by Nacho Iborra

Development Environments
Block 1
Unit 1: Programs, languages and compilers

1.1.1. Software & program

If we look for the definition of *software* on the Internet, we can find some of them, although they are basically the same. We call **software** the set of programs, documentation and data related with a computer system.

Each software product is (or can be) different from the rest, because it is developed for a different customer, or to fulfill a different purpose. So, developing this product implies some previous steps: understand what we have to do, make a previous design and implement it, as we will see later. Therefore, we can't compare software development with industrial production (such as keyboard manufacturing, for instance), where everything is much more automated. Software development requires the creation of a software project, and a group of people working in a coordinated way. Besides, software does not break along time, although its performance can be reduced because of its own updates and improvements.

1.1.1.1. Software components

From previous definition of software, we can figure out that it is composed of three elements:

- **Programs**: sets of instructions that provide the desired functionality. They are written in a specific programming language.
- **Data**: programs need data to work with. These data can be retrieved and/or stored from/in databases or files.
- **Documentation**: software documents explain how to use it, which elements are part of it and how they are interconnected, so that it can be updated or fixed in the future.

1.1.1.2. Software types

There are two main types of software:

- **Application software**: they bring some kind of service to the customer, such as text processors, spreadsheets, drawing, etc. In this type of software we can find some subcategories, such as management software (payroll, accounting), engineering or scientific software (CAD, simulators...), network software (web browsers, FTP clients or servers...), etc.

- **System software**: they manage a computer system. They are basically programs that support or help other programs. In this category we can talk about operating systems or compilers, for instance.

1.1.2. Languages and compilers

We have seen that programs are sets of instructions that are provided to a computer to complete a task. These instructions are written in a programming **language** of our choice, and this way we create some text files called **source code**, written in the chosen language.

1.1.2.1. Language types

When we want to choose a specific programming language, we distinguish between **high level** languages (close to human language, and so, easier to understand by the programmers), and **low level** languages (close to machine language, and so, harder to understand by the programmers, but more efficient).

- We have a wide variety of **high level** languages to choose, depending on the type of application that we want to implement. We can talk about C, C++, C#, Java, Javascript, PHP, Python, etc.
- Among **low level** languages, maybe the most popular one is the assembly language, a very concrete set of instructions that are easily translated into machine language.

Here we can see a simple program written in Java that just prints "Hello" on the screen:

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

The same program written in C could look more or less like this:

```
#include <stdio.h>

int main()
{
    printf("Hello");
    return 0;
}
```

1.1.2.2. Compilers and other language processors

Computers can't understand any of the programming languages that humans use to create their programs. In order to make them work, their instructions need to be translated into a language that computers can understand. This language is called *machine code*, and is composed of bits (zeros and ones).

If we want to translate a given programming language into machine code, we use a tool called **compiler**, although this assertion is not completely true. There are some different language processors that can be used, depending on the language itself:

- **Compilers**: they translate the code written in a specific language into machine code, and they generate an executable file with the result. For instance, if we compile a program written in C under Windows, we will get an *EXE* file that we will be able to run.
- **Interpreters**: they translate from the specified programming language to machine code "on the fly". In other words, they don't generate any executable file. So, every time we need to run the program, we need to have the source file available. This kind of language processor is very usual in languages like PHP or Python. This way, the response time increases a little, but the program can be run in multiple platforms.
- **Virtual machines**: an intermediate solution between compilers and interpreters is the one used by languages like Java. These programs are not compiled into a native machine code (there is no *EXE* file in Java, for instance), and they are not interpreted either. Java compiles the source code and translates it into its own intermediate machine code. Then, it runs its virtual machine (JVM, *Java Virtual Machine*), that is in charge of interpreting and running this code every time we need. This way, we don't need to have the source code available before we run the program, and we don't depend on a given platform either (Windows, Linux, etc.). We only need to have a JVM installed in our system in order to run our Java programs. The same thing happens with C# and its virtual platorm *.NET*.

Let's install some compilers for some programming language, and test them with some sample programs. To be more precise, we are going to install Java and C/C++ compilers.

