</code> | <code>finally</code> | <code>long</code> | <code>strictfp</code> | <code>volatile</code> |
| <code>const</code> | <code>float</code> | <code>native</code> | <code>super</code> | <code>while</code> |

Table 1-1 Java reserved keywords



Although they are reserved as keywords, `const` and `goto` are not used in Java programs, and they have no function. Both words are used in other languages and were reserved in case developers of future versions of Java wanted to implement them.

It is a Java standard, although not a requirement, to begin class identifiers with an uppercase letter and employ other uppercase letters as needed to improve readability. The style that joins words in which each begins with an uppercase letter is called **Pascal casing**, or sometimes **upper camel casing**. You should follow established conventions for Java so your programs will be easy for other programmers to interpret and follow. This book uses established Java programming conventions.

Table 1-2 lists some valid and conventional class names that you could use when writing programs in Java. Table 1-3 provides some examples of class names that *could* be used in Java (if you use these class names, the class will compile) but that are unconventional and not recommended. Table 1-4 provides some class name examples that are illegal.

| Class Name | Description |
|------------------|--|
| Employee | Begins with an uppercase letter |
| UnderGradStudent | Begins with an uppercase letter, contains no spaces, and emphasizes each new word with an initial uppercase letter |
| InventoryItem | Begins with an uppercase letter, contains no spaces, and emphasizes the second word with an initial uppercase letter |
| Budget2014 | Begins with an uppercase letter and contains no spaces |

Table 1-2 Some valid class names in Java

| Class Name | Description |
|------------------|--|
| Undergradstudent | New words are not indicated with initial uppercase letters, making this identifier difficult to read |
| Inventory_Item | Underscore is not commonly used to indicate new words |
| BUDGET2014 | Using all uppercase letters for class identifiers is not conventional |
| budget2014 | Conventionally, class names do not begin with a lowercase letter |

Table 1-3 Legal but unconventional and nonrecommended class names in Java

| Class Name | Description |
|----------------|---|
| Inventory Item | Space character is illegal in an identifier |
| class | class is a reserved word |
| 2014Budget | Class names cannot begin with a digit |
| phone# | The number symbol (#) is illegal in an identifier |

Table 1-4 Some illegal class names in Java

In Figure 1-4 (and again in Figure 1-6), the line `public class First` is the class header; it contains the keyword `class`, which identifies `First` as a class. The reserved word `public` is an access specifier. An **access specifier** defines the circumstances under which a class can be accessed and the other classes that have the right to use a class. Public access is the most liberal type of access; you will learn about public access and other types of access in the chapter *Using Methods, Classes, and Objects*.

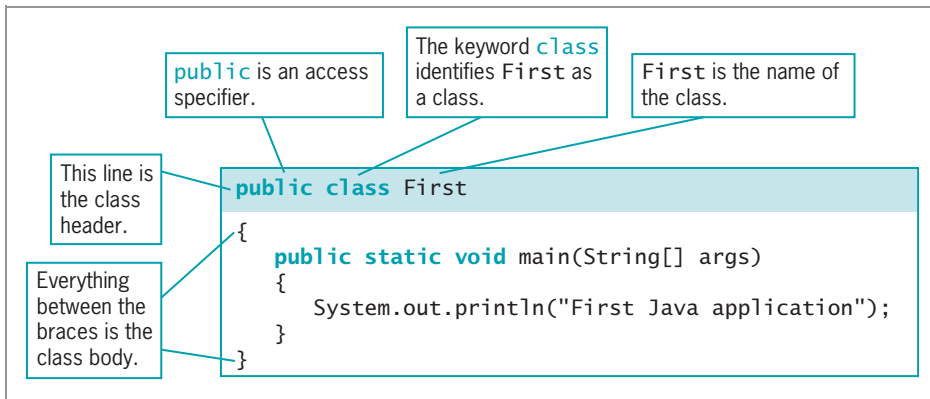


Figure 1-6 The parts of a typical class

After the class header, you enclose the contents of a class within curly braces (`{` and `}`); any data items and methods between the curly braces make up the **class body**. A class body can be composed of any number of data items and methods. In Figure 1-4 (and again in Figure 1-6), the class `First` contains only one method within its curly braces. The name of the method is `main()`, and the `main()` method, like the `println()` method, includes its own set of parentheses. The `main()` method in the `First` class contains only one statement—the statement that uses the `println()` method. The `main()` method does not *contain* any other methods, but it *calls* the `println()` method.

Indent Style

In general, whitespace is optional in Java. **Whitespace** is any combination of nonprinting characters. You use whitespace to organize your program code and make it easier to read. You can insert whitespace between words or lines in your program code by typing spaces, tabs, or blank lines because the compiler ignores these extra spaces. However, you cannot use whitespace within an identifier or keyword.

For every opening curly brace (`{`) in a Java program, there must be a corresponding closing curly brace (`}`), but the placement of the opening and closing curly braces is not important to the compiler. For example, the following class executes in exactly the same way as the one shown in Figure 1-4. The only difference is the layout of the braces—the line breaks occur in different locations.

```
public class First{
    public static void main(String[] args){
        System.out.println("First Java application");
    }
}
```

The indent style shown in the preceding example, in which opening braces are not on separate lines, is known as the **K & R style** and is named for Kernighan and Ritchie, who wrote the first book on the C programming language. The indent style shown in Figure 1-4 and used

throughout this book, in which curly braces are aligned and each occupies its own line, is called the **Allman style** and is named for Eric Allman, a programmer who popularized the style. Java programmers use a variety of indent styles, and all can produce workable Java programs. When you write your own code, you should develop a consistent style. In school, your instructor might have a preferred style, and when you get a job as a Java programmer, your organization most likely will have a preferred style. With many development environments, indentations are made for you automatically as you type.

Most programmers indent a method's statements a few spaces more than its curly braces. Some programmers indent two spaces, some three, and some four. Some programmers use the Tab key to create indentations, but others are opposed to this practice because the Tab key can indicate different indentation sizes on different systems. Some programmers don't care whether tabs or spaces are used, as long as you don't mix them in the same program. The Java compiler does not care how you indent. Again, the most important rule is to develop a consistent style of which your organization approves.

Understanding the `main()` Method

The method header for the `main()` method is quite complex. Figure 1-7 shows the parts of the `main()` method.

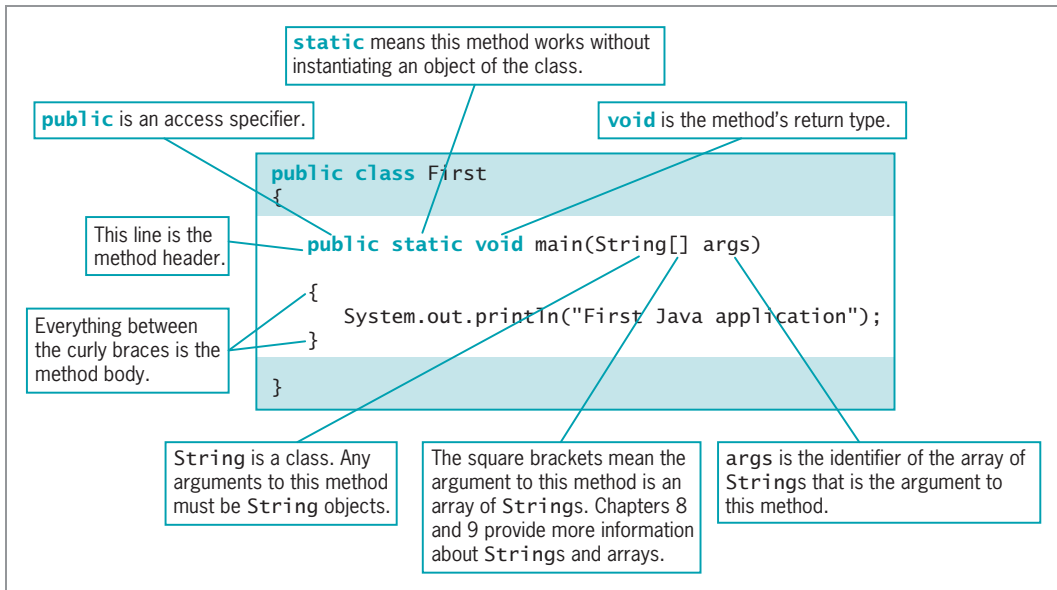


Figure 1-7 The parts of a typical `main()` method

The meaning and purpose of each of the terms used in the method header will become clearer as you complete this textbook; a brief explanation will suffice for now.

