Claudio De Sio Cesari

Java for Aliens

learn Java from scratch and become a Pro

Volume 1

1

Introduction to Java

Goals:

At the end of this chapter the reader should:

- ✔ Know some basic computer concepts (Unit 1.1).
- ✔ Know how to define the Java programming language and its features (Unit 1.2).
- ✔ Be able to interact with the development environment: The Java Development Kit (Units 1.3, 1.4, 1.5).
- ✔ Know how to type, compile and run a simple application (Units 1.3, 1.4, 1.5).
- ✔ Choose a development environment (Unit 1.6).

The primary objective of this chapter is to allow the reader to immediately see some initial concrete results. After reading this first chapter, you will have created and executed your first Java program. This is a very simple program, but this will allow you to immediately become familiar with Java programming. To achieve this goal, we will not take anything for granted: we will define the language and the Java technology, describe the development environment and detail a process that will allow us to perform our first application step by step. Finally, you'll learn to understand any error messages and solve our first programming problems.

It is quite probable that, without the basics of programming, it will be hard to understand some concepts, but do not be discouraged. In this first chapter, we have everything that is needed to start from scratch. You don't have to have extra-terrestrial abilities, but you should always be focused on the target, and it is not advisable to skip sentences or even sections. If something is not clear, a re-reading is recommended.

1.1 Prerequisites: Basic IT Concepts

To be able to code with Java, you need to know some basic concepts of computer science. In this book, these concepts are described in order to make programming more approachable. So, even someone who completely lacks the basic concepts to make a computer work can start studying Java without always having to research on Google. For this reason, also in the next chapters you will find discussions on basic concepts. Having a common vocabulary is the first step in order to fully understand this book. We will, however, try to limit the explanations only to the concepts that are really necessary, and we will use the synthesis so that we don't bury the desire to learn under the weight of too much superfluous information.

Let's start from scratch by setting out some definitions that we will use extensively in this book. To be able to program in Java, you need a computer, an electronic device that aggregates a series of physical components that are usually referred to, by the term **hardware**. Examples of hardware are all the interface components with which the user interacts: the mouse, the keyboard, the monitor, the printer, and so on. In a computer, however, there are also many other less "visible" hardware components such as the processor, the motherboard and the RAM memory. However, a computer needs **instructions** to be able to operate and perform operations. These instructions are defined by the **programs**, which are generally referred to by the term **software**. So, a computer is defined by its physical part (hardware) and by the programs that run on it (software). Nowadays there are many types of devices that can be defined as computers. In this book, we assume that the reader will use a **desktop** or a **laptop** computer, not a tablet, smartphone, or other device.

1.1.1 Hardware

In this context, we will distinguish three types of hardware. The first type deals with collecting **input** data and includes devices, such as a **mouse** or a **keyboard**, that accept input from the computer user. The second type has the task of returning **output** data. Examples of components belonging to this second type are **monitors** and **printers**. These two types are to be considered "interface hardware" and are those with which the user interacts. For this reason, you should already have a basic understanding of these devices. The type of hardware that we will be most interested in, is that with which we do not interact directly: the "inside the box" hardware. Before talking about programs, you need to have at least an idea of how a computer stores data, that is, for example, how the memory and the processor work.

The **processor** or **CPU** ("**Central Processing Unit**"), is the hardware component that executes the instructions of a program. Its main task is to perform elementary arithmetic operations, and to manage the shifting operations of the data in the memory. It may seem strange, but it is the processor that makes everything that we do with a computer possible.

The **memory** preserves the data that is used by the programs and there are two types: a main

memory and an auxiliary one. The **main memory** is usually called **RAM** ("**Random Access Memory**"). This is a memory that contains the programs that are running, along with their data. This memory is temporary (it is also called *volatile*) and is deleted when the computer is turned off. The **auxiliary memory** (also called **secondary memory**), on the other hand, preserves data even when the computer is switched off. The auxiliary memory is therefore usually much larger than the main one, but is also less efficient from the point of view of data storage speed. Examples of auxiliary memory are hard drives and other devices such as solid-state memories, USB sticks, etc.

1.1.2 Software

To use a computer, we need programs that contain instructions to be executed by the underlying hardware. In particular, there is at least one main program that manages the whole computer, which is often identified as being the computer itself: the **operating system**. Nowadays, the most famous operating systems are Microsoft Windows systems, Apple Mac operating systems, and Linux and Unix systems. When we turn on a computer, it starts its operating system and this manages the entire computer. The operating system, among other things, makes it possible to install and launch other programs. For example, the Microsoft Windows 10 operating system, allows us to install and launch programs such as Microsoft Word, a browser such as Mozilla Firefox or Google Chrome, a Java development IDE such as Netbeans or Eclipse, a media player like VLC, or our own program written in Java.
There are thousands of types of programs, very different from each other. For example, a *browser* is a program that allows us to surf the internet. An *IDE* is a programming environment that integrates different tools ("IDE" stands for "Integrated Development Environment"). A *media player* allows us to view videos or listen to audio. So, each program executes instructions that very often rely on input data, that perform certain activities and that produce a result as output. For example, a browser can take as input an Internet address entered by the user using the keyboard. It can perform a process that includes identifying the requested resource online, retrieving the code that represents the web page and interpreting the languages and technologies that are used in the page code. Finally, it can output the page formatted as the user sees it.

1.1.3 Programming Languages

A **programming language** facilitates the communication of instructions to the computer so that it executes these instructions in the context of a program. It is just a language with its own vocabulary and rules. The vocabulary is almost always quite limited, but usually there are a lot of rules.