Installing some compilers: Java

Java implementations have changed along time. At the beginning, it was Sun Microsystems the company in charge of implementing the earliest versions of the language... until version 6, where Oracle bought Sun Microsystems. Then, there was an official Oracle implementation branch, and an open implementation branch called *OpenJDK* that is an official reference implementation of Java Standard Edition since version 7. From that point, many other additional implementation branches have arised, such as *AdoptOpenJDK*, *Excelsior JET* and so on. Some of them are free, as *OpenJDK*, and some other are commercial, or have a commercial version for some specific purposes, as Oracle has done with the latest versions.

In order to narrow the range of available options, let's focus just on Oracle and OpenJDK implementations. The first one has some license restrictions for some given uses, whereas the second one is absolutely free to use. However, Oracle implementations offer LTS (*Long Term Support*) versions from time to time (version 8 was LTS, and so is version 11), whereas OpenJDK versions are no longer supported as soon as a new version comes into scene.

In this module we are going to use **Java version 11**, and we are going to rely on **OpenJDK implementation**, since its installation is cleaner, and it lets us easily deal with multiple installed versions. However, we are going to start by showing how to install official Oracle version for every platform.

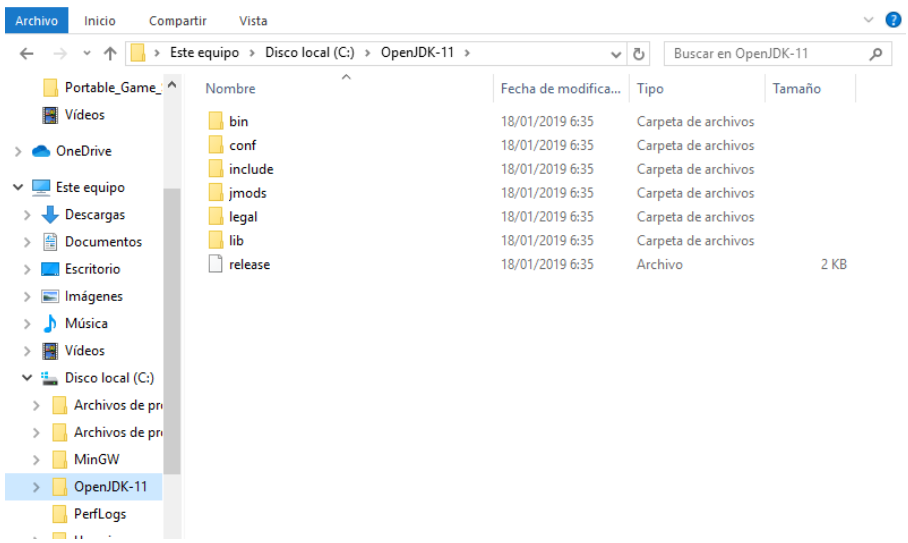
Option 1: Installing Oracle JDK

If you want to use Oracle's official JDK, then you need to follow these steps:

- Go to the Oracle JDK download website (you may be asked to sign up before downloading the software). Download the package according to your operating system and install it.
- In case you are using **Linux**, you may be forced to download a *tar.gz* file from Oracle web site, and unzip it in your chosen folder.
- After following these steps, JDK commands may not work yet, specially under Windows and Linux, until we configure some environment variables, as we will see later.

Option 2: Installing OpenJDK (*recommended*)

If we choose OpenJDK implementation (as we are going to do in this module), then we need to download the appropriate *zip* or *tar.gz* package. We can do it from this website. Then, unzip the file in your chosen folder. For instance, it can be `C:\OpenJDK-11` for Windows, or `/home/alumno/openJDK-11` for your Linux virtual machine. Anyway, there must be a `bin` subfolder (among others) into your main installation folder after unzipping (for instance, `C:\OpenJDK-11\bin` in Windows).



> **NOTE**: regarding Mac OS X, there will be a `Contents/Home` folder inside JDK main installation folder. Inside this `Home` folder you will find the `bin` subfolder where all JDK commands are placed.