- In the method header `public static void main(String[] args)`, the word **public** is an access specifier, just as it is when you use it to define the `First` class.
- In Java, the reserved keyword **static** means that a method is accessible and usable even though no objects of the class exist.
- The keyword **void** used in the `main()` method header indicates that the `main()` method does not return any value when it is called. This doesn't mean that `main()` doesn't produce output—in fact, the method in Figure 1-4 (and in Figure 1-7) does. It only means that the `main()` method does not send any value back to any other method that might use it. You will learn more about return values in the chapter *Methods, Classes, and Objects*.
- The name of the method is `main()`. Not all classes have a `main()` method; in fact, many do not. All Java *applications*, however, must include a class containing a public method named `main()`, and most Java applications have additional classes and methods. When you execute a Java application, the JVM always executes the `main()` method first.
- In the method header `public static void main(String[] args)`, the contents between the parentheses, `String[] args`, represent the type of argument that can be passed to the `main()` method, just as the string "First Java application" is an argument passed to the `println()` method. `String` is a Java class that can be used to hold character strings. The identifier `args` is used to hold any `String` objects that might be sent to the `main()` method. The `main()` method could do something with those arguments, such as display them, but in Figure 1-4, the `main()` method does not actually use the `args` identifier. Nevertheless, you must place an identifier within the `main()` method's parentheses. The identifier does not need to be named `args`—it could be any legal Java identifier—but the name `args` is traditional.



You won't pass any arguments to the `main()` method in this book, but when you run a program, you could. Even though you pass no arguments, the `main()` method must contain `String[]` and a legal identifier (such as `args`) within its parentheses. When you refer to the `String` class in the `main()` method header, the square brackets indicate an array of `String` objects. You will learn more about the `String` class and arrays in Chapters 7, 8, and 9.

The simple application shown in Figure 1-4 has many pieces to remember. However, for now you can use the Java code shown in Figure 1-8 as a shell, in which you replace `AnyClassName` with a class name you choose and the line `/*****/` with any statements that you want to execute.



Watch the video *A Java Program*.

```
public class AnyClassName
{
    public static void main(String[] args)
    {
        /*****/
    }
}
```

Figure 1-8 Shell code

Saving a Java Class

When you write a Java class, you must save it using a storage medium such as a disk, DVD, or USB device. In Java, if a class is `public` (that is, if you use the `public` access specifier before the class name), you must save the class in a file with exactly the same name and a `.java` extension. For example, the `First` class must be stored in a file named `First.java`. The class name and filename must match exactly, including the use of uppercase and lowercase characters. If the extension is not `.java`, the Java compiler does not recognize the file as containing a Java class. Appendix A contains additional information on saving a Java application.

TWO TRUTHS & A LIE

Analyzing a Java Application that Produces Console Output

1. In the method header `public static void main(String[] args)`, the word `public` is an access specifier.
2. In the method header `public static void main(String[] args)`, the word `static` means that a method is accessible and usable, even though no objects of the class exist.
3. In the method header `public static void main(String[] args)`, the word `void` means that the `main()` method is an empty method.

The false statement is #3. In the method header `public static void main(String[] args)`, the word `void` means that the `main()` method does not return any value when it is called.



You Do It

Your First Application

Now that you understand the basics of an application written in Java, you are ready to enter your own Java application into a text editor. It is a tradition among programmers that the first program you write in any language produces “Hello, world!” as its output. You will create such a program now. You can use any text editor, such as Notepad or TextPad, or a development environment, such as jGRASP.



It is best to use the simplest available text editor when writing Java programs. Multifeatured word-processing programs save documents as much larger files because of all the built-in features, such as font styles and margin settings, which the Java compiler cannot interpret.

1. Start any text editor, and then open a new document.
2. Type the class header as follows:
public class Hello

In this example, the class name is `Hello`. You can use any valid name you want for the class. If you choose *Hello*, you always must refer to the class as *Hello*, and not as *hello*, because Java is case sensitive.
3. Press **Enter** once, type `{`, press **Enter** again, and then type `}`. You will add the `main()` method between these curly braces. Although it is not required, the convention used in this book is to place each curly brace on its own line and to align opening and closing curly brace pairs with each other. Using this format makes your code easier to read.
4. As shown in the shaded portion of Figure 1-9, add the `main()` method header between the curly braces, and then type a set of curly braces for `main()`.

```
public class Hello
{
    public static void main(String[] args)
    {
    }
}
```

Figure 1-9 The `main()` method shell for the `Hello` class

(continues)

(continued)

5. Next, add the statement within the `main()` method that will produce the output "Hello, world!". Use Figure 1-10 as a guide for adding the shaded `println()` statement to the `main()` method.

```
public class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello, world!");
    }
}
```

Figure 1-10 Complete Hello class

6. Save the application as **Hello.java**. The class name and filename must match, and you must use the `.java` extension.

Compiling a Java Class and Correcting Syntax Errors

After you write and save an application, two steps must occur before you can view the application's output.

1. You must compile the class you wrote (called the source code) into bytecode.
2. You must use the Java interpreter to translate the bytecode into executable statements.

Compiling a Java Class

To compile your source code from the command line, your prompt should show the folder or directory where your program file is stored. Then, you type `javac` followed by the name of the file that contains the source code. For example, to compile a file named `First.java`, you type the following and then press Enter:

```
javac First.java
```

There will be one of three outcomes:

- You receive a message such as 'javac' is not recognized as an internal or external command, operable program or batch file.
- You receive one or more program language error messages.
- You receive no messages, which means that the application compiled successfully.



When compiling, if the source code file is not in the current path, you can type a full path with the filename. For example:

```
javac c:\java\MyClasses\Chapter.01\First.java
```



In a DOS environment, you can change directories using the `cd` command. For example, to change from the current directory to a subdirectory named `MyClasses`, you type `cd MyClasses` and press Enter. Within any directory, you can back up to the root directory by typing `cd\` and pressing Enter.

If you receive an error message that the command is not recognized, it might mean one of the following:

- You misspelled the command `javac`.
- You misspelled the filename.
- You are not within the correct subfolder or subdirectory on your command line.
- Java was not installed properly. (See Appendix A for information on installation.)

If you receive a programming language error message, there are one or more syntax errors in the source code. Recall that a syntax error is a programming error that occurs when you introduce typing errors into your program or use the programming language incorrectly. For example, if your class name is `first` (with a lowercase *f*) in the source code but you saved the file as `First.java` (with an uppercase *F*), you will receive an error message when you compile the application. The error message will be similar to `class first is public, should be declared in a file named first.java` because *first* and *First* are not the same in a case-sensitive language. If this error occurs, you must reopen the text file that contains the source code and make the necessary corrections.



