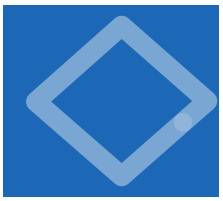


1.1 Your First Program

IN THIS SECTION, OUR PLAN IS to lead you into the world of Java programming by taking you through the basic steps required to get a simple program running. The Java system is a collection of applications, not unlike many of the other applications that you are accustomed to using (such as your word processor, email program, and internet browser). As with any application, you need to be sure that Java is properly installed on your computer. It comes preloaded on many computers, or you can download it easily. You also need a text editor and a terminal application. Your first task is to find the instructions for installing such a Java programming environment on *your* computer by visiting

1.1.1 Hello, World 6
1.1.2 Using a command-line argument . . 8
Programs in this section

```
http://www.cs.princeton.edu/IntroProgramming
```

We refer to this site as the *booksite*. It contains an extensive amount of supplementary information about the material in this book for your reference and use. You will find it useful to have your browser open to this site while programming.

Programming in Java To introduce you to developing Java programs, we break the process down into three steps. To program in Java, you need to:

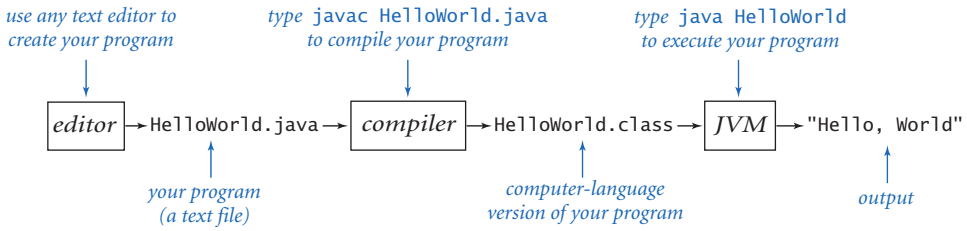
- *Create* a program by typing it into a file named, say, `MyCode.java`.
- *Compile* it by typing `javac MyCode.java` in a terminal window.
- *Run* (or *execute*) it by typing `java MyCode` in the terminal window.

In the first step, you start with a blank screen and end with a sequence of typed characters on the screen, just as when you write an email message or a paper. Programmers use the term *code* to refer to program text and the term *coding* to refer to the act of creating and editing the code. In the second step, you use a system application that *compiles* your program (translates it into a form more suitable for the computer) and puts the result in a file named `MyCode.class`. In the third step, you transfer control of the computer from the system to your program (which returns control back to the system when finished). Many systems have several different ways to create, compile, and execute programs. We choose the sequence described here because it is the simplest to describe and use for simple programs.

Creating a program. A Java program is nothing more than a sequence of characters, like a paragraph or a poem, stored in a file with a `.java` extension. To create one, therefore, you need only define that sequence of characters, in the same way as you do for email or any other computer application. You can use any *text editor* for this task, or you can use one of the more sophisticated program development environments described on the booksite. Such environments are overkill for the sorts of programs we consider in this book, but they are not difficult to use, have many useful features, and are widely used by professionals.

Compiling a program. At first, it might seem that Java is designed to be best understood by the computer. To the contrary, the language is designed to be best understood by the programmer (that's you). The computer's language is far more primitive than Java. A *compiler* is an application that translates a program from the Java language to a language more suitable for executing on the computer. The compiler takes a file with a `.java` extension as input (your program) and produces a file with the same name but with a `.class` extension (the computer-language version). To use your Java compiler, type in a terminal window the `javac` command followed by the file name of the program you want to compile.

Executing a program. Once you compile the program, you can run it. This is the exciting part, where your program takes control of your computer (within the constraints of what the Java system allows). It is perhaps more accurate to say that your computer follows your instructions. It is even more accurate to say that a part of the Java system known as the *Java Virtual Machine* (the *JVM*, for short) directs your computer to follow your instructions. To use the JVM to execute your program, type the `java` command followed by the program name in a terminal window.