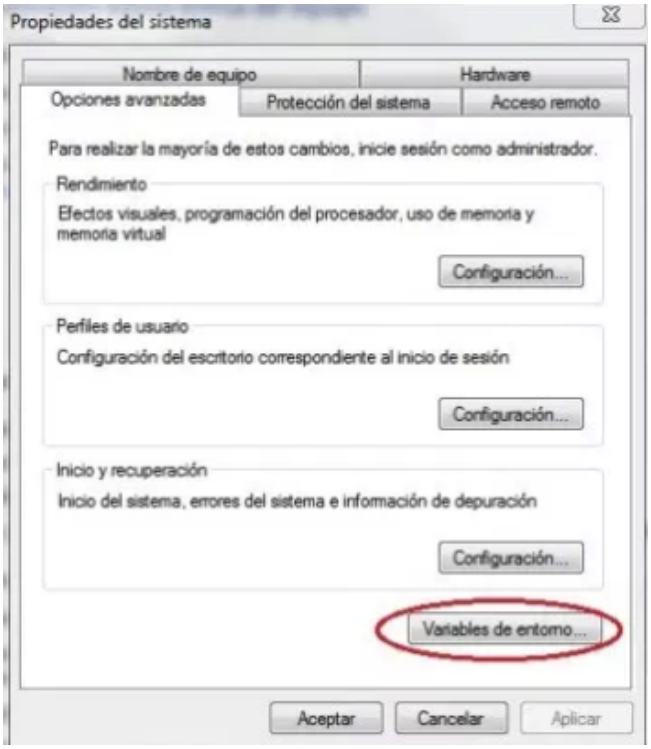
> **NOTE**: if you want to use the latest available OpenJDK version instead of the one that we are going to use along this module, you can get it from this link.

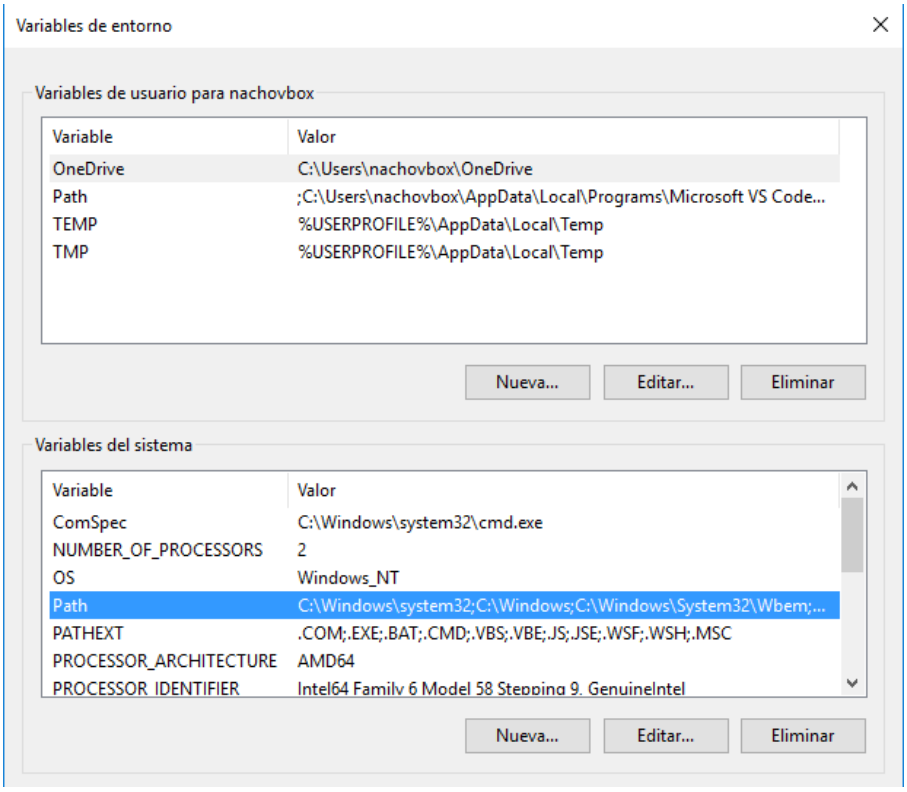
Setting up the *path* environment variable

In order for JDK commands to work, we need to add to the system *path* the folder where these commands are placed (typically, a `bin` subfolder inside the main installation folder). This way, the operating system will recognize the compilation commands whenever we type them from a terminal. Let's see how to do these steps in the most common operating systems:

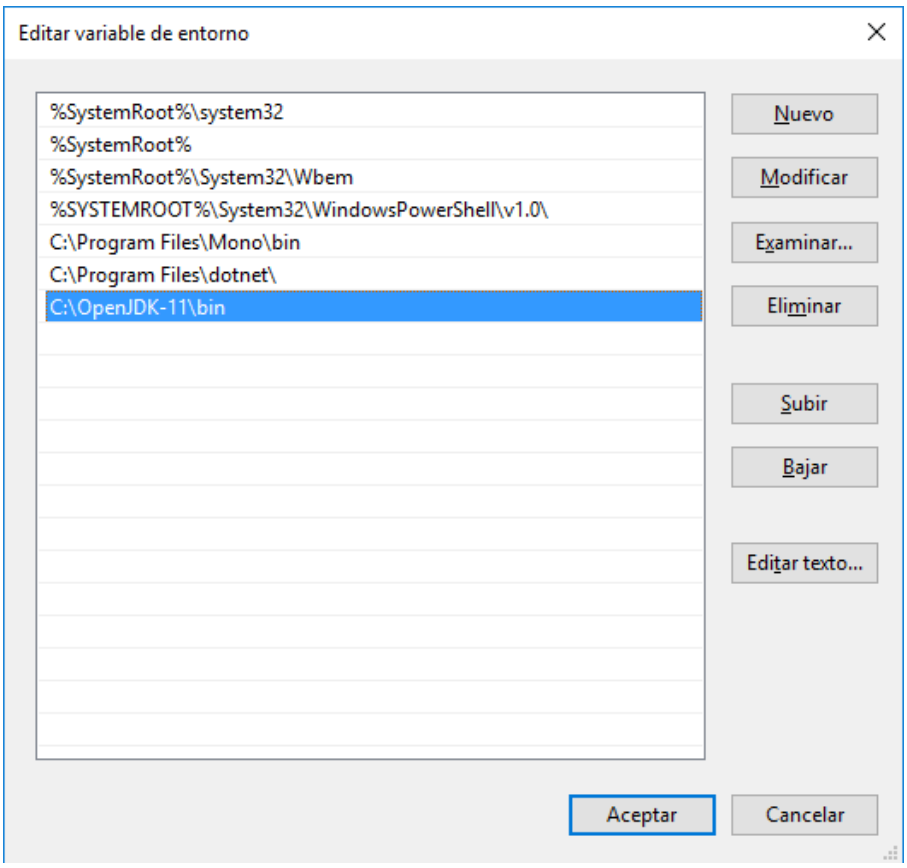
Under **Windows**:

1. We open Windows search (Cortana, in case you are using Windows 10), and look for *Editar variables de entorno*. Then, we click on *Variables de entorno* button.





2. Then we choose `Path` variable from the lower list and click on Edit button. Next, we add the path to the Java compiler. If we unzipped OpenJDK under `C:\OpenJDK-11`, for instance, then we add `C:\OpenJDK-11\bin` path to the variable.



Under **Mac OSX**:

Open a terminal an type these commands. They update `PATH` environment variable by adding a new path to the JDK bin folder. We assume that you unzipped JDK under `/openjdk-11`, but if not, replace this path with the appropriate one in the first instruction:

```
echo "export PATH=$PATH:/openjdk-11/Contents/Home/bin" >> ~/.bash_profile
source ~/.bash_profile
```

Under **Linux**:

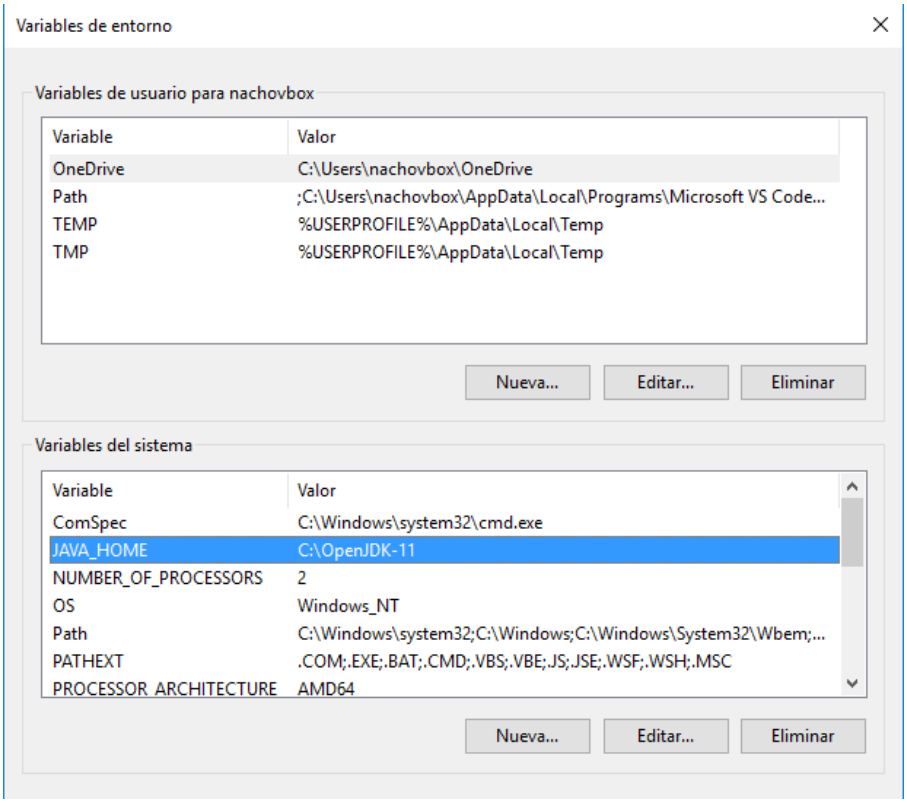
Open a terminal an type these commands. They update `PATH` environment variable by adding a new path to the JDK bin folder. We assume that you unzipped JDK under `/home/alumno/openjdk-11`, but if not, replace this path with the appropriate one in the first instruction:

```
echo "export PATH=$PATH:/home/alumno/openjdk-11/bin" >> ~/.bashrc
source ~/.bashrc
```