Appendix A contains information on troubleshooting, including how to change filenames in a Windows environment.

If you receive no error messages after compiling the code in a file named `First.java`, the application compiled successfully, and a file named `First.class` is created and saved in the same folder as the text file that holds the application code. After a successful compile, you can run the class file on any computer that has a Java language interpreter.

Correcting Syntax Errors

Frequently, you might make typing errors as you enter Java statements into your text editor. When you issue the command to compile the class containing errors, the Java compiler produces one or more error messages. The exact error message that appears varies depending on the compiler you are using.

The `FirstWithMissingSemicolon` class shown in Figure 1-11 contains an error—the semicolon is missing from the end of the `println()` statement. (Of course, this class has been helpfully named to alert you to the error.) When you compile this class, an error message similar to the one shown in Figure 1-12 is displayed.

```
public class FirstWithMissingSemicolon
{
    public static void main(String[] args)
    {
        System.out.println("First Java application")
    }
}
```

The statement-ending semicolon has been omitted.

Figure 1-11 The `FirstWithMissingSemicolon` class

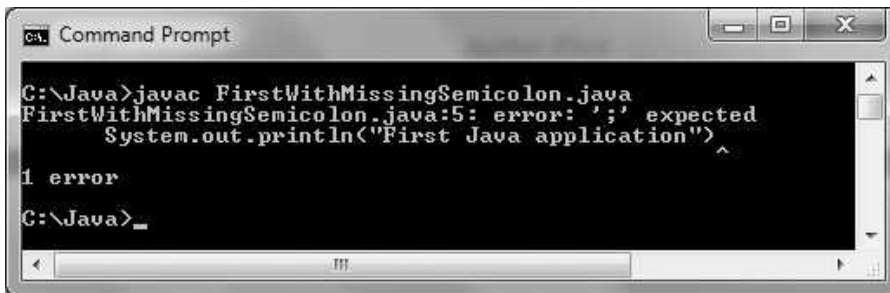


Figure 1-12 Error message generated when the `FirstWithMissingSemicolon` class is compiled

The first line of the error message in Figure 1-12 displays the name of the file in which the error was found (`FirstWithMissingSemicolon.java`), the line number in which it was found (5), and the nature of the error ("`;` expected"). The next line of the error message displays the statement that contains the error, including a caret that points to the exact location where the error was first discovered. As you will see when you write and compile Java programs, the place where an error is discovered is not necessarily where the error was made. Sometimes, it takes a little detective work to interpret an error message and determine its cause.

Finally, the message generated in Figure 1-12 includes a count of the number of errors found—in this case, there is just one error. This is a **compile-time error**, or one in which the compiler detects a violation of language syntax rules and is unable to translate the source code to machine code.

When you compile a class, the compiler reports as many errors as it can find so that you can fix as many errors as possible. Sometimes, one error in syntax causes multiple error messages that normally would not be errors if the first syntax error did not exist, so fixing one error might eliminate multiple error messages. Sometimes, when you fix a compile-time error and recompile a program, new error messages are generated. That's because when you fix the first error, the compiler can proceed beyond that point and possibly discover new errors. Of course, no programmer intends to type a program containing syntax errors, but when you do, the compiler finds them all for you.

TWO TRUTHS & A LIE

Compiling a Java Class and Correcting Syntax Errors

1. After you write and save an application, you can compile the bytecode to create source code.
2. When you compile a class, you create a new file with the same name as the original file but with a .class extension.
3. Syntax errors are compile-time errors.

The false statement is #1. After you write and save an application, you can compile the source code to create bytecode.



You Do It

Compiling a Java Class

You are ready to compile the `Hello` class that you created in the previous “You Do It” section.

1. If it is not still open on your screen, open the **Hello.java** file that you saved in the previous “You Do It” section.
2. Go to the command-line prompt for the drive and folder or subdirectory in which you saved `Hello.java`. At the command line, type:

```
javac Hello.java
```

After a few moments, you should return to the command prompt. If you see error messages instead, reread the previous section to discover whether you can determine the source of the error.

If the error message indicates that the command was not recognized, make sure that you spelled the `javac` command correctly, including using the correct case. Also, make sure you are using the correct directory or folder where the `Hello.java` file is stored.

If the error message indicates a language error, check your file against Figure 1-10, making sure it matches exactly. Fix any errors, and compile the application again. If errors persist, read through the next section to see if you can discover the solution.

(continues)

(continued)

Correcting Syntax Errors

In this section, you examine error messages and gain firsthand experience with syntax errors.

1. If your version of the `Hello` class did not compile successfully, examine the syntax error messages. Now that you know the messages contain line numbers and carets to pinpoint mistakes, it might be easier for you to fix problems. After you determine the nature of any errors, resave the file and recompile it.
2. Even if your `Hello` class compiled successfully, you need to gain experience with error messages. Your student files contain a file named **HelloErrors.java**. Find this file and open it in your text editor. If you do not have access to the student files that accompany this book, you can type the file yourself, as shown in Figure 1-13.

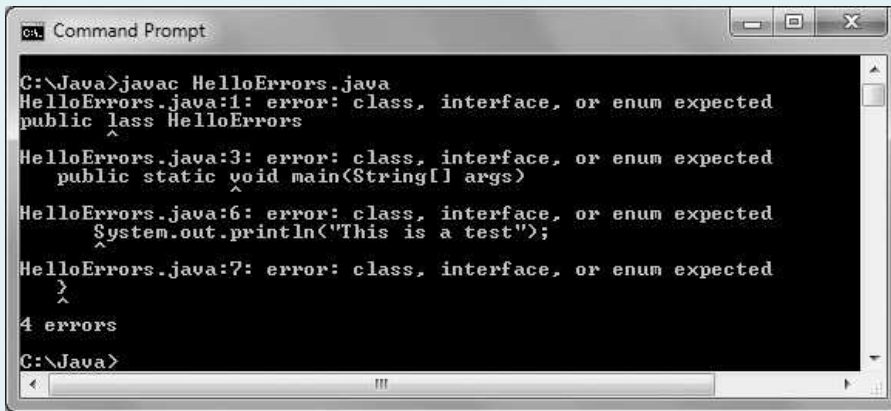
```
public class HelloErrors
{
    public static void main(String[] args)
    {
        System.out.println("Hello");
        System.out.println("This is a test");
    }
}
```

Figure 1-13 The `HelloErrors` class

3. Save the file as **HelloErrors.java** in the folder in which you want to work. Then compile the class using the following command to confirm that it compiles without error:
javac HelloErrors.java
4. In the first line of the file, remove the `c` from `class`, making the first line read **public lass HelloErrors**. Save the file and compile the program. Error messages are generated similar to those shown in Figure 1-14. Even though you changed only one keystroke in the file, four error messages appear. The first indicates that `class`, `enum`, or `interface` is expected in line 1. You haven't learned about the Java keywords `enum` or `interface` yet, but you know that you caused the error by altering the word `class`. The next three errors in lines 3, 6, and 7 show that the compile is continuing to look for one of the three keywords but fails to find them.