The set of rules is sometimes called "language grammar".

There are many programming languages and their history is often fascinating. Languages like Assembly, Fortran, Basic, COBOL, Pascal, C, C++, LISP, Prolog, SmallTalk, Visual Basic, Javascript, and of course Java, are just some of the languages that are part of the history of programming (the full list would be very long). The question arises: can a computer understand all of these languages? Obviously not! The hardware of a computer is able to understand a single language, called **machine language**. Machine language has a vocabulary defined by the **binary numeral system**, and therefore formed by only two symbols: `0` and `1`. By arranging these symbols in sequence, we can obtain "words", and with these words we can create instructions to be performed by the processor.

Since there are different types of processors, it is technically incorrect to speak of the existence of one single machine language. Indeed, there are many. Each processor can interpret a particular machine language.

So how can we create a computer program without knowing which processor it will be executed on and therefore without knowing which machine language must be used? And how does a computer understand a program written in any programming language when it is only capable of understanding machine language? It seems that we require a "translator" and, in fact, that's exactly what we use.

Modern languages like Java are called **high-level languages**, that is, languages that are very similar to the language that we use every day to communicate, and consequently quite different from machine languages (which can only use symbols `0` and `1` to compose instructions). On the other hand, languages like Assembly, that are very similar to machine languages, are called **low-level languages**. But programs written with both Assembly and with Java require a machine language translation to be understood by the processor. This translation is made by a software that is part of the language programming environment, called **compiler**. A compiler is therefore a software that translates instructions written with a programming language into instructions written with a machine language.

The process of developing a program includes the following steps:

1. The developer writes a program using a programming language. By correctly following the language rules, he saves the instructions within one or more text files. We say that these files contain the **source code** of the program, and the text files themselves are called **source files**.
2. The developer uses a compiler to which will pass as input the source files. After having checked the correctness of the source code, the compiler will translate the instructions

from the source code into instructions written in machine language. To do this, the compiler will create the so-called **binary files**. The binary files are directly executable by the processor each time they are launched, and represent the program itself. Programs ready to use are also called **executable files**, or also **object files**.

Languages that have the support of a compiler are called **compiled languages**.
But there are also languages that do not use a compiler to translate the source code into machine language but, rather, use a software that is called **interpreter**. Unlike a compiler, an interpreter is a tool that can launch the program, translating on the fly the source code into object code, instruction after instruction. A language that uses an interpreter instead of a compiler is called an **interpreted language**. An interpreter can represent a more flexible solution within the dynamics of development, allowing us to alternate the writing of the code and its execution without having to perform the compilation phase. But unfortunately, it suffers from an obvious disadvantage. The translation of the source code into machine code is included in the execution phase of the program. This is inevitably slower than the execution of a program that has already been compiled. Furthermore, this alternation between translation and execution occurs every time the program is interpreted.
Each programming language requires a different compiler or interpreter, depending on the hardware where the program is to be executed. In many languages, this can result in different behaviours for programs launched on different computers and generated from the same source. This problem, however, is brilliantly solved by Java as we'll soon see.

1.2 Introduction to Java

Having introduced some basic concepts of computer science, now is the time for a brief introduction to Java. But we will not talk about the heroic deeds of the creators of Java, nor about the legends of the past and predictions for the future. In fact, several topics will not feature in these two volumes, because they have been moved into special online appendices to be downloaded. Among these appendices, you will also find one relating to the creation and history of Java: Appendix A. If you are curious to know why the language is called Java, who the creators are and what the conditions are that determined the birth and success of Java, etc., you should read Appendix A. Further you can find important info on the new licence and support model adopted by Oracle. In these pages, instead, we will try to lead straight to our goal, without boring those who cannot wait to get their hands on the code.

To limit the number of pages and consequently the cost of this book, a lot of material has been moved online! You can download all of the appendices, about 600 pages of exercises, all of the code . . .

. . . presented in the book and the errata file by visiting the address http://www.javaforaliens.com**. From here you can download all of the material.**

1.2.1 What is Java?

With the term "Java", we usually refer to:

1. The most used programming language on the planet.
2. A technology that includes several "sub-technologies" that have established themselves in different areas of software use. Nowadays, Java technology is the most used on enterprise applications, and there are billions of electronic devices all over the world that use Java technology: smartphones, SCADA, mobile phones, satellites, decoders, smart cards, robots that roam on Mars, and so on.

In this book, we will focus on the Java programming language which is the starting point for accessing all Java technologies. We will mention and introduce some of them, but just enough to arouse curiosity. Then anyone who wants to further study these technologies, can do it later with other didactic resources.

1.2.2 Java Language Features

The language was presented in 1995 by Sun Microsystems, a historic and glorious American company that since 2010 has been acquired by Oracle.
The main features of Java are:

- **Syntax**: it is similar to that of other historic programming languages such as C and C++. Other languages with very similar syntaxes are languages that were born after Java, such as C# and JavaScript. This means that if the reader already has knowledge of one of these languages, he will have a significant advantage in this first part of the study. However, it is a syntax considered to be among the clearest in comparison to those of other programming languages, and is characterized by a high readability (sometimes it seems like writing in a 'natural' language).
- **Robustness**: a language is more robust than another when it has a greater ability to prevent programming errors, and Java is one of the most robust languages around. It defines a clear and functioning exception management, and has an automatic mechanism for memory management (Garbage Collection) that exempts the programmer from having

to de-allocate memory when needed (one of the most delicate points in programming). Furthermore, the Java compiler is very "severe". The programmer is actually forced to solve all "unclear" situations, guaranteeing the program a better chance to work properly. The logic is: "it is much better to get an error at compile-time than at execution-time".