Setting up the *JAVA_HOME* environment variable

Besides, we may need to add a new environment variable called `JAVA_HOME` (or edit the existing one, if any), to point to the main installation folder of JDK.

- Under **Windows**, we just add a new environment variable from *Editar variables de entorno*, just like we did before with *Path* variable, and set its value to the JDK main installation folder (`C:\OpenJDK-11`, for instance).



- Under **Mac OSX**, type these commands from a terminal (again, replace the JDK main folder with your actual folder):

```
echo "export JAVA_HOME=/openjdk-11/Contents/Home" >> ~/.bash_profile
source ~/.bash_profile
```

- Under **Linux**, type these commands from a terminal (again, replace the JDK main folder with your actual folder):

```
echo "export JAVA_HOME=/home/alumno/openjdk-11" >> ~/.bashrc
source ~/.bashrc
```

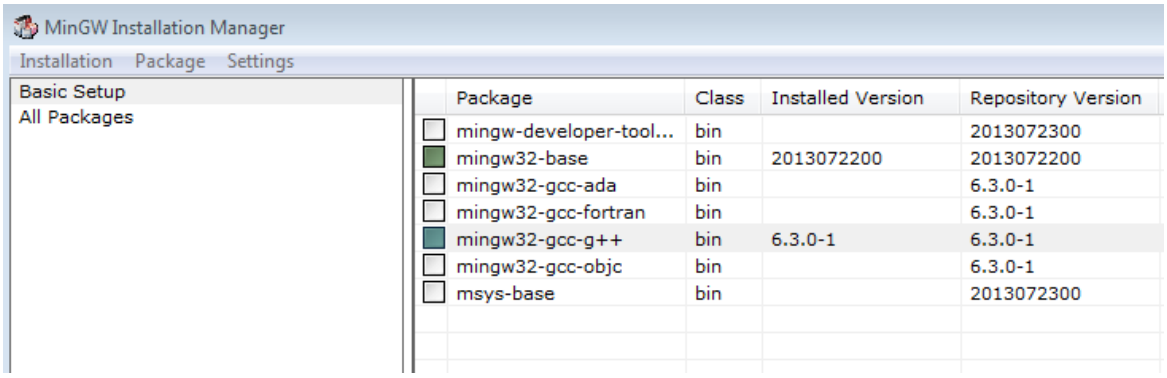
Checking JDK commands

In order to test that everything is correctly installed, open a terminal and type `javac -version`. You should see something like this in the terminal (the version may vary depending on your actual installation):