(continues)

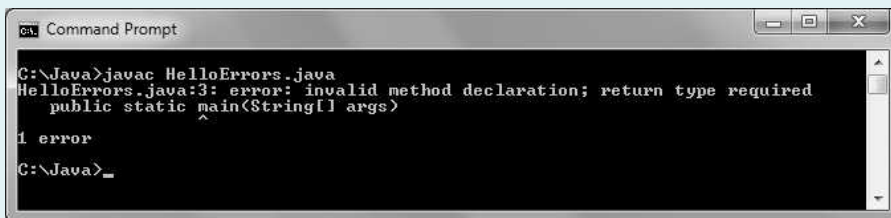
(continued)



```
C:\Java>javac HelloErrors.java
HelloErrors.java:1: error: class, interface, or enum expected
public lass HelloErrors
    ^
HelloErrors.java:3: error: class, interface, or enum expected
    public static void main<String[] args>
                ^
HelloErrors.java:6: error: class, interface, or enum expected
        System.out.println("This is a test");
        ^
HelloErrors.java:7: error: class, interface, or enum expected
    }
    ^
4 errors
C:\Java>
```

Figure 1-14 Error messages generated when `class` is misspelled in the `HelloErrors` program

5. Repair the program by reinserting the `c` in `class`. Save the file and compile it again. The program should compile successfully. When you fix one error, four error messages are removed.
6. Next, remove the word `void` from the third line of the program. Save the file and compile it. Figure 1-15 shows the error message, which indicates that a return type is required. The message does not indicate that `void` is missing because Java supports many return types for methods. In this case, however, `void` is the correct return type, so reinsert it into the correct place in the program, and then save and recompile the file.



```
C:\Java>javac HelloErrors.java
HelloErrors.java:3: error: invalid method declaration; return type required
    public static main<String[] args>
            ^
1 error
C:\Java>
```

Figure 1-15 Error message generated when `void` is omitted from the `main()` method header in the `HelloErrors` program

7. Remove the final closing curly brace from the `HelloErrors` program. Save the file and recompile it. Figure 1-16 shows the generated message “reached end of file while parsing.” **Parsing** is the process the compiler uses to divide your source code into meaningful portions; the message means that the compiler was in the process of analyzing the code when the end of the file was

(continues)

(continued)

encountered prematurely. If you repair the error by reinserting the closing curly brace, saving the file, and recompiling it, you remove the error message.



Figure 1-16 Error message generated when the closing curly brace is omitted from the `HelloErrors` program

8. Continue to introduce errors in the program by misspelling words, omitting punctuation, and adding extraneous keystrokes. Remember to save each program version before you recompile it; otherwise, you will recompile the previous version. When error messages are generated, read them carefully and try to understand their meaning in the context of the error you purposely caused. Occasionally, even though you inserted an error into the program, no error messages will be generated. That does not mean your program is correct. It only means that the program contains no syntax errors. A program can be free of syntax errors but still not be correct, as you will learn in the next section.

Running a Java Application and Correcting Logical Errors

After a program compiles with no syntax errors, you can execute it. However, just because a program compiles and executes, that does not mean the program is error free.

Running a Java Application

To run the `First` application in Figure 1-4 from the command line, you type the following:

```
java First
```

Figure 1-17 shows the application's output in the command window. In this example, you can see that the `First` class is stored in a folder named `Java` on the `C` drive. After you type the `java` command to execute the program, the literal string in the call is output to the `println()` method in the program ("`First Java application`"). Control then returns to the command prompt.



Figure 1-17 Output of the `First` application



The procedure to confirm the storage location of your `First.java` class varies depending on your operating system. In a Windows operating system, for example, you can open Windows Explorer, locate the icon representing the storage device you are using, find the folder in which you have saved the file, and expand the folder. You should see the `First.java` file.

When you run a Java application using the `java` command, do not add the `.class` extension to the filename. If you type `java First`, the interpreter looks for a file named `First.class`. If you type `java First.class`, the interpreter incorrectly looks for a file named `First.class.class`.

Modifying a Compiled Java Class

After viewing the application output, you might decide to modify the class to get a different result. For example, you might decide to change the `First` application's output from `First Java application` to the following:

My new and improved
Java application

To produce the new output, first you must modify the text file that contains the existing class. You need to change the existing literal string, and then add an output statement for another text string. Figure 1-18 shows the class that changes the output.

```
public class First
{
    public static void main(String[] args)
    {
        System.out.println("My new and improved");
        System.out.println("Java application");
    }
}
```

Figure 1-18 First class containing output modified from the original version

The changes to the `First` class include the addition of the statement `System.out.println("My new and improved");` and the removal of the word *First* from the string in the other `println()` statement.

If you make changes to the file, as shown in Figure 1-18, and then save the file and execute the program by typing `java First` at the command line, you will not see the new output—you will see the old output without the added line. Even though you save a text file that contains the modified source code for a class, the class in the already-compiled class file executes. After you save the file named `First.java`, the old compiled version of the class with the same

name is still stored on your computer. Before the new source code can execute, you must do the following:

1. Save the file with the changes (using the same filename).
2. Compile the class with the `javac` command. (Actually, you are *recompiling* the class.)
3. Interpret the class bytecode and execute the class using the `java` command.

Figure 1-19 shows the new output.

When you recompile a class, the original version of the compiled file with the `.class` extension is replaced, and the original version no longer exists. When you modify a class, you must decide whether you want to retain the original version. If you do, you must give the new version a new class name and a new filename.



Figure 1-19 Execution of modified First class



Once in a while, when you make a change to a Java class and then recompile and execute it, the old version still runs. The simplest solution is to delete the `.class` file and compile again. Programmers call this creating a **clean build**.



Watch the video *Compiling and Executing a Program*.

Correcting Logical Errors

Besides syntax errors, a second kind of error occurs when the syntax of the program is correct and the program compiles but produces incorrect results when you execute it. This type of error is a **logic error**, which is often more difficult to find and resolve. For example, Figure 1-20 shows the output of the execution of a successfully compiled program named `FirstBadOutput`. If you glance at the output too quickly, you might not notice that Java is misspelled. The compiler does not find spelling errors within a literal string; it is legitimate to produce any combination of letters as output.

Other examples of logic errors include multiplying two values when you meant to add, printing one copy of a report when you meant to print five, or forgetting to produce a total at the end of a report when a user has requested one. Errors of this type must be detected by carefully examining the program output. It is the responsibility of the program author to test programs and find any logic errors.



Figure 1-20 Output of FirstBadOutput program

You have already learned that syntax errors are compile-time errors. A logic error is a type of **run-time error**—an error not detected until the program asks the computer to do something wrong, or even illegal, while executing. Not all run-time errors are the fault of the programmer. For example, a computer's hardware might fail while a program is executing. However, good programming practices can help to minimize errors.



The process of fixing computer errors has been known as debugging since a large moth was found wedged into the circuitry of a mainframe computer at Harvard University in 1947. See these Web sites for interesting details and pictures: www.jamesshuggins.com/h/tek1/first_computer_bug.htm and www.history.navy.mil/photos/images/h96000/h96566kc.htm.

TWO TRUTHS & A LIE

Running a Java Application and Correcting Logical Errors

1. In Java, if a class is `public`, you must save the class in a file with exactly the same name and a `.java` extension.
2. To compile a file named `MyProgram.java`, you type `java MyProgram`, but to execute the program you type `java MyProgram.java`.
3. When you compile a program, sometimes one error in syntax causes multiple error messages.