- **Platform Independence**: thanks to the Java Virtual Machine concept, every application, once compiled, can be executed on any platform (for example a PC with a Windows operating system or a Unix workstation). This is definitely the most important feature of Java. In fact, if you have to implement a program designed to run on different platforms, there will be no need to create different specific versions. This is obviously a huge advantage.

- **Java Virtual Machine**: what makes the platform independence real, is the **Java Virtual Machine** ("**JVM**"). This is software capable of playing the role of interpreter. Java can actually be considered both a compiled and interpreted language. In fact, after writing our source code with Java, we will have to use the Java compiler first to compile it. We will not directly produce, however, an executable file containing instructions in machine language, but a file that contains the translation of our code in a language very similar to machine language called **bytecode**. Once this file is obtained, the programmer can pass as input it to the JVM which will interpret the bytecode and our program will be finally executed. So, if a platform has a Java Virtual Machine, this will be enough to execute bytecode. This is the case when it comes to the most widespread mobile operating system (Android by Google), where the most of the native applications are written in Java. That's why there are so many different models of smartphones and tablets (and also cookers, washing machines, refrigerators, robots, etc.) that use the Android operating system: the secret is the virtual machine. After translating our source code into bytecode, our program is potentially executable on many types of devices. The Java Virtual Machine is called a "virtual machine" because this software has been implemented to simulate a hardware. It could be said that "the machine language is to a computer as the bytecode is to a Java Virtual Machine". In addition to allowing platform independence, JVM provides Java with many other advantages. For example, it is a multi-threaded language (usually a feature of operating systems), that is, capable of executing several processes in parallel mode. Moreover, it guarantees very powerful security mechanisms, the "supervision" of the code by the Garbage Collector for the automatic management of memory, and much more.

- **Object-Oriented**: object orientation is a fundamental programming style that allows us to program in a more "natural" way. We will talk extensively about this topic from the sixth chapter onwards. In fact, Java provides us with some tools that practically "force us"

to program with objects. The fundamental paradigms of object-oriented programming (inheritance, encapsulation, polymorphism) are more easily appreciated and comprehensible. Java is clearer and more schematic than any other object-oriented language. If you learn Java, later you can certainly access the knowledge of other object-oriented languages in a more natural way, since you will have an object-oriented way of thinking about programming. However, as the years have passed, the language has evolved, and it has also opened up to other programming paradigms.

- **Ease of Development**: before the release of Version 5, Java was advertised as a "simple to learn" language. Since the release of Version 5, the term "simplicity" has been replaced with "ease of development". In fact, with the introduction of revolutionary characteristics such as generic and annotation types, Java has created for itself a clearly outlined path: that of allowing the language to evolve so that it keeps pace with modern developments, even at the expense of simplicity. The language today has become more difficult to learn, but programs should be easier to write. In short, learning Java will not be simple, but once we have understood the basic logic, we will have in our hands a very powerful tool.
- **Library and Standardization**: Java has a huge, standard and well documented *software library*. So we can use in our code pieces of already working software. This makes Java a *high level language*, and also allows newbies to create complex applications in a fairly short time. For example, it is relatively easy to create graphical interfaces, database links, and network connections regardless of the computer on which the application is developed. Moreover, thanks to Oracle's specifications, standardization problems as compilers that produce different outputs on different platforms, will not exist.
- **Open**: using Java does not mean moving in a closed environment where everything is standard. From the choice of the development tool (Eclipse, Netbeans, etc.) to the interaction with other languages, external libraries and technologies (SQL, XML, open source framework, etc.) you'll never stop learning! The Java code is also *open source*, which means that it is possible to read the source code. Since its inception, Java has always been considered as an "open" language, and interaction with its developers has always been a strong point of its evolution. The reference site today is **http://openjdk.java.net** and it is also possible to actively contribute to the development of Java with tests, proposals and bug fixing. The new features of the latest versions have all been proposed by developers who have contributed with ideas. Java is made by developers for developers.

There are many other features that could be mentioned, but these are the ones that are most relevant for readers of this book.

1.2.3 False Beliefs About Java

The following are a series of gossips, prejudices, misconceptions and inaccurate or at least superficial beliefs, concerning the Java language:

- **Java is slow**: in the early years, Java was a slow language compared to other compiled languages. The performance level of the bytecode interpretation by the JVM was not optimized and the slowness was clear. But a few years after Java was created, Sun Microsystems created its first **HotSpot Performance Engine-based JVM**, which became the default JVM from Java Version 1.3 onwards. This is a JVM with **Just-In-Time** (**JIT**) compilation that, during execution, continuously analyzes the program, identifies the parts of the code that are executed more often (calls precisely **hot spots**) and translates them into machine language, optimizing translation as much as possible. Furthermore, the Java compiler also performs many optimizations when it creates bytecode. Today, after about twenty years of evolution, the JVM has performance levels that are comparable to those of the most performing compiled languages such as C and C ++ which, according to some benchmarks, are even exceeded.

  

  Oracle is focusing on testing a new type of JVM based on **Ahead-Of-Time** (**AOT**) compilation. AOT aims to significantly reduce the time of compilation on the fly (JIT), especially in the application startup phase. Oracle has therefore released, with Java 9, **GRAAL** for the 64-bit Linux platform, which is a JIT compiler written in Java, that will allow us to test the feasibility of creating a future AOT compiler written in Java. So, in the future, we could have even better performance levels.

