A First Simple Program

Now that the basic object-oriented underpinning of Java has been discussed, let's look at some actual Java programs. Let's start by compiling and running the short sample program shown here. As you will see, this involves a little more work than you might imagine.

```
/*
   This is a simple Java program.
   Call this file "Example.java".
*/
class Example {
  // Your program begins with a call to main().
  public static void main(String args[]) {
    System.out.println("This is a simple Java program.");
  }
}
```

Note. *The descriptions that follow use the standard Java 2 SDK (Software Development Kit), which is available from Sun Microsystems. If you are using a different Java development environment, then you may need to follow a different procedure for compiling and executing Java programs. In this case, consult your compiler's documentation for details.*

Entering the Program

For most computer languages, the name of the file that holds the source code to a program is arbitrary. However, this is not the case with Java. The first thing that you must learn about Java is that the name you give to a source file is very important. For this example, the name of the source file should be **Example.java**. Let's see why.

In Java, a source file is officially called a *compilation unit*. It is a text file that contains one or more class definitions. The Java compiler requires that a source file use the **.java** filename extension. Notice that the file extension is four characters long. As you might guess, your operating system must be capable of supporting long filenames. This means that DOS and Windows 3.1 are not capable of supporting Java. However, Windows 95/98 and Windows NT/2000/XP work just fine.

As you can see by looking at the program, the name of the class defined by the program is also **Example**. This is not a coincidence. In Java, all code must reside inside a class. By convention, the name of that class should match the name of the file that holds the program. You should also make sure that the capitalization of the filename matches the class name. The reason for this is that Java is case-sensitive. At this point, the convention that filenames correspond to class names may seem arbitrary. However, this convention makes it easier to maintain and organize your programs.

Compiling the Program

To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

```
C:\>javac Example.java
```

The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program. As discussed earlier, the Java bytecode is the intermediate representation of your program that contains instructions the Java interpreter will execute. Thus, the output of **javac** is not code that can be directly executed.

To actually run the program, you must use the Java interpreter, called **java**. To do so, pass the class name **Example** as a command-line argument, as shown here:

```
C:\>java Example
```

When the program is run, the following output is displayed:

```
This is a simple Java program.
```

When Java source code is compiled, each individual class is put into its own output file named after the class and using the **.class** extension. This is why it is a good idea to give your Java source files the same name as the class they contain—the name of the source file will match the name of the **.class** file. When you execute the Java interpreter as just shown, you are actually specifying the name of the class that you want the interpreter to execute. It will automatically search for a file by that name that has the **.class** extension. If it finds the file, it will execute the code contained in the specified class.

A Closer Look at the First Sample Program

Although **Example.java** is quite short, it includes several key features which are common to all Java programs. Let's closely examine each part of the program.

The program begins with the following lines:

```
/*
   This is a simple Java program.
   Call this file "Example.java".
*/
```

This is a *comment*. Like most other programming languages, Java lets you enter a remark into a program's source file. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code. In this case, the comment describes the program and reminds you that the source file should be called **Example.java**. Of course, in real applications, comments generally explain how some part of the program works or what a specific feature does.

Java supports three styles of comments. The one shown at the top of the program is called a *multiline comment*. This type of comment must begin with /* and end with */. Anything between these two comment symbols is ignored by the compiler. As the name suggests, a multiline comment may be several lines long.

The next line of code in the program is shown here:

```
class Example {
```

This line uses the keyword **class** to declare that a new class is being defined. **Example** is an *identifier* that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}). The use of the curly braces in Java is identical to the way they are used in C, C++, and C#. For the moment, don't worry too much about the details of a class except to note that in Java, all program activity occurs within one. This is one reason why all Java programs are (at least a little bit) object-oriented.

The next line in the program is the *single-line comment*, shown here:

```
// Your program begins with a call to main().
```

This is the second type of comment supported by Java. A *single-line comment* begins with a // and ends at the end of the line. As a general rule, programmers use multiline comments for longer remarks and single-line comments for brief, line-by-line descriptions.

The next line of code is shown here:

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

This line begins the **main( )** method. As the comment preceding it suggests, this is the line at which the program will begin executing. All Java applications begin execution by calling **main( )**. (This is just like C/C++.) The exact meaning of each part of this line cannot be given now, since it involves a detailed understanding of Java's approach to encapsulation. However, since most of the examples in the first part of this book will use this line of code, let's take a brief look at each part now.

The **public** keyword is an *access specifier*, which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared. (The opposite of **public** is **private**, which prevents a member from being used by code defined outside of its class.) In this case, **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started. The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java interpreter before any objects are made. The keyword **void** simply tells the compiler that **main( )** does not return a value. As you will see, methods may also return values. If all this seems a bit confusing, don't worry. All of these concepts will be discussed in detail in subsequent chapters.

