A First Simple Program

Let's start by compiling and running the short sample program shown here:

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

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

You will follow these three steps:

1. Enter the program.
2. Compile the program.
3. Run the program.

Entering the Program

The programs shown in this book are available from McGraw-Hill's Web site: **www.oraclepressbooks.com**. However, if you want to enter the programs by hand, you are free to do so. In this case, you must enter the program into your computer using a text editor, not a word processor. Word processors typically store format information along with text. This format information will confuse the Java compiler. If you are using a Windows platform, you can use WordPad or any other programming editor that you like.

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 (among other things) one or more class definitions. (For now, we will be using source files that contain only one class.) The Java compiler requires that a source file use the **.java** filename extension. 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 the main 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:

```
javac Example.java
```

The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program. Remember, bytecode is not executable code. Bytecode must be executed by a Java Virtual Machine. Thus, the output of **javac** is not code that can be directly executed.

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

```
java Example
```

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

```
Java drives the Web.
```

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.

NOTE

If, when you try to compile the program, the computer cannot find **javac** (and assuming that you have installed the JDK correctly), you may need to specify the path to the command-line tools. In Windows, for example, this means that you will need to add the path to the command-line tools to the paths defined for the **PATH** environmental variable. For example, if JDK 7 was installed in the default directories, then the path to the command-line tools is **C:\Program Files\Java\jdk1.7.0\bin**. (Of course, the specific version of the JDK may differ.) You will need to consult the documentation for your operating system on how to set the path, because this procedure differs between OSes. If you are in a hurry and just want to try the first sample program, you can also fully specify the path to each tool. For example,

```
C:\"Program Files"\Java\jdk1.7.0\bin\javac Example.java
C:\"Program Files"\Java\jdk1.7.0\bin\java Example
```

However, this approach should not be used as a general practice. It is much better to set the path.

The First Sample Program Line by Line

Although **Example.java** is quite short, it includes several key features that 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. As mentioned, the class is Java's basic unit of encapsulation. **Example** is the name of the class. The class definition begins with the opening curly brace ({) and ends with the closing curly brace (}). The elements between the two braces are members of the class. 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:

```
// A Java 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 mentioned earlier, in Java, a subroutine is called a *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( )**. The exact meaning of each part of this line cannot be given now, since it involves a detailed understanding of several other of Java's features. However, since many of the examples in this book will use this line of code, let's take a brief look at each part now.

The **public** keyword is an *access modifier.* An access modifier determines how other parts of the program can access the members of the class. When a class member is preceded by **public**, then that member can 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 before an object of the class has been created. This is necessary because **main( )** is called by the JVM 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. 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 no parameters are required for a given method, you still need to include the empty parentheses. In **main( )** there is only one parameter, **String args[ ]**, which declares a parameter named **args**. This is an array of objects of type **String**. (*Arrays* are collections of similar objects.) Objects of type **String** store sequences of characters. 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 included in a method will occur between the method's opening curly brace and its closing curly brace.

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

```
System.out.println("Java drives the Web.");
```

This line outputs the string "Java drives the Web." 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 that 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. Thus, **System.out** is an object that encapsulates console output. The fact that Java uses an object to define console output is further evidence of its object-oriented nature.

As you have probably guessed, console output (and input) is not used frequently in real-world Java applications. Since most modern computing environments are windowed and graphical in nature, console I/O is used mostly for simple utility programs, for demonstration programs, and for server-side code. 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.