Developing a Java program

Program 1.1.1 Hello, World

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

This code is a Java program that accomplishes a simple task. It is traditionally a beginner's first program. The box below shows what happens when you compile and execute the program. The terminal application gives a command prompt (% in this book) and executes the commands that you type (`javac` *and then* `java` *in the example below). The result in this case is that the program prints a message in the terminal window (the third line).*

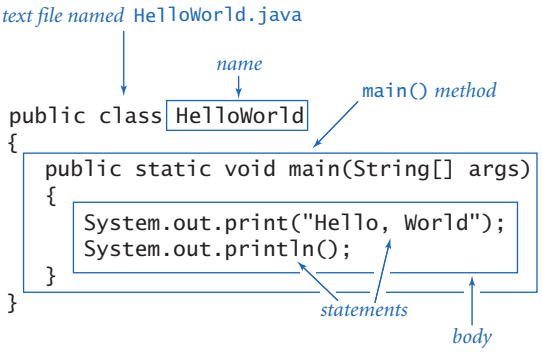
```
% javac HelloWorld.java
% java HelloWorld
Hello, World
```

PROGRAM 1.1.1 IS AN EXAMPLE OF a complete Java program. Its name is `HelloWorld`, which means that its code resides in a file named `HelloWorld.java` (by convention in Java). The program's sole action is to print a message back to the terminal window. For continuity, we will use some standard Java terms to describe the program, but we will not define them until later in the book: PROGRAM 1.1.1 consists of a single *class* named `HelloWorld` that has a single *method* named `main()`. This method uses two other methods named `System.out.print()` and `System.out.println()` to do the job. (When referring to a method in the text, we use `()` after the name to distinguish it from other kinds of names.) Until SECTION 2.1, where we learn about classes that define multiple methods, all of our classes will have this same structure. For the time being, you can think of "class" as meaning "program."

The first line of a method specifies its name and other information; the rest is a sequence of *statements* enclosed in braces and each followed by a semicolon. For the time being, you can think of "programming" as meaning "specifying a class

name and a sequence of statements for its `main()` method." In the next two sections, you will learn many different kinds of statements that you can use to make programs. For the moment, we will just use statements for printing to the terminal like the ones in `HelloWorld`.

When you type `java` followed by a class name in your terminal application, the system calls the `main()` method that you defined in that class, and executes its statements in order, one by one. Thus, typing `java HelloWorld` causes the system to call on the `main()` method in PROGRAM 1.1.1 and execute its two statements. The first statement calls on `System.out.print()` to print in the terminal window the message between the quotation marks, and the second statement calls on `System.out.println()` to terminate the line.



Anatomy of a program

Since the 1970s, it has been a tradition that a beginning programmer's first program should print `"Hello, World"`. So, you should type the code in PROGRAM 1.1.1 into a file, compile it, and execute it. By doing so, you will be following in the footsteps of countless others who have learned how to program. Also, you will be checking that you have a usable editor and terminal application. At first, accomplishing the task of printing something out in a terminal window might not seem very interesting; upon reflection, however, you will see that one of the most basic functions that we need from a program is its ability to tell us what it is doing.

For the time being, all our program code will be just like PROGRAM 1.1.1, except with a different sequence of statements in `main()`. Thus, you do not need to start with a blank page to write a program. Instead, you can

- Copy `HelloWorld.java` into a new file having a new program name of your choice, followed by `.java`.
- Replace `HelloWorld` on the first line with the new program name.
- Replace the `System.out.print()` and `System.out.println()` statements with a different sequence of statements (each ending with a semicolon).

Your program is characterized by its sequence of statements and its name. Each Java program must reside in a file whose name matches the one after the word `class` on the first line, and it also must have a `.java` extension.