The false statement is #2. To compile a file named `MyProgram.java`, you type `javac MyProgram.java`, but to execute the program you type the following:

```
java MyProgram
```

Adding Comments to a Java Class

As you can see, even the simplest Java class requires several lines of code and contains somewhat perplexing syntax. Large applications that perform many tasks include much more code, and as you write larger applications it becomes increasingly difficult to remember why you included steps or how you intended to use particular variables. Documenting your program code helps you remember why you wrote lines of code the way you did. **Program comments** are nonexecuting statements that you add to a program for the purpose of documentation. In other words, comments are designed for people reading the source code and not for the computer executing the program.

Programmers use comments to leave notes for themselves and for others who might read their programs in the future. At the very least, your Java class files should include comments indicating the author, the date, and the class name or function. The best practice dictates that you also include a brief comment to describe the purpose of each method you create within a class.

As you work through this book, add comments as the first lines of every file. The comments should contain the class name and purpose, your name, and the date. Your instructor might ask you to include additional comments.

Comments also can be useful when you are developing an application. If a program is not performing as expected, you can “comment out” various statements and subsequently run the program to observe the effect. When you **comment out** a statement, you turn it into a comment so the compiler does not translate it, and the JVM does not execute its command. This can help you pinpoint the location of errant statements in malfunctioning programs.

There are three types of comments in Java:

- **Line comments** start with two forward slashes (//) and continue to the end of the current line. A line comment can appear on a line by itself or at the end (and to the right) of a line following executable code. Line comments do not require an ending symbol.
- **Block comments** start with a forward slash and an asterisk (/*) and end with an asterisk and a forward slash (*/). A block comment can appear on a line by itself, on a line before executable code, or on a line after executable code. Block comments also can extend across as many lines as needed.
- **Javadoc** comments are a special case of block comments called **documentation comments** because they are used to automatically generate nicely formatted program documentation with a program named javadoc. Javadoc comments begin with a forward slash and two asterisks (/**) and end with an asterisk and a forward slash (*/). Appendix E teaches you how to create javadoc comments.



The forward slash (/) and the backslash (\) characters often are confused, but they are two distinct characters. You cannot use them interchangeably.



The Java Development Kit (JDK) includes the javadoc tool, which you can use when writing programs in Java. The tool produces HTML pages that describe classes and their contents.

Figure 1-21 shows how comments are used in code. In this example, the only statement that executes is the `System.out.println("Hello");` statement; everything else (all the shaded parts) is a comment.

```
// Demonstrating comments
/* This shows
   that these comments
   don't matter */
System.out.println("Hello"); // This line executes
// up to where the comment started
/* Everything but the println()
   is a comment */
```

Figure 1-21 A program segment containing several comments

You might want to create comments simply for aesthetics. For example, you might want to use a comment that is simply a row of dashes or asterisks to use as a visual dividing line between parts of a program.

TWO TRUTHS & A LIE

Adding Comments to a Java Class

1. Line comments start with two forward slashes (//) and end with two backslashes (\\\); they can extend across as many lines as needed.
2. Block comments start with a forward slash and an asterisk (/*) and end with an asterisk and a forward slash (*/); they can extend across as many lines as needed.
3. Javadoc comments begin with a forward slash and two asterisks (/**) and end with an asterisk and a forward slash (*/); they are used to generate documentation with a program named javadoc.

The false statement is #1. Line comments start with two forward slashes (//) and continue to the end of the current line; they do not require an ending symbol.



You Do It

Adding Comments to a Class

In this exercise, you add comments to your Hello.java application and save it as a new class named Hello2 so that you can retain copies of both the original and modified classes.

1. Open the **Hello.java** file you created earlier in this chapter. Enter the following comments at the top of the file, inserting your name and today's date where indicated.

```
// Filename Hello2.java
// Written by <your name>
// Written on <today's date>
```

2. Change the class name to **Hello2**, and then type the following block comment after the class header:

```
/* This class demonstrates the use of the println()
method to print the message Hello, world! */
```

(continues)

(continued)

3. Save the file as **Hello2.java**. The file must be named Hello2.java because the class name is Hello2.
4. Go to the command-line prompt for the drive and folder or subdirectory in which you saved Hello2.java, and type the following command to compile the program:
javac Hello2.java
5. When the compile is successful, execute your application by typing **java Hello2** at the command line. The comments have no effect on program execution; the output should appear on the next line.



After the application compiles successfully, a file named Hello2.class is created and stored in the same folder as the Hello2.java file. If your application compiled without error but you receive an error message, such as “Exception in thread ‘main’ java.lang.NoClassDefFoundError,” when you try to execute the application, you probably do not have your class path set correctly. See Appendix A for details.

Creating a Java Application that Produces GUI Output

Besides allowing you to use the `System` class to produce command window output, Java provides built-in classes that produce GUI output. For example, Java contains a class named `JOptionPane` that allows you to produce dialog boxes. A **dialog box** is a GUI object resembling a window in which you can place messages you want to display. Figure 1-22 shows a class named `FirstDialog`. The `FirstDialog` class contains many elements that are familiar to you; only the two shaded lines are new.

```
import javax.swing.JOptionPane;
public class FirstDialog
{
    public static void main(String[] args)
    {
        JOptionPane.showMessageDialog(null, "First Java dialog");
    }
}
```

Figure 1-22 The `FirstDialog` class



In older versions of Java, any application that used a `JOptionPane` dialog was required to end with a `System.exit(0);` statement or the application would not terminate. You can add this statement to your programs, and they will work correctly, but it is not necessary.

In Figure 1-22, the first shaded line is an **import** statement. You use an **import statement** when you want to access a built-in Java class that is contained in a group of classes called a **package**. To use the `JOptionPane` class, you must import the package named `javax.swing.JOptionPane`. Any import statement you use must be placed outside of any class you write in a file. You will learn more about import statements in general, and the `javax.swing` packages in particular, as you continue to study Java.



You do not need to use an **import** statement when you use the `System` class (as with the `System.out.println()` method) because the `System` class is contained in the package `java.lang`, which is automatically imported in every Java program. You *could* include the statement `import java.lang;` at the top of any file in which you use the `System` class, but you are not required to do so.

The second shaded statement in the `FirstDialog` class in Figure 1-22 uses the `showMessageDialog()` method that is part of the `JOptionPane` class. Like the `println()` method that is used for console output, the `showMessageDialog()` method is followed by a set of parentheses. However, whereas the `println()` method requires only one argument between its parentheses to produce an output string, the `showMessageDialog()` method requires two arguments. (Whenever a method requires multiple arguments, they are separated by commas.) When the first argument to `showMessageDialog()` is `null`, as it is in the class in Figure 1-22, it means the output message box should be placed in the center of the screen. The second argument, after the comma, is the string that is displayed.



Earlier in this chapter, you learned that `true`, `false`, and `null` are all reserved words that represent values.



You will learn more about dialog boxes, including how to position them in different locations and how to add more options to them, in Chapter 2.

When a user executes the `FirstDialog` class, the dialog box in Figure 1-23 is displayed. The user must click the OK button or the Close button to dismiss the dialog box.



Figure 1-23 Output of the `FirstDialog` application

TWO TRUTHS & A LIE

Creating a Java Application that Produces GUI Output

1. A dialog box is a GUI object resembling a window in which you can place messages you want to display.
2. You use an `append` statement when you want to access a built-in Java class that is contained in a group of classes called a package.
3. Different methods can require different numbers of arguments.

The false statement is #2. You use an `import` statement when you want to access a built-in Java class that is contained in a group of classes called a package.



You Do It

Creating a Dialog Box

Next, you write a Java application that produces output in a dialog box.