- **Java is not sure**: in the early years there was a sort of competition to find how critical some Java technologies were, such as Applets. This received a lot of publicity, but Sun actually resolved it quite quickly. Even more emphasis was put on some bugs found in the Java Plug-in in the early '10s, so as to remove them from the most important browsers in a matter of a few months. That was nothing compared to the damage caused throughout those years by viruses and worms distributed with programs executable on Windows. Actually, Java is among the most important languages, and undoubtedly the safest, especially after the giant steps made in recent years by Oracle. An ironic video was created about that strange period to publicize the 2013 JavaZone conference in Oslo. You can find it here: http://www.javaforaliens.com/ext/javapocalypse.

- **Java is verbose**: undoubtedly, compared to other languages, Java is a verbose language. But it was worse at the beginning. It is also true that in recent years, a significant part of the evolution of the language has focused on making its syntax more concise. Multi-catch, try with resources, factory method for collections, Lambda expressions, streams

and pipelines, and the local variable types inference are just some of the latest changes that make Java a more modern language. Unfortunately, often the critical nature of some projects, means that developers get stuck on old versions of Java, and are not being able to upgrade to take advantage of the new features.

- **Java is easy to learn**: once Java was really easy to learn. Today, however, it is a complex language, and some features are really difficult to learn. So, Java is not just for aliens, but to study it will not be that simple.
- **You must pay to use Java**: after Java 8, it is true that Oracle has introduced a new payment policy to support of programs running on the Oracle JDK in production, but it is also true that you can download the OpenJDK, which is aligned with the Oracle JDK. The "price to pay" to get the security patches, is to update the JDK every time a new version comes out. You can also choose other JDKs such as those that offer AdoptOpenJDK and Azul (as well as many others), that also propose different solutions for support. So, the response is no, Java is still a language that can be used for free.
- **Javascript is a simplified version of Java**: actually, apart from the name (which was chosen because at the time Java was very popular) and a quite similar syntax, Javascript has nothing to do with Java. Javascript is a scripting language that runs on web pages, thanks to the fact that each browser implements an interpreter. It is an **untyped language** (as opposed to Java which is **strongly typed one**) and does not have a compiler.

1.3 Development Environment

In this book, we will assume that the reader uses a computer on which a Windows 7 operating system or higher is installed (recommended Windows 10), but obviously, it is possible to program on other platforms such as Linux or Mac.

To write a Java program we need two pieces of software:

1. **A program that allows us to write the Java code**. A simple text editor such as Windows Notepad can be a good start. We do not recommend WordPad, Word or any other editor that handles styles, formatting, etc.
2. **The Java Development Kit Standard Edition** ("**JDK**"). This is the official development environment of Java. It's a suite of software including a compiler and the Java Virtual

Machine we mentioned in the previous section. The version 13 of the JDK can be downloaded for free at this address: **http://jdk.java.net/13 choosing the link of the** choosing the link indicating the reference platform, in the **Builds** section (see Figure 1.1). In Appendix B (available for download at **http://www.javaforaliens.com**) we describe the process to be followed to obtain and correctly install the development environment on a Windows operating system.

Other alternative development environments will be discussed at the end of the chapter.

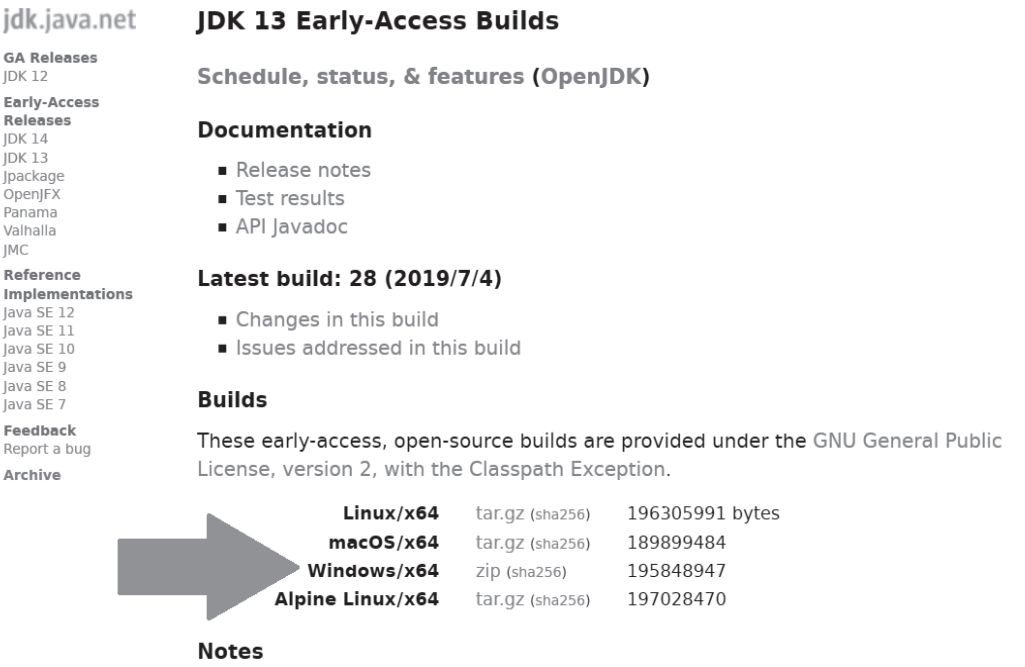

Figure 1.1 – JDK page.