Program 1.1.2 Using a command-line argument

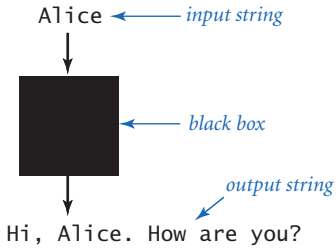
```
public class UseArgument
{
   public static void main(String[] args)
   {
      System.out.print("Hi, ");
      System.out.print(args[0]);
      System.out.println(". How are you?");
   }
}
```

This program shows the way in which we can control the actions of our programs: by providing an argument on the command line. Doing so allows us to tailor the behavior of our programs.

```
% javac UseArgument.java
% java UseArgument Alice
Hi, Alice. How are you?
% java UseArgument Bob
Hi, Bob. How are you?
```

Errors. It is easy to blur the distinction among editing, compiling, and executing programs. You should keep them separate in your mind when you are learning to program, to better understand the effects of the errors that inevitably arise. You can find several examples of errors in the Q&A at the end of this section. You can fix or avoid most errors by carefully examining the program as you create it, the same way you fix spelling and grammatical errors when you compose an email message. Some errors, known as *compile-time* errors, are caught when you compile the program, because they prevent the compiler from doing the translation. Other errors, known as *run-time* errors, do not show up until you execute the program. In general, errors in programs, also commonly known as *bugs*, are the bane of a programmer's existence: the error messages can be confusing or misleading, and the source of the error can be very hard to find. One of the first skills that you will learn is to identify errors; you will also learn to be sufficiently careful when coding, to avoid making many of them in the first place.

Input and Output Typically, we want to provide *input* to our programs: data that they can process to produce a result. The simplest way to provide input data is illustrated in `UseArgument` (PROGRAM 1.1.2). Whenever `UseArgument` is executed, it reads the *command-line argument* that you type after the program name and prints it back out to the terminal as part of the message. The result of executing this program depends on what we type after the program name. After compiling the program once, we can run it for different command-line arguments and get different printed results. We will discuss in more detail the mechanism that we use to pass arguments to our programs later, in SECTION 2.1. In the meantime, you can use `args[0]` within your program's body to represent the string that you type on the command line when it is executed, just as in `UseArgument`.



A bird's-eye view of a Java program

Again, accomplishing the task of getting a program to write back out what we type in to it may not seem interesting at first, but upon reflection you will realize that another basic function of a program is its ability to respond to basic information from the user to control what the program does. The simple model that `UseArgument` represents will suffice to allow us to consider Java's basic programming mechanism and to address all sorts of interesting computational problems.

Stepping back, we can see that `UseArgument` does neither more nor less than implement a function that maps a string of characters (the argument) into another string of characters (the message printed back to the terminal). When using it, we might think of our Java program as a black box that converts our input string to some output string. This model is attractive because it is not only simple but also sufficiently general to allow completion, in principle, of any computational task. For example, the Java compiler itself is nothing more than a program that takes one string of characters as input (a `.java` file) and produces another string of characters as output (the corresponding `.class` file). Later, we will be able to write programs that accomplish a variety of interesting tasks (though we stop short of programs as complicated as a compiler). For the moment, we live with various limitations on the size and type of the input and output to our programs; in SECTION 1.5, we will see how to incorporate more sophisticated mechanisms for program input and output. In particular, we can work with arbitrarily long input and output strings and other types of data such as sound and pictures.

Q&A

Q. Why Java?

A. The programs that we are writing are very similar to their counterparts in several other languages, so our choice of language is not crucial. We use Java because it is widely available, embraces a full set of modern abstractions, and has a variety of automatic checks for mistakes in programs, so it is suitable for learning to program. There is no perfect language, and you certainly will be programming in other languages in the future.

Q. Do I really have to type in the programs in the book to try them out? I believe that you ran them and that they produce the indicated output.

A. Everyone should type in and run `HelloWorld`. Your understanding will be greatly magnified if you also run `UseArgument`, try it on various inputs, and modify it to test different ideas of your own. To save some typing, you can find all of the code in this book (and much more) on the booksite. This site also has information about installing and running Java on your computer, answers to selected exercises, web links, and other extra information that you may find useful or interesting.

Q. What is the meaning of the words `public`, `static` and `void`?

A. These keywords specify certain properties of `main()` that you will learn about later in the book. For the moment, we just include these keywords in the code (because they are required) but do not refer to them in the text.

