Chapter 1. Computer Programming

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas, specifically computations. Like engineers, they design things, assembling components into systems and evaluating trade-offs among alternatives. And like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

An important skill for a computer scientist is **problem solving**. It involves the ability to formulate problems, think creatively about solutions, and express solutions clearly and accurately. As it turns out, the process of learning to program computers is an excellent opportunity to develop problem-solving skills. On one level you will be learning to write Java programs, a useful skill by itself. But on another level you will use programming as a means to an end. As we go along, that end will become clearer.

What Is a Computer?

When people hear the word computer, they often think of a desktop or laptop. Not surprisingly, searching for "computer" on *https://images.google.com/* displays rows and rows of these types of machines. However, in a more general sense, a computer can be any type of device that stores and processes data.

Dictionary.com defines a computer as "a programmable electronic device designed to accept data, perform prescribed mathematical and logical operations at high speed, and display the results of these operations. Mainframes, desktop and laptop computers, tablets, and smartphones are some of the different types of computers."

Each type of computer has its own unique design, but internally they all share the same type of **hardware**. The two most important hardware components are **processors** (or CPUs) that perform simple calculations

and **memory** (or RAM) that temporarily stores information. Figure 1-1 shows what these components look like.

Figure 1-1. Example processor and memory hardware.

Figure 1-2. Example processor and memory hardware.

Users generally see and interact with touchscreens, keyboards, and monitors, but it's the processors and memory that perform the actual computation. Nowadays it's fairly standard, even for a smartphone, to have at least eight processors and four gigabytes (four billion cells) of memory.

What Is Programming?

A **program** is a sequence of instructions that specifies how to perform a computation on computer hardware. The computation might be something mathematical, like solving a system of equations or finding the roots of a polynomial. It could also be a symbolic computation, like searching and replacing text in a document or (strangely enough) compiling a program.

The details look different in different languages, but a few basic instructions appear in just about every language:

input:

Get data from the keyboard, a file, a sensor, or some other device.

output:

Display data on the screen, or send data to a file or other device.

math:

Perform basic mathematical operations like addition and division.

decision:

Check for certain conditions and execute the appropriate code.

repetition:

Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of small instructions that look much like these. So you can think of **programming** as the process of breaking down a large, complex task into smaller and smaller subtasks. The process continues until the subtasks are simple enough to be performed with the electronic circuits provided by the hardware.

The Hello World Program

Traditionally, the first program you write when learning a new programming language is called the hello world program. All it does is output the words "Hello, World!" to the screen. In Java, it looks like this:

```
public class Hello {

    public static void main(String[] args) {
        // generate some simple output
        System.out.println("Hello, World!");
    }
}
```

When this program runs it displays:

```
Hello, World!
```

Notice that the output does not include the quotation marks.

Java programs are made up of *class* and *method* definitions, and methods are made up of *statements*. A **statement** is a line of code that performs a basic action. In the hello world program, this line is a **print statement** that displays a message to the user:

```
System.out.println("Hello, World!");
```

`System.out.println` displays results on the screen; the name `println` stands for "print line". Confusingly, *print* can mean both "display on the screen" and "send to the printer". In this book, we'll try to say "display" when we mean output to the screen. Like most statements, the print statement ends with a semicolon (`;`).

Java is “case-sensitive”, which means that uppercase and lowercase are not the same. In the hello world program, `System` has to begin with an uppercase letter; `system` and `SYSTEM` won’t work.

A **method** is a named sequence of statements. This program defines one method named `main`:

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

The name and format of `main` is special: when the program runs, it starts at the first statement in `main` and ends when it finishes the last statement. Later, we will see programs that define more than one method.

This program defines a class named `Hello`. For now, a **class** is a collection of methods; we’ll have more to say about this later. You can give a class any name you like, but it is conventional to start with a capital letter. The name of the class has to match the name of the file it is in, so this class has to be in a file named `Hello.java`.

Java uses curly braces ({ and }) to group things together. In `Hello.java`, the outermost braces contain the class definition, and the inner braces contain the method definition.

The line that begins with two slashes (`//`) is a **comment**, which is a bit of English text that explains the code. When Java sees `//`, it ignores everything from there until the end of the line. Comments have no effect on the execution of the program, but they make it easier for other programmers (and your future self) to understand what you meant to do.

Compiling Java Programs

The programming language you will learn in this book is Java, which is a **high-level language**. Other high-level languages you may have heard of include Python, C and C++, PHP, Ruby, and JavaScript.

Before they can run, programs in high-level languages have to be translated into a **low-level language**, also called “machine language”. This translation takes some time, which is a small disadvantage of high-level languages. But high-level languages have two major advantages:

- It is *much* easier to program in a high-level language. Programs take less time to write, they are shorter and easier to read, and they are more likely to be correct.
- High-level languages are **portable**, meaning they can run on different kinds of computers with few or no modifications. Low-level programs can only run on one kind of computer.

Two kinds of programs translate high-level languages into low-level languages: interpreters and compilers. An **interpreter** reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations. Figure 1-3 shows the structure of an interpreter.

Figure 1-3. How interpreted languages are executed.

In contrast, a **compiler** reads the entire program and translates it completely before the program starts running. In this context, the high-level program is called the **source code**. The translated program is called the **object code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation. As a result, compiled programs often run faster than interpreted programs.

Note that object code, as a low-level language, is not portable. You cannot run an executable compiled for a Windows laptop on an Android phone, for example. In order to run a program on different types of machines, it must be compiled multiple times. It can be difficult to write source code that compiles and runs correctly on different types of machines.

To address this issue, Java is *both* compiled and interpreted. Instead of translating source code directly into an executable, the Java compiler generates code for a **virtual machine**. This “imaginary” machine has the functionality common to desktops, laptops, tablets, phones, etc. Its language, called Java **byte code**, looks like object code and is easy and fast to interpret.

As a result, it’s possible to compile a Java program on one machine, transfer the byte code to another machine, and run the byte code on the

other machine. Figure 1-4 shows the steps of the development process. The Java compiler is a program named `javac`. It translates `.java` files into `.class` files that store the resulting byte code. The Java interpreter is another program, named `java`, which is short for "Java Virtual Machine" (JVM).

Figure 1-4. The process of compiling and running a Java program.

The programmer writes source code in the file `Hello.java` and uses `javac` to compile it. If there are no errors, the compiler saves the byte code in the file `Hello.class`. To run the program, the programmer uses `java` to interpret the byte code. The result of the program is then displayed on the screen.

Although it might seem complicated, these steps are automated for you in most development environments. Usually you only have to press a button or type a single command to compile and interpret your program. On the other hand, it is important to know what steps are happening in the background, so if something goes wrong you can figure out what it is.