1.3.1 Structure of the JDK

Once the JDK is installed correctly, it should be placed in the **C:\Program Files\Java** folder. With the introduction of version 9, the folder structure of the JDK has been radically changed. Compared to the previous version, some folders have been removed, while others have been added. Below is a brief explanation of

the current folders:

- **bin**: contains all the executable files of the JDK, for example **javac**, **java**, **jar**, **jlink**, **javadoc** and others, which will allow us to compile, execute, create file archives, etc.
- **conf**: contains configuration files used by the JDK.
- **include** and **lib**: contain libraries written in C and Java that are used by the JDK. With the term "library", we mean a software that makes it possible for other software (in our case the JDK) to use the features it defines.
- **legal**: contains the JDK license files.
- **jmods**: contains the standard Java library reorganized into *modules*. The most important novelty introduced by JDK version 9, concerns the introduction of this new way of organizing our programs: *modularization*. We will see how to use the modules in a dedicated chapter in the second volume of this book.

Up to Java 10, in addition to the JDK, the Java Runtime Environment (JRE) could also be installed separately. From Version 11 onwards, the JRE is included within the JDK and is no longer downloadable separately. But for a Java application to be executable on a particular machine, as long as only the JRE is installed, there is no need for the entire JDK. With the introduction of the **jlink** tool in Version 9 of the JDK, we can create optimized JRE versions, to be distributed together with our programs. So, in order for a Java program to run on a certain computer, we will not even need to install the JRE anymore as it will be distributed with the application itself.

Let's summarize the difference between JVM, JRE and JDK:

The JVM is the Java Virtual Machine: the software that simulates hardware capable of interpreting and executing the bytecode contained in a compiled Java file.

The JRE is the Java Runtime Environment and consists of software that provides the environment to use the JVM (and, in fact, contains it). To be able to execute Java code, a computer needs the JRE only.

The JDK is the Java Development Kit, the software we need to develop Java programs, which contains a suite of tools that support development. So, the JDK is used to develop Java code, while the JRE is used to run Java programs. But the JDK contains the JRE, and thus . . .

. . . can be used to run Java applications too. In addition, with the JDK jlink **tool, we can create optimized JREs to be distributed with our applications, which will then no longer require a separate JRE installation, as previously.**

1.3.2 Step by Step Developing Guide

After having correctly installed the JDK and appropriately set any environment variables (see Appendix B), we will be ready to write our first application (see next section).

If you are using a Windows operating system, our recommendation is to enable the display of all the file extensions from the Folder Options**, as programmers must always distinguish different types of files. Unfortunately, the default setting of the Windows operating system requires that the extension for the known file types be hidden. To fix this, you have to: open the** Control Panel**, select the item** File Explorer Options **(on older Windows systems, select the item** Browse Folder**), select the** View **tab and uncheck the checkbox** Hide extensions for known file types**. Then the extensions of every single file on the hard disk will be visible.**

In general, every time we develop, we will have to perform the following steps:

1. **Code writing**: we will write the source code of our application using an editor. As previously stated, we recommend using Windows Notepad for this initial test.
2. **Saving phase**: with the Notepad application, we will save our file with the extension **.java** in the secondary memory, usually on a hard drive.
3. **Opening a Command Prompt**: after saving our Java file, we need to open a **command prompt window** (also called **DOS prompt**).

If you do not find the link to open the command prompt, you can type the cmd **command in the search field of the Windows start menu and press the** Enter **key. If you are not familiar with the DOS operating system (the Microsoft original operating system that evolved in early versions of Windows), in Appendix C, you can find a short tutorial that will allow you to operate in this environment.**

4. **Positioning**: within this prompt, we must move to the folder where our source file was saved. See Appendix C if you're not able to do it.
5. **Compiling phase**: type the compile command:

```
javac fileName.java
```

and press the **Enter** key to execute it. If the compilation is successful, a file named **fileName.class** will be created, in the same folder where the file **fileName.java** exists. The **.class** file, as we have already said, will contain the translation in bytecode of the source file.

6. **Execution phase**: at this point, we can run the program, invoking the interpretation of the Java Virtual Machine. Just write the following command in the DOS prompt:

```
java fileName
```

(without suffixes). Press the **Enter** key to confirm.

The application, unless there are errors in the code, will be performed by the JVM. In particular, the **class loader** of the Java Virtual Machine will ensure that the bytecode present in the **.class** file is loaded into the main memory (the RAM memory) to be executed. Then the **bytecode verifier** of the JVM will verify that the code does not violate the security restrictions imposed by Java and does not damage the computer (as a virus could do). Finally, through the **JIT compilation** ("Just In Time"), the JVM will silently execute a second compilation phase, parallel to the interpretation of the bytecode. In fact, it will identify the parts of bytecode that are used several times (called **hot spots**) and translate them into machine language. These parts must no longer be interpreted because they have already been translated into machine code. This behavior improves the performance of our program.
Now we are ready to write our first Java program.

1.4 First Approach to the Code

Let's take a look at the classic "Hello World" application. This is the typical first program that is written when learning a new programming language. In practice, it is a simple program that prints the phrase "Hello World!". In this way, we will begin to familiarize ourselves with the syntax and with some fundamental concepts such as that of *class* and *method*.

Note that, inevitably, some points will remain obscure, and that therefore some parts of the code will have to be taken for granted. This is the didactic approach that this book will follow: instead of trying to explain some concepts prematurely, we will come back to them in later chapters.