Q. What is the meaning of the `//`, `/*`, and `*/` character sequences in the code?

A. They denote *comments*, which are ignored by the compiler. A comment is either text in between `/*` and `*/` or at the end of a line after `//`. As with most online code, the code on the booksite is liberally annotated with comments that explain what it does; we use fewer comments in code in this book because the accompanying text and figures provide the explanation.

Q. What are Java's rules regarding tabs, spaces, and newline characters?

A. Such characters are known as *whitespace* characters. Java compilers consider all whitespace in program text to be equivalent. For example, we could write `Hel-`

loWorld as follows:

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

But we do normally adhere to spacing and indenting conventions when we write Java programs, just as we always indent paragraphs and lines consistently when we write prose or poetry.

Q. What are the rules regarding quotation marks?

A. Material inside quotation marks is an exception to the rule defined in the previous question: things within quotes are taken literally so that you can precisely specify what gets printed. If you put any number of successive spaces within the quotes, you get that number of spaces in the output. If you accidentally omit a quotation mark, the compiler may get very confused, because it needs that mark to distinguish between characters in the string and other parts of the program.

Q. What happens when you omit a brace or misspell one of the words, like `public` or `static` or `void` or `main`?

A. It depends upon precisely what you do. Such errors are called *syntax errors* and are usually caught by the compiler. For example, if you make a program `Bad` that is exactly the same as `HelloWorld` except that you omit the line containing the first left brace (and change the program name from `HelloWorld` to `Bad`), you get the following helpful message:

```
% javac Bad.java
Bad.java:2: '{' expected
   public static void main(String[] args)
   ^
1 error
```

From this message, you might correctly surmise that you need to insert a left brace. But the compiler may not be able to tell you exactly what mistake you made, so the error message may be hard to understand. For example, if you omit the second left brace instead of the first one, you get the following messages:

```
% javac Bad.java
Bad.java:4: ';' expected
      System.out.print("Hello, World");
      ^
Bad.java:7: 'class' or 'interface' expected
}
^
Bad.java:8: 'class' or 'interface' expected
^
3 errors
```

One way to get used to such messages is to intentionally introduce mistakes into a simple program and then see what happens. Whatever the error message says, you should treat the compiler as a friend, for it is just trying to tell you that something is wrong with your program.

Q. Can a program use more than one command-line argument?

A. Yes, you can use many arguments, though we normally use just a few. Note that the count starts at 0, so you refer to the first argument as `args[0]`, the second one as `args[1]`, the third one as `args[2]`, and so forth.

Q. What Java methods are available for me to use?

A. There are literally thousands of them. We introduce them to you in a deliberate fashion (starting in the next section) to avoid overwhelming you with choices.

Q. When I ran `UseArgument`, I got a strange error message. What's the problem?

A. Most likely, you forgot to include a command-line argument:

```
% java UseArgument
Hi, Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
        at UseArgument.main(UseArgument.java:6)
```

The JVM is complaining that you ran the program but did not type an argument as promised. You will learn more details about array indices in SECTION 1.4. Remember this error message: you are likely to see it again. Even experienced programmers forget to type arguments on occasion.

Exercises

1.1.1 Write a program that prints the `Hello, World` message 10 times.

1.1.2 Describe what happens if you omit the following in `HelloWorld.java`:

a. `public`
b. `static`
c. `void`
d. `args`

1.1.3 Describe what happens if you misspell (by, say, omitting the second letter) the following in `HelloWorld.java`:

a. `public`
b. `static`
c. `void`
d. `args`

1.1.4 Describe what happens if you try to execute `UseArgument` with each of the following command lines:

a. `java UseArgument java`
b. `java UseArgument @!&^%`
c. `java UseArgument 1234`
d. `java UseArgument.java Bob`
e. `java UseArgument Alice Bob`

1.1.5 Modify `UseArgument.java` to make a program `UseThree.java` that takes three names and prints out a proper sentence with the names in the reverse of the order given, so that, for example, `java UseThree Alice Bob Carol` gives `Hi Carol, Bob, and Alice`.