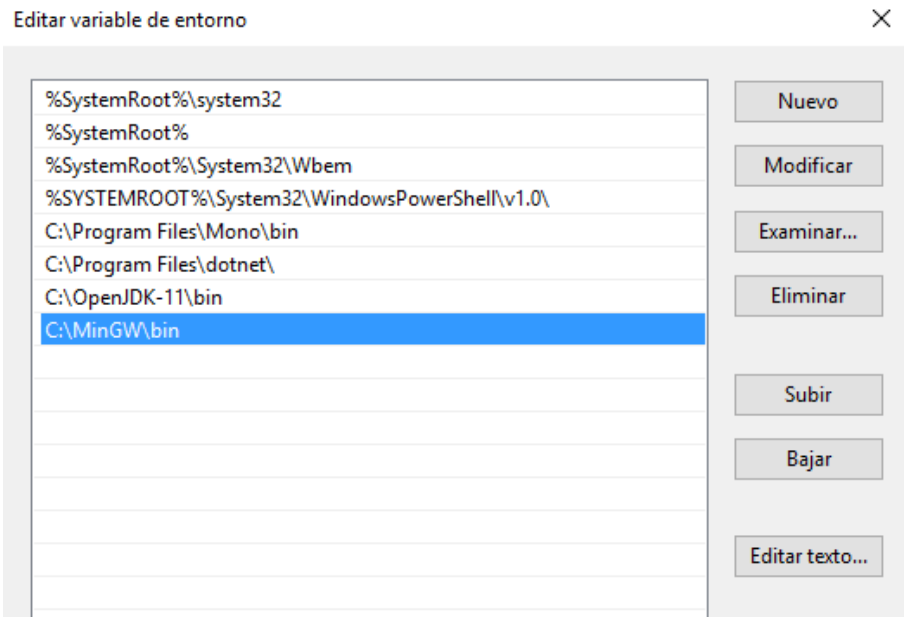
```
javac 11.0.2
```

Installing some compilers: C/C++

Let's try now with a C/C++ compiler called *gcc*. In case of Linux distributions, this compiler is installed by default, and in Mac OSX systems you can also have it by installing XCode. Regarding Windows, you can get it by installing *MinGW* package, that can be downloaded from its official website. In the installer, you must choose the C/C++ compiler (*mingw32-gcc-g++*) and the base (*mingw32-base*).



Once MinGW has been installed, we need to add the `bin` installation subfolder to the `PATH` environment variable, as we did before with JDK. In this case, we need to add `C:\MinGW\bin` folder to this variable, right as we did before for Java compiler.



After doing this, if we type `gcc` command in the terminal, we should see something like this:

```
gcc: fatal error: no input files
```

It is an error message that indicates that we haven't specified any input file to the compiler, but the message itself shows that the compiler has been succesfully located.

Proposed exercises:

1.1.2.1. There are some studies and analysis that try to classify the programming languages according to its popularity or current usage. Maybe one of the most accurate ones is carried out by RedMonk web site. It is based on crossing data between the main source code repository (GitHub) and the main programming help page (StackOverflow). Here you can check one of their latest analysis. Take a look at it, check the results and answer these questions: How many of the top 20 languages did you know? Do you miss any other language in this top 20 list?

1.1.2.2. Create a new text file called "Test.java" in your working folder, and write the same Java example seen before that writes "Hello" on the screen. Compile it from the terminal, by typing this instruction from the same folder in which the source file is placed:

```
javac Test.java
```

Then, a new file called "Test.class" will have been created. This is the compiled file for the Java Virtual Machine. To run this program, type this command from the same folder than the *.class* file (you should see "Hello" in the screen):

```
java Test
```

1.1.2.3. Create a source file called "test.c" in your working folder, and type the code seen before to write "Hello" in C language. Compile it using this instruction from the terminal:

```
gcc -o test test.c
```

`-o` parameter sets the name of the executable (*test* in this case). After this, a new executable called *test* (or *test.exe* in Windows) will have been created in our working folder. We can run just by typing `test` from our working folder. We should also see the "Hello" message in the terminal.

1.1.3. Some popular languages

To finish with this unit, let's see some popular programming languages and their main features, so that we can guess why they are so popular, and their usefulness nowadays.

1.1.3.1. C language

If we have to define in a few words what C language is, maybe the most suitable definition would be *the language*, because most of current languages are based on it. In fact, C language is an evolution of an older language, called B, but it was not so popular.

Before C language arrived, programmers used some kind of physical devices to implement their programs, such as perforated cards, or in the best case, some assembly, low level language, hard to understand, write and maintain.

Therefore, we can figure out that the arrival of C language in the beginning of the 70s was a complete success. Despite being a language with high level structures, it also has some low level features, such as direct memory control through pointers.