1. Open a new file in your text editor. Type comments similar to the following, inserting your own name and today's date where indicated.

```
// Filename HelloDialog.java
// Written by <your name>
// Written on <today's date>
```

2. Enter the `import` statement that allows you to use the `JOptionPane` class:

```
import javax.swing.JOptionPane;
```

3. Enter the `HelloDialog` class:

```
public class HelloDialog
{
    public static void main(String[] args)
    {
        JOptionPane.showMessageDialog(null, "Hello, world!");
    }
}
```

(continues)

(continued)

4. Save the file as **HelloDialog.java**. Compile the class using the following command:

```
javac HelloDialog.java
```

If necessary, eliminate any syntax errors, resave the file, and recompile. Then execute the program using the following command:

```
java HelloDialog
```

The output appears as shown in Figure 1-24.



Figure 1-24 Output of HelloDialog application

5. Click **OK** to dismiss the dialog box.

Finding Help

As you write Java programs, you can frequently consult this book as well as other Java documentation. A great wealth of helpful material exists at the Java Web site, www.oracle.com/technetwork/java/index.html. Of particular value is the Java application programming interface, more commonly referred to as the **Java API**. The Java API is also called the Java class library; it contains information about how to use every prewritten Java class, including lists of all the methods you can use with the classes.

Also of interest at the Java Web site are frequently asked questions (**FAQs**) that provide brief answers to many common questions about Java software and products. You can also find several versions of the Java Development Kit (**JDK**) that you can download for free. The JDK is an **SDK**—a software development kit that includes tools used by programmers. Versions are available for Windows, Linux, and Solaris operating systems. You can search and browse documentation online or you can download the documentation file for the JDK and install it on your computer. After it is installed, you can search and browse documentation locally.

A downloadable set of lessons titled “The Java Tutorial” with hundreds of complete working examples is available from <http://docs.oracle.com/javase/tutorial/>. The tutorial is organized into trails—groups of lessons on a particular subject. You can start the tutorial at the beginning and navigate sequentially to the end, or you can jump from one trail to another. As you study each chapter in this book, you are encouraged to make good use of these support materials.



You Do It

Exploring the Java Web Site

In this section, you explore some of the material at the Java Web site.

1. Open an Internet browser and navigate to **www.oracle.com/technetwork/java/index.html**.
2. Oracle could change the layout of its Web site after this book is published. However, you should be able to find and click a link for **Java APIs** and then another for **Java SE 7**. (If you are using an older version of Java, you can select that version instead.)
3. All the Java classes are listed in a panel labeled **All Classes**. Scroll until you can select the **System** class. The largest panel on the page should display details about the `System` class.
4. You can see that the `System` class contains three fields. You are already familiar with the `out` field, and you can see that it is an object of type `PrintStream`. Click the hypertext for the **PrintStream** type to be taken to a new page with details about that class.
5. Scroll through the methods of the `PrintStream` class. Notice that the class contains several versions of the `print()` and `println()` methods. Find the version of the `println()` method that accepts a `String` argument. Click the link to the method to read details about it, such as that it “prints a `String` and then terminates the line.”
6. Many parts of the Java documentation won't mean much to you until you study data types and methods in more detail in the next few chapters of this book. For now, you can explore the Java Web site to get an idea of the wealth of classes that have been created for you.

Don't Do It

At the end of each chapter, a Don't Do It list will alert you to common mistakes made by beginning programmers.

- Don't forget that in Java, a public file's name must match the name of the class it contains. For example, if a file is named `Program1.java`, you can't simply rename it `Program1BackUp.java` and expect it to compile unless you change the class name within the file.

- Don't confuse the terms *parentheses*, *braces*, *brackets*, *curly braces*, *square brackets*, and *angle brackets*. When you are writing a program or performing some other computerized task and someone tells you, "Now, type some braces," you might want to clarify which term is meant. Table 1-5 summarizes these punctuation marks.

| Punctuation | Name | Typical use in Java | Alternate names |
|-------------|-----------------|--|--|
| () | Parentheses | Follows method names as in <code>println()</code> | Parentheses can be called <i>round brackets</i> , but such usage is unusual |
| { } | Curly braces | A pair surrounds a class body, a method body, and a block of code; when you learn about arrays in Chapter 8, you will find that curly braces also surround lists of array values | Curly braces might also be called <i>curly brackets</i> |
| [] | Square brackets | A pair signifies an array; arrays are covered in Chapter 8 | Square brackets might be called <i>box brackets</i> or <i>square braces</i> |
| < > | Angle brackets | A pair of angle brackets surrounds HTML tags, as you will learn in Chapter 17; in Java, a pair also is used with generic arguments in parameterized classes | When angle brackets appear with nothing between them, they are called a <i>chevron</i> |

Table 1-5 Braces and brackets used in Java

- Don't forget to end a block comment. Every `/*` must have a corresponding `*/`, even if it is several lines later. It's harder to make a mistake with line comments (those that start with `//`), but remember that nothing on the line after the `//` will execute.
- Don't forget that Java is case sensitive.
- Don't forget to end every statement with a semicolon, but *not* to end class or method headers with a semicolon.
- Don't forget to recompile a program to which you have made changes. It can be very frustrating to fix an error, run a program, and not understand why you don't see evidence of your changes. The reason might be that the `.class` file does not contain your changes because you forgot to recompile.

- Don't panic when you see a lot of compiler error messages. Often, fixing one will fix several.
- Don't think your program is perfect when all compiler errors are eliminated. Only by running the program multiple times and carefully examining the output can you be assured that your program is logically correct.

Key Terms

A **computer program** is a set of instructions that you write to tell a computer what to do.

Hardware is the general term for computer equipment.

Software is the general term for computer programs.

Application software performs tasks for users.

System software manages the computer.

The **logic** behind any program involves executing the various statements and procedures in the correct order to produce the desired results.

Machine language is a circuitry-level language that represents a series of on and off switches.

Machine code is another term for machine language.

A **low-level programming language** is written to correspond closely to a computer processor's circuitry.

A **high-level programming language** allows you to use an English-like vocabulary to write programs.

Syntax refers to the rules of a language.

Keywords are the words that are part of a programming language.

Program statements are similar to English sentences; they carry out the tasks that programs perform.

Commands are program statements.

A **compiler** is a program that translates language statements into machine code; it translates an entire program at once before any part of the program can execute.

An **interpreter** is a program that translates language statements into machine code; it translates one statement at a time, allowing a program to execute partially.

Executing a statement or program means to carry it out.

At run time is a phrase that describes the period of time during which a program executes.

A **syntax error** is a programming error that occurs when you introduce typing errors into your program or use the programming language incorrectly. A program containing syntax errors will not compile.

Debugging a program is the process that frees it of all errors.

Semantic errors occur when you use a correct word in the wrong context in program code.

Procedural programming is a style of programming in which sets of operations are executed one after another in sequence.

Variables are named computer memory locations that hold values that might vary.

Procedures are sets of operations performed by a computer program.

To **call a procedure** is to temporarily abandon the current logic so that the procedure's commands can execute.

Writing **object-oriented programs** involves creating classes, creating objects from those classes, and creating applications that use those objects. Thinking in an object-oriented manner involves envisioning program components as objects that are similar to concrete objects in the real world; then, you can manipulate the objects to achieve a desired result.

Computer simulations are programs that attempt to mimic real-world activities so that their processes can be improved or so that users can better understand how the real-world processes operate.

Graphical user interfaces, or **GUIs** (pronounced "gooeys"), allow users to interact with a program in a graphical environment.

A **class** is a group or collection of objects with common properties.