We will also see how to compile and run our simple application. As already mentioned, its goal is to print the message "Hello World!" on the screen (in the same command prompt).
The code is the following:

```
1 public class HelloWorld
2 {
3     public static void main(String args[])
4     {
5         System.out.println("Hello World!");
6     }
7 }
```

Line numbers are not part of the application (you do not have to write them) but they will be useful for our analysis.

This program must be written using the Notepad application. Then it must be saved in a file named exactly as the class that is declared (also paying attention to capitals and lowercase letters) and with the **.java** extension. The name of the file that will contain the Java code just presented must therefore be **HelloWorld.java**.
We do not recommend a "copy - paste" of the code. At least for the first few times, try to write as much code as possible. We recommend writing the program with Notepad line by line after reading the following analysis. In this way you will become more aware of what you're writing.

1.4.1 Analysis of the HelloWorld Program

Let's analyze the previous code line by line.

Line 1:

```
1 public class HelloWorld
```

This is the declaration of a **class** called `HelloWorld`. As we will see, each Java application consists of classes. The concept of class will be detailed in the next section. For now, we only need to understand that all operational programming instructions will always be contained in classes (or other similar programming elements). We must also keep in mind that programs consisting of a single class (and therefore of a single file), are usually sample programs (like the one we have just presented). In the real-world, Java programs are frequently composed of several classes, each declared in its own file. Each class has a visibility that determines its usability by other classes. The `HelloWorld` class has been declared **publicly accessible** (the freest degree

of accessibility) using the `public` modifier declaration. In Java, a modifier characterizes a class, just as in a natural language an adjective can characterize a noun. In this case, it can therefore be said that the `HelloWorld` class is public. This topic will also be dealt with in detail in the next chapters.

Line 2:

```
2 {
```

This open brace indicates the beginning of the `HelloWorld` class, which will close on line 7 with a closed brace. The block of code included in these two braces defines the `HelloWorld` class.

Line 3:

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

This line should be memorized immediately since it must be defined in every Java application. This is the declaration of the `main()` method. In Java, the term "method" is synonymous with "action" (methods will be discussed in detail in the next chapter). In particular, the `main()` method defines the starting point for the execution of each program. The first instruction that will then be executed at runtime by the program, will be the one that the JVM will find immediately after opening the block of code that defines this method.
In addition to the word `main`, line 3 contains other words which we will study in detail in the next chapters. Unfortunately, as already stated, when you start to study an object-oriented language like Java, it is impossible to open a discussion without opening many others. For now, the reader will have to settle for the following table:

Word	Explanation
`public`	This is a **method modifier**. Modifiers are used in Java as adjectives are used in human languages. If you place a modifier before the declaration of a Java element (a method, a class, etc.) this will change its properties (depending on the meaning of the modifier). In this case, `public` represents an *access specifier* that actually makes the method accessible, even outside the class in which it has been defined.
`static`	This is another **method modifier**. We can mark a Java element with multiple modifiers. The definition of `static` is quite complex. For now, it must suffice that this modifier is essential for the definition of the `main()` method.

`void`	It is the **method return type**, which is the type of data that this method must return as output. The word "void" should be interpreted as "empty", and this implies that this method does not return any type of value. The `main()` method must always have `void` as return type.
`main`	This is the **name** of the method, also called the **identifier** of the method.
`(String args[])`	To the right of the identifier of a method, we always define a pair of round brackets that optionally encloses a **list of parameters** (also called **method arguments**) that represent the input of the method. The `main()` method always requires an array of type `String` as a parameter (there will be an entire section dedicated to arrays in the third chapter). Note that `args` is the identifier of this array and is the only word that can be changed in the definition of the `main()` method, even if the word `args` is always used by convention.

Line 4:

```
4     {
```

This brace indicates the beginning of the `main()` method that will end at line 6 with a closed brace. The block of code between these two braces defines the method.

Line 5:

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

This command will print the string “Hello World!”. Since we are introducing topics for which the reader is not yet ready, we will postpone a detailed explanation of this command to later chapters. For now, it is sufficient to say that we are invoking a method called `println()` belonging to the standard Java library, passing the string to be printed to it as a parameter. This method takes the string “Hello World” parameter and prints it on the screen.

Line 6:

```
6     }
```

This closed brace closes the last one opened, that is, it closes the block of code that defines the `main()` method.

Line 7:

```
7 }
```

This brace closes the block of code that defines the `HelloWorld` class. Note that all of the pairs of brackets have been aligned intentionally.

1.4.2 Compiling and Running the HelloWorld Program

Once the code has been written using Notepad, we have to save our file in a folder. You can name this folder for example **java**, and save it in the root folder **C:**, but you can call this folder whatever you like and save it wherever you want. It is advisable to have a name that does not contain spaces in order to avoid unnecessary complications. So, a folder like **C:\java** would be preferable.