Nowadays, C language is not very popular when building applications, but it is still used for developing operating systems, libraries and some tools such as compilers for other languages. It is very appreciated because of the efficiency of its code.

1.1.3.2. C++

Next evolution step after C language was C++, at the end of the 70s. It extended C language to let us work with classes and objects. So C++ is a hybrid language (it lets us use either traditional, structured programming and object oriented programming).

Currently, its main area of application is the video game world. We can develop either game engines, such as Unreal Engine, or video games with some libraries, such as Unreal, Cocos2D, SDL... As C++ is an extension of C language, it still lets us control some low level features (such as direct communication with graphical devices), and it has lots of available libraries to make the programmer's task much more comfortable.

1.1.3.3. Java

Java is an object oriented language, created at the beginning of the 90s by *Sun Microsystems*. As we have seen before, it has its own virtual machine to run the programs, so Java applications are platform independent, and we can run them either on Linux, Windows, Mac and other systems. In fact, Java was conceived to program several types of electronic devices, including home appliances. But it got so popular that it quickly focused on computer applications.

With Java and some other languages of those years, we started saying goodbye to this "open door" to the low level world. Memory and system access is much more restricted, and the language provides other high level features, such as the *garbage collector*, which is in charge of cleaning the memory periodically, removing every element that is no longer used. This task was completely manual in previous languages (C o C++).

Java has a wide area of application nowadays: we can develop desktop applications with libraries like JavaFX, mobile applications (Android uses Java), web applications (with servlets and JSP pages, or even with more advanced frameworks, such as Spring), etc.

1.1.3.4. C#

C# was created at the beginning of the 21st century, approximately, as a new extension of its eldest brothers C and C++. However, it has some influences from other popular languages, such as Java. If you take a look at the code written in these two languages, you will find many similarities, so it is easy to learn one of them once we have learnt the other one.

It also took from Java language the usage of a virtual machine to run its applications: *.NET* platform, which is in charge of running C# programs in different platforms.

Current area of application for C# language is also wide: from desktop applications using Windows Forms or WPF to web applications (ASP .NET) or video games (Unity).

1.1.3.5. Javascript

We leave for now multi-purpose languages (like Java or C#), with which we can develop many different types of applications, to focus on another language that, initially, was conceived to develop web applications: Javascript. This interpreted language was born in the middle of the 90s, and took its name from Java language, because of its popularity, although they have nothing in common.

At the beginning, the main purpose of Javascript was to develop web applications in the client side, this is, in the browser. This way. we could add dynamic content to our pages, add or remove elements without calling the server. With AJAX technology, we could also call the server, get the results and refresh only a portion of the web page.

In the last years, Javascript has taken giant steps in its market scope, and it is not only used for client side application development, but also for the server side through Node.js framework. This way, we can use Javascript in the same way that we use PHP, JSP or ASP.NET. Besides, we can also develop desktop applications with frameworks like Electron, and mobile applications with hybrid technologies such as Ionic.

1.1.3.6. PHP

PHP is another language designed for web development, and it is currently used to develop most of the web applications in the server side. It was created in the middle of the 90s as well, but its evolution has not been as notorious as Javascript. It is still only focused on server side web applications, but its ease of use and possibilities make it the preferred language for this task.

1.1.3.7. Other languages

The languages that we have seen so far are, maybe, the most popular ones, and the ones with the highest market share. But we must also pay attention to other languages that are also popular, due to different reasons. For instance:

- **Python**, an interpreted language that is really easy to learn (it is usually chosen in many educational institutions to teach programming). Resulting code is usually much more compact and understandable that the same code in other traditional languages, for the same task. We can also develop several types of applications with this language, such as system scripts, video games (with libraries like *PyGame*) or server side web applications (with frameworks like Django).
- **Go** is a compiled language based on C, created by Google in 2009. It lets us use either traditional structured programming or object oriented programming, and it also includes some interesting features from other languages (Python, for instance). Taking into account who is behind this language, we can bet it will be very important in the future.
- **Swift** is a language developed by Apple in 2014. It has replaced Objective-C language in the development of Mac OSX and iOS applications, so it is expected to be very useful and popular in the next years.

> **Proposed exercises**:

> **1.1.3.1.** Look for an online code editor that lets us work with many different languages. Here you can get one: Ideone. Find out how to write a program to say "Hello" in all the languages seen in this unit, and/or in any other language of your choice. Write your answers in a text file and save it.