A **class definition** describes what attributes its objects will have and what those objects will be able to do.

Attributes are the characteristics that define an object as part of a class.

Properties are attributes of a class.

An **object** is an instance of a class.

An **instance** of a class is an object.

To **instantiate** is to create an instance.

The **state** of an object is the set of values for its attributes.

A **method** is a self-contained block of program code, similar to a procedure.

Encapsulation refers to the enclosure of data and methods within an object.

Inheritance is the ability to create classes that share the attributes and methods of existing classes but with more specific features.

Polymorphism describes the feature of languages that allows the same word to be interpreted correctly in different situations based on the context.

Java was developed by Sun Microsystems as an object-oriented language used both for general-purpose business applications and for interactive, World Wide Web–based Internet applications.

Architecturally neutral describes the feature of Java that allows you to write programs that run on any platform (operating system).

The **Java Virtual Machine (JVM)** is a hypothetical (software-based) computer on which Java runs.

Source code consists of programming statements written in a high-level programming language.

jGRASP is a development environment and source code editor.

A **development environment** is a set of tools that help you write programs by providing such features as displaying a language’s keywords in color.

Bytecode consists of programming statements that have been compiled into binary format.

The **Java interpreter** is a program that checks bytecode and communicates with the operating system, executing the bytecode instructions line by line within the Java Virtual Machine.

“Write once, run anywhere” (WORA) is a slogan developed by Sun Microsystems to describe the ability of one Java program version to work correctly on multiple platforms.

Applets are Java programs that are embedded in a Web page.

Java applications are stand-alone Java programs.

Console applications support character output to a computer screen in a DOS window.

Windowed applications create a graphical user interface (GUI) with elements such as menus, toolbars, and dialog boxes.

A **literal string** is a series of characters that appear exactly as entered. Any literal string in Java appears between double quotation marks.

Arguments are information passed to a method so it can perform its task.

Passing arguments is the act of sending them to a method.

The **standard output device** is normally the monitor.

An **identifier** is a name of a program component such as a class, object, or variable.

Unicode is an international system of character representation.

Pascal casing is a naming convention in which identifiers start with an uppercase letter and use an uppercase letter to start each new word.

Upper camel casing is Pascal casing.

An **access specifier** defines the circumstances under which a class can be accessed and the other classes that have the right to use a class.

The **class body** is the set of data items and methods between the curly braces that follow the class header.

Whitespace is any combination of nonprinting characters, such as spaces, tabs, and carriage returns (blank lines).

The **K & R style** is the indent style in which the opening brace follows the header on the same line; it is named for Kernighan and Ritchie, who wrote the first book on the C programming language.

The **Allman style** is the indent style in which curly braces are aligned and each occupies its own line; it is named for Eric Allman, a programmer who popularized the style.

The keyword **static** means that a method is accessible and usable even though no objects of the class exist.

The keyword **void**, when used in a method header, indicates that the method does not return any value when it is called.

A **compile-time error** is one in which the compiler detects a violation of language syntax rules and is unable to translate the source code to machine code.

Parsing is the process the compiler uses to divide source code into meaningful portions for analysis.

A **clean build** is created when you delete all previously compiled versions of a class before compiling again.

A **logic error** occurs when a program compiles successfully but produces an error during execution.

A **run-time error** occurs when a program compiles successfully but does not execute.

Program comments are nonexecuting statements that you add to a Java file for the purpose of documentation.

To **comment out** a statement is to turn it into a comment so the compiler will not execute its command.

Line comments start with two forward slashes (//) and continue to the end of the current line. Line comments can appear on a line by themselves or at the end of a line following executable code.

Block comments start with a forward slash and an asterisk (/*) and end with an asterisk and a forward slash (*/). Block comments can appear on a line by themselves, on a line before executable code, or on a line after executable code. Block comments also can extend across as many lines as needed.

Javadoc comments are block comments that generate documentation. They begin with a forward slash and two asterisks (`/**`) and end with an asterisk and a forward slash (`*/`).

Documentation comments are comments that automatically generate nicely formatted program documentation.

A **dialog box** is a GUI object resembling a window in which you can place messages you want to display.

An **import statement** accesses a built-in Java class that is contained in a package.

A **package** contains a group of built-in Java classes.

The **Java API** is the application programming interface, a collection of information about how to use every prewritten Java class.

FAQs are frequently asked questions.

The **JDK** is the Java Development Kit.

An **SDK** is a software development kit, or a set of tools useful to programmers.

Chapter Summary

- A computer program is a set of instructions that tells a computer what to do. You can write a program using a high-level programming language, which has its own syntax, or rules of the language. After you write a program, you use a compiler or interpreter to translate the language statements into machine code.
- Writing object-oriented programs involves creating classes, creating objects from those classes, and creating applications—stand-alone executable programs that use those objects. Object-oriented programming languages support encapsulation, inheritance, and polymorphism.
- A program written in Java is run on a standardized hypothetical computer called the Java Virtual Machine (JVM). When your class is compiled into bytecode, an interpreter within the JVM subsequently interprets the bytecode and communicates with your operating system to produce the program results.
- Everything within a Java program must be part of a class and contained within opening and closing curly braces. Methods within classes hold statements. All Java programming statements end with a semicolon. Periods (called dots) are used to separate classes, objects, and methods in program code. All Java applications must have a method named `main()`. Most Java applications have additional methods.
- To compile your source code from the command line, type `javac` followed by the name of the file that contains the source code. The compiler might issue syntax error messages that you must correct. When you successfully compile your source code, the compiler creates a file with a `.class` extension.

- You can run a compiled .class file on any computer that has a Java language interpreter by entering the `java` command followed by the name of the class file. When you modify a class, you must recompile it for the changes to take effect. After a program executes, you must examine the output for logical errors.
- Program comments are nonexecuting statements that you add to a file for the purpose of documentation. Java provides you with three types of comments: line comments, block comments, and javadoc comments.
- Java provides you with built-in classes that produce GUI output. For example, Java contains a class named `JOptionPane` that allows you to produce dialog boxes.

Review Questions

1. The most basic circuitry-level computer language, which consists of on and off switches, is _____.
 - a. a high-level language
 - b. machine language
 - c. Java
 - d. C++
2. Languages that let you use a vocabulary of descriptive terms, such as *read*, *write*, or *add*, are known as _____ languages.
 - a. high-level
 - b. machine
 - c. procedural
 - d. object-oriented
3. The rules of a programming language constitute its _____.
 - a. objects
 - b. logic
 - c. format
 - d. syntax
4. A _____ translates high-level language statements into machine code.
 - a. programmer
 - b. syntax detector
 - c. compiler
 - d. decipherer
5. Named computer memory locations are called _____.
 - a. compilers
 - b. variables
 - c. addresses
 - d. appellations
6. The individual operations used in a computer program are often grouped into logical units called _____.
 - a. procedures
 - b. variables
 - c. constants
 - d. logistics
7. Envisioning program components as objects that are similar to concrete objects in the real world is the hallmark of _____.
 - a. command-line operating systems
 - b. procedural programming
 - c. object-oriented programming
 - d. machine languages