Please pay attention when saving the file, since Notepad is a generic text editor, not an editor for Java, so it tends to save files with the .txt **extension. So, if we try to save the file with the name** HelloWorld.java**, Notepad will save the** HelloWorld.java.txt **file, and this is not what we want. To avoid this behavior, you must include the file name (which we must remember is** HelloWorld.java**) between quotes in this way "**HelloWorld.java**". Alternatively, on the save screen, you can select the value** All files (*) **in the** Save as **field, located just below the text field where you can write the file name.**

At this point, we can open a command prompt and look inside our folder. After making sure that the **HelloWorld.java** file is placed in the folder (use the `dir` command), we can move on to the compilation phase.
If we run the command:

```
javac HelloWorld.java
```

we're passing our source file (**HelloWorld.java**) as input to the compiler (**javac**) provided by the JDK. If, at the end of the compilation, there is no error message displayed, it means that the compilation was successful.
Then we can see that a file named **HelloWorld.class** has been created in our folder. This is precisely the source file translated into bytecode, ready to be interpreted by the JVM.
So if we run the command:

```
java HelloWorld
```

our program, if no exceptions are raised by the JVM, will be sent for execution, printing the long-awaited message:

```
Hello World!
```

1.4.3 Our Interaction with the Computer

Here are the steps we have taken to interact with the computer:

1. We asked the operating system to open the Notepad application by clicking on the corresponding icon with the mouse.
2. We wrote our source code within the Notepad graphical interface using the keyboard.
3. We saved the file on our hard disk (auxiliary memory), in a specific folder (**C:\java**).
4. We opened a session of the "command prompt" program running the **CMD** command from the **Start** menu, then we moved to the folder containing the file just saved.
5. Then we compiled our file by running the **javac** application contained in the JDK bin folder, by running the command "`javac HelloWorld.java`" directly on the command prompt interface.
6. So, the compiler program has been loaded into the main memory (RAM) along with its input (the **HelloWorld.java** source file).
7. The compiler has executed its instructions thanks to the processor (CPU) and produced, as output, the translation of the source code in bytecode, saving it in a **HelloWorld.class** file.
8. We then ran the interpreter provided by the Java Virtual Machine with the **java** command contained in the JDK **bin** folder, by writing the "`java HelloWorld`" command directly on the command prompt interface.
9. The JVM was loaded into the main memory (RAM) along with its input (the **HelloWorld.class** source file).
10. The Java interpreter translated the Java instructions into machine language that is understandable to the processor, and then the string "`Hello World!`" was printed.

1.5 Problems That Can Occur During the Compilation and Execution Phases

Usually, in the early days, newbie Java programmers often receive seemingly mysterious messages from the compiler and the Java interpreter. Do not be discouraged! You must be patient and learn to read the messages that are returned. Initially, it may seem difficult but

after a short time you'll realize that the mistakes you make are often the same. Knowing how to read the compiler errors will make you a better programmer.

1.5.1 Problems That Can Occur During the Compilation Phase

```
"javac" javac is not recognized as an internal or external command.
```

In this case, it is not the compiler that is reporting a problem to us, but the operating system that does not recognize the **javac** command that should run the JDK compiler. Therefore, the latter has not been installed correctly. In this case, it is likely that the `PATH` environment variable has not been correctly set (see Appendix B).

```
HelloWorld.java:1: error: class Helloworld is public, should be
declared in a file named Helloworld.java
public class Helloworld {
       ^
1 error
```

In this case, we called the file `HelloWorld` while the class has been called `Helloworld` (note the lowercase "w"). The compiler is not clever enough to understand that we did not do it voluntarily.

```
HelloWorld.java:3: error: cannot find symbol
        System.out.printl();
                   ^
  symbol:   method printl()
  location: variable out of type PrintStream
1 error
```

If we receive this message, we know that we wrote `printl` instead of `println`. The compiler cannot realize by itself that it was simply a typo, and then told us that the `printl()` method was not found. It is essential to understand the error messages of the compiler, keeping in mind however, that there are limits to the clarity of these messages. The compiler does not always report the problem to us correctly:

```
HelloWorld.java:1: error: class, interface, or enum expected
public Class HelloWorld {
       ^
HelloWorld.java:2: error: class,
interface, or enum expected
    public static void main(String args[]) {
                   ^
HelloWorld.java:4: error: class,
interface, or enum expected
    }
    ^
3 errors
```

In this case, it is not true that there are three errors. In fact, there is only one error: we wrote the word "`class`" with a capital letter and then the JVM has explicitly requested a class declaration (or interface or enum, concepts that we will clarify later). Remember, "`Class`" is not the same as "`class`" in Java.

```
HelloWorld.java:3: error: ';' expected
        System.out.printl()
                           ^
1 error
```

In this case, the compiler has immediately understood that we have forgotten the semicolon that is used to conclude every instruction. Unfortunately, our compiler will not always be that precise. In some cases, if we forget a semicolon, or worse, if we forget to close a block of code with a brace, the compiler may report the existence of non-existent errors in successive lines of code.

1.5.2 Problems That Can Occur During the Execution Phase

In this phase, so-called "exceptions" are usually raised by the JVM. The ninth chapter is dedicated to exceptions and related concepts.

```
Error: Main method not found in class HelloWorld, please define the main method as:
    public static void main(String[] args)
```

If we receive this message once the `HelloWorld` program has been launched, we have probably incorrectly defined the `main()` method. We probably forgot to declare it `static` or `public`, or we mistyped the list of arguments (which must be `String args[]`) or maybe we did not call the method "`main`".

```
Error: could not find or load main class
 HelloWorld.class
Caused by: java.lang.ClassNotFoundException: HelloWorld.class
```

In this case we tried to run the command:

"`Java HelloWorld.class`" instead of "`java HelloWorld`".