As stated, **main( )** is the method called when a Java application begins. Keep in mind that Java is case-sensitive. Thus, **Main** is different from **main**. It is important to understand that the Java compiler will compile classes that do not contain a **main( )** method. But the Java interpreter has no way to run these classes. So, if you had typed **Main** instead of **main**, the compiler would still compile your program. However, the Java interpreter would report an error because it would be unable to find the **main( )** method.

Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called *parameters*. If there are no parameters required for a given method, you still need to include the empty parentheses. In **main( )**, there is only one parameter, albeit a complicated one. **String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String**. (*Arrays* are collections of similar objects.) Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed. This program does not make use of this information, but other programs shown later in this book will.

The last character on the line is the {. This signals the start of **main( )**'s body. All of the code that comprises a method will occur between the method's opening curly brace and its closing curly brace.

One other point: **main( )** is simply a starting place for your program. A complex program will have dozens of classes, only one of which will need to have a **main( )** method to get things started. When you begin creating applets—Java programs that are embedded in Web browsers—you won't use **main( )** at all, since the Web browser uses a different means of starting the execution of applets.

The next line of code is shown here. Notice that it occurs inside **main( )**.

```
System.out.println("This is a simple Java program.");
```

This line outputs the string "This is a simple Java program." followed by a new line on the screen. Output is actually accomplished by the built-in **println( )** method. In this case, **println( )** displays the string which is passed to it. As you will see, **println( )** can be used to display other types of information, too. The line begins with **System.out**. While too complicated to explain in detail at this time, briefly, **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

As you have probably guessed, console output (and input) is not used frequently in real Java programs and applets. Since most modern computing environments are windowed and graphical in nature, console I/O is used mostly for simple, utility programs and for demonstration programs. Later in this book, you will learn other ways to generate output using Java. But for now, we will continue to use the console I/O methods.

Notice that the **println( )** statement ends with a semicolon. All statements in Java end with a semicolon. The reason that the other lines in the program do not end in a semicolon is that they are not, technically, statements.

The first } in the program ends **main( )**, and the last } ends the **Example** class definition.

A Second Short Program

Perhaps no other concept is more fundamental to a programming language than that of a variable. As you probably know, a *variable* is a named memory location that may be assigned a value by your program. The value of a variable may be changed during the execution of the program. The next program shows how a variable is declared and how it is assigned a value. In addition, the program also illustrates some new aspects of console output. As the comments at the top of the program state, you should call this file **Example2.java**.

```
/*
   Here is another short example.
   Call this file "Example2.java".
*/
```

```
class Example2 {
  public static void main(String args[]) {
    int num; // this declares a variable called num

    num = 100; // this assigns num the value 100

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

    num = num * 2;

    System.out.print("The value of num * 2 is ");
    System.out.println(num);
  }
}
```

When you run this program, you will see the following output:

```
This is num: 100
The value of num * 2 is 200
```

Let's take a close look at why this output is generated. The first new line in the program is shown here:

```
int num; // this declares a variable called num
```

This line declares an integer variable called **num**. Java (like most other languages) requires that variables be declared before they are used.

Following is the general form of a variable declaration:

type var-name;

Here, *type* specifies the type of variable being declared, and *var-name* is the name of the variable. If you want to declare more than one variable of the specified type, you may use a comma-separated list of variable names. Java defines several data types, including integer, character, and floating-point. The keyword **int** specifies an integer type.

In the program, the line

```
num = 100; // this assigns num the value 100
```

assigns to **num** the value 100. In Java, the assignment operator is a single equal sign.

The next line of code outputs the value of **num** preceded by the string "This is num:".

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

In this statement, the plus sign causes the value of **num** to be appended to the string that precedes it, and then the resulting string is output. (Actually, **num** is first converted from an integer into its string equivalent and then concatenated with the string that precedes it. This process is described in detail later in this book.) This approach can be generalized. Using the **+** operator, you can string together as many items as you want within a single **println( )** statement.

The next line of code assigns **num** the value of **num** times 2. Like most other languages, Java uses the * operator to indicate multiplication. After this line executes, **num** will contain the value 200.

Here are the next two lines in the program:

```
System.out.print("The value of num * 2 is ");
System.out.println(num);
```

Several new things are occurring here. First, the built-in method **print( )** is used to display the string "The value of num * 2 is ". This string is *not* followed by a newline. This means that when the next output is generated, it will start on the same line. The **print( )** method is just like **println( )**, except that it does not output a newline character after each call. Now look at the call to **println( )**. Notice that **num** is used by itself. Both **print( )** and **println( )** can be used to output values of any of Java's built-in types.