8. The values of an object's attributes also are known as its _____.
 - a. state
 - b. orientation
 - c. methods
 - d. condition
9. An instance of a class is a(n) _____.
 - a. object
 - b. procedure
 - c. method
 - d. class
10. Java is architecturally _____.
 - a. specific
 - b. oriented
 - c. neutral
 - d. abstract
11. You must compile classes written in Java into _____.
 - a. bytecode
 - b. source code
 - c. javadoc statements
 - d. object code
12. All Java programming statements must end with a _____.
 - a. period
 - b. comma
 - c. semicolon
 - d. closing parenthesis
13. Arguments to methods always appear within _____.
 - a. parentheses
 - b. double quotation marks
 - c. single quotation marks
 - d. curly braces
14. In a Java program, you must use _____ to separate classes, objects, and methods.
 - a. commas
 - b. semicolons
 - c. dots
 - d. forward slashes
15. All Java applications must have a method named _____.
 - a. `method()`
 - b. `main()`
 - c. `java()`
 - d. `Hello()`
16. Nonexecuting program statements that provide documentation are called _____.
 - a. classes
 - b. notes
 - c. comments
 - d. commands
17. Java supports three types of comments: _____, _____, and javadoc.
 - a. line, block
 - b. string, literal
 - c. constant, variable
 - d. single, multiple

18. After you write and save a Java application file, you _____ it.
- a. interpret and then compile
 - b. interpret and then execute
 - c. compile and then resave
 - d. compile and then interpret
19. The command to execute a compiled Java application is _____.
- a. run
 - b. execute
 - c. javac
 - d. java
20. You save text files containing Java source code using the file extension _____.
- a. .java
 - b. .class
 - c. .txt
 - d. .src

Exercises



Programming Exercises

1. For each of the following Java identifiers, note whether it is legal or illegal:
 - a. budgetApproval
 - b. German Shepherd
 - c. static
 - d. HELLO
 - e. 212AreaCode
 - f. qhu6TRfg
 - g. ssn#
 - h. 4999
 - i. 17
 - j. Accounts_Receivable
 - k. 32MPG
 - l. rulesOfOrder
2. Name at least three attributes that might be appropriate for each of the following classes:
 - a. CruiseShip
 - b. InsurancePolicy
 - c. StudentAcademicRecord
3. Name at least three objects that are instances of each of the following classes:
 - a. Song
 - b. BaseballTeam
 - c. Playwright
4. Name at least three classes to which each of these objects might belong:
 - a. myRedSweater
 - b. londonEngland
 - c. thursdaysDinner

5. Write, compile, and test a class that displays your full name on the screen. Save the class as **FullName.java**.



As you work through the programming exercises in this book, you will create many files. To organize them, you might want to create a separate folder in which to store the files for each chapter.

6. Write, compile, and test a class that displays your full name, e-mail address, and phone number on three separate lines on the screen. Save the class as **PersonalInfo.java**.
7. Write, compile, and test a class that displays the following pattern on the screen:

```

X               X
X               X
X       XXXXXXXX X
XXXXX  X       X  XXXXX
X  X  X       X  X  X
X  X  X       X  X  X

```

Save the class as **TableAndChairs.java**.

8. Write, compile, and test a class that displays the word “Java” on the screen. Compose each large letter using the appropriate character, as in the following example:

```

      J      A      V      V      A
      J      A A      V      V      A A
      J      A  A      V      V      A  A
J      J      AAAAAA      V V      AAAAAA
JJJJJJ      A      A      V      A      A

```

Save the class as **BigJavaWord.java**.

9. Write, compile, and test a class that displays at least four lines of your favorite song. Save the class as **FavoriteSong.java**.
10. Write, compile, and test a class that uses the command window to display the following statement about comments:

“Program comments are nonexecuting statements you add to a file for the purpose of documentation.”

Also include the same statement in three different comments in the class; each comment should use one of the three different methods of including comments in a Java class. Save the class as **Comments.java**.

11. Modify the Comments.java program in Exercise 10 so that the statement about comments is displayed in a dialog box. Save the class as **CommentsDialog.java**.

12. From 1925 through 1963, Burma Shave advertising signs appeared next to highways all across the United States. There were always four or five signs in a row containing pieces of a rhyme, followed by a final sign that read “Burma Shave.” For example, one set of signs that has been preserved by the Smithsonian Institution reads as follows:

```
Shaving brushes
You'll soon see 'em
On a shelf
In some museum
Burma Shave
```

Find a classic Burma Shave rhyme on the Web. Write, compile, and test a class that produces a series of four dialog boxes so that each displays one line of a Burma Shave slogan in turn. Save the class as **BurmaShave.java**.



Debugging Exercises

- Each of the following files in the Chapter01 folder in your downloadable student files has syntax and/or logic errors. In each case, determine the problem and fix the errors. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, DebugOne1.java will become FixDebugOne1.java.
 - DebugOne1.java
 - DebugOne2.java
 - DebugOne3.java
 - DebugOne4.java



When you change a filename, remember to change every instance of the class name within the file so that it matches the new filename. In Java, the filename and class name must always match.



Game Zone

- In 1952, A. S. Douglas wrote his University of Cambridge Ph.D. dissertation on human-computer interaction and created the first graphical computer game—a version of Tic-Tac-Toe. The game was programmed on an EDSAC vacuum-tube mainframe computer. The first computer game is generally assumed to be “Space-war!”, developed in 1962 at MIT; the first commercially available video game was “Pong,” introduced by Atari in 1973. In 1980, Atari’s “Asteroids” and “Lunar Lander” became the first video games to be registered in the U. S. Copyright Office. Throughout the 1980s, players spent hours with games that now seem very simple and unglamorous; do you recall playing “Adventure,” “Oregon Trail,” “Where in the World Is Carmen Sandiego?,” or “Myst”?

Today, commercial computer games are much more complex; they require many programmers, graphic artists, and testers to develop them, and large management and marketing staffs are needed to promote them. A game might cost many millions

of dollars to develop and market, but a successful game might earn hundreds of millions of dollars. Obviously, with the brief introduction to programming you have had in this chapter, you cannot create a very sophisticated game. However, you can get started.

For games to hold your interest, they almost always include some random, unpredictable behavior. For example, a game in which you shoot asteroids loses some of its fun if the asteroids follow the same, predictable path each time you play the game. Therefore, generating random values is a key component in creating most interesting computer games.

Appendix D contains information on generating random numbers. To fully understand the process, you must learn more about Java classes and methods. However, for now, you can copy the following statement to generate and use a dialog box that displays a random number between 1 and 10:

```
JOptionPane.showMessageDialog(null,"The number is " +  
    (1 + (int)(Math.random() * 10)));
```

Write a Java application that displays two dialog boxes in sequence. The first asks you to think of a number between 1 and 10. The second displays a randomly generated number; the user can see whether his or her guess was accurate. (In future chapters you will improve this game so that the user can enter a guess and the program can determine whether the user was correct. If you wish, you also can tell the user how far off the guess was, whether the guess was high or low, and provide a specific number of repeat attempts.) Save the file as **RandomGuess.java**.



Case Problems

The case problems in this section introduce two fictional businesses. Throughout this book, you will create increasingly complex classes for these businesses that use the newest concepts you have mastered in each chapter.

1. Carly's Catering provides meals for parties and special events. Write a program that displays Carly's motto, which is "Carly's makes the food that makes it a party." Save the file as **CarlysMotto.java**. Create a second program that displays the motto surrounded by a border composed of asterisks. Save the file as **CarlysMotto2.java**.
2. Sammy's Seashore Supplies rents beach equipment such as kayaks, canoes, beach chairs, and umbrellas to tourists. Write a program that displays Sammy's motto, which is "Sammy's makes it fun in the sun." Save the file as **SammysMotto.java**. Create a second program that displays the motto surrounded by a border composed of repeated Ss. Save the file as **SammysMotto2.java**.