1.6 Other Development Environments

Up to now, we have seen how to interact with the command line to invoke the compiler with the **javac** command, and the interpreter with the `java` command. To write our code, we used a simple text editor such as Notepad. We consider it important to understand how low-level

Java works, and we can continue to study the entire book using these tools, but it is objectively awkward to work with two different windows.
When we start programming seriously, we will probably use an IDE like Netbeans or Eclipse. These fantastic tools, however, are not recommended for those who are just starting to code. Their complexity can distract from the real goal, namely that of studying Java. If we also need to learn how to use the IDE, it may distract us from focusing on the language. Also, using an IDE for the small programs that we will initially write, does not seem appropriate. Finally, the comforts offered by an IDE such as auto-completion of the code or code suggestions, can cause dependence and laziness! It wouldn't be difficult to meet people who completely lack some basic concepts, even though they have been programming with these instruments for years.
So, if you are a newbie, we suggest not using an IDE immediately. In the fifth chapter, we will introduce other development environments including IDEs, explaining how to start using them. From the fifth chapter onwards, it should be easier to start working with IDEs, because you should already have some confidence with the language.

Further, in the same chapter, we'll introduce other tools to accelerate the way we write code, like JShell (introduced with Java 9) and the option to launch the single-file programs using only the source-code file, skipping the compilation step (introduced with Java 11).

Of course, you may disagree and choose to ignore our suggestion to postpone the use of an IDE. In that case, at least try not to forget what your main goal is.
I've created an open source Java editor designed for those who start programming: **EJE** (**Everyone's Java Editor**). It is not an IDE, but a simple editor that was created not for writing text files but to code in Java. It offers some smart features compared to a generic text editor. Among its features are the coloring of Java syntax, and text completion for some expressions. Above all, it is possible to compile and execute files, by pressing simple buttons, without having to move to a command prompt and edit boring commands. EJE is a really simple program, and there is no need to study it. You can download if for free at http://www.claudiodesio.com and http://sourceforge.net/projects/eje.
Conclusion: if you are already familiar with integrated development environments (IDEs) then you may also choose to use Netbeans or Eclipse. If you are starting to program just now, or in any case you want to try to focus only on the study of Java, our suggestion is to use EJE for the first few times, then move on to an IDE after the fifth chapter.

Summary

We call **hardware** the physical part of the computer. Some hardware components collect **inputs** from the outside (mouse, keyboard, etc.), others provide **outputs** (monitor, printer, etc.), and other components are not used directly by the computer user (motherboard, processor, memory, etc.). The **processor** (CPU) performs mathematical operations and manages the data movements in the memory. The **memory** is divided into **main memory** (RAM memory) where the programs are loaded (with their data) during their execution, and **auxiliary memory** (usually hard drives) that, unlike the main memory, survives after the computer has been shut down.

Software is the term we use to refer to programs that run on a computer. The **operating system** is a program that manages a computer in its entirety and also allows for the installation and execution of other programs. A **programming language** always has a vocabulary and a set of rules. A processor cannot interpret a programming language because it is capable of using two symbols only (`0` and `1`) and a particular **machine language** that uses instructions written with these two symbols. To program we must then write the source code with the programming language in text files, and then translate it into machine language using a software such as a compiler or an interpreter. A **compiler** translates the instructions written in the programming language, into machine language instructions creating executable **binary files**. An **interpreter**, on the other hand, translates the instructions contained in the source code into instructions of machine code on the fly during the runtime. The term "Java" is used both to refer to the programming language, and to the technology generated by it (which branches off into many sub-technologies). Its main features are robustness, standardization, syntax, power and, above all, **platform independence**.

Java is a language both compiled and interpreted. Compiling, however, does not create files containing instructions in machine language, but files containing instructions written in a language similar to the machine language called **bytecode**. The Java interpreter defined within the Java Virtual Machine (JVM) is able to interpret the bytecode with excellent proficiency.

The **Java Development Kit** (**JDK**) is the official development environment for Java. With a simple text editor such as Notepad, and the basic knowledge of the command prompt, it allows us to program in Java. The steps to be taken to write and run a program are always fixed: write the code, save it, compile it and run the program.

A Java program is written using the **class** concept, which contains all the operating code of the program. Each class must be written in its own **source file** (with suffix **.java**) and must have exactly the same name as the file. A file to be executed must contain a method called `main`, which is the starting point of a Java program. The compilation process will create a corresponding file with the extension **.class** that will contain the bytecode. This file can be passed as input to

any Java Virtual Machine of any hardware-software platform to be executed. It is particularly important to know how to interpret compiler **error messages**, and those launched by the JVM during program execution. It is not recommended for beginners to start programming directly with an **IDE** so that the focus must be only on the study of the language. It is advisable to postpone the use of an IDE for as long as possible (at least until studying the fifth chapter) in order not to be overwhelmed by the complexity of the IDE itself. For beginners, I've expressly created EJE, a user-friendly editor with basic functionalities that simplify the Java coding.

Exercises, source code, appendices and other resources are available at http://www.javaforaliens.com.
Bear in mind that the exercises are an integral part of the text and have only been outsourced in order to minimize the number of pages and consequently the cost of this book. In the exercises, you will modify and create new code, confirm your knowledge, and above all learn new things. It can seem strange but understanding the theory perfectly does not automatically imply that you will be able to program correctly. It is surprising how many difficulties you can find while coding, even when your ideas seem very clear. Without practice, there is no Java programming.

Chapter Goals

Have the following goals been achieved?

Goal	Achieved	Achievement Date
Know some basic computer concepts (Unit 1.1)	○	
Know how to define the Java programming language and its features (Unit 1.2)	○	
Be able to interact with the development environment: the Java Development Kit (Units 1.3, 1.4, 1.5)	○	
Know how to type, compile and run a simple application (Units 1.3, 1.4, 1.5)	○	
Choose a development environment (Unit 1.6)	○	