Functional programming is a very famous programming paradigm (emphasized in academia rather than in commercial software development) that has its structure based on expressions and/or declarations instead of statements, which are very common in another programming paradigm so called imperative programming. Functional programming evaluates computation as mathematical functions and discourages mutable data and changing the program state in order to reduce or eliminate side effects. Functional programming has its roots based on Lambda Calculus, which is a formal mathematical model for high order functions as atoms, meaning that the every single abstraction can be translated into a function that takes a function as its parameter and returns another function as its output On the hand, procedural programming bases itself in procedure, functions or sub-routines calls (not to be confused with mathematical functions related to functional programming). It also follows the imperative paradigm as it makes explicit reference to changing-state, performs a series of well-structured sequential steps and may have side effects. Systematically, it embraces a succession of statements and functions to perform a computational task or program. Pascal, C and Go are example of procedural languages. **Software Packages:** Packages are collections of files (e.g binary files, scripts and so on), that are bundled together and can be installed and removed as a group, Microsoft Office for example. Sometimes are also called libraries when they are related to code reuse. They may or not be dependent on other packages in order to function. **Package Managers:** A package manager, as its name suggests, is a software used to install, uninstall and manage software packages. They make the task software code/binary distribution easier, designed to eliminate the need for manual installs and updates. Homebrew is an example of package manager. **API:** Application Programming Interface is a set of functions and protocols for software applications by providing specifications that software programs can follow to communicate with each other. Examples on that would be: REST, OpenGL and others. **Wrappers:** Wrapper is a software that functions as an adapter for other softwares. An example would be software for a bank that would encapsulate a cryptography package/library in its source code. Programming languages are designed to be easily understood by machines and people, having the purpose of interfacing this interaction. They are very simple if compared with natural languages, such as English and Spanish. Also, programming languages are usually written, unambiguous and have a well-specified grammar, whereas natural languages may be spoken and be related to culture and region, being very ambiguous. There are various types of languages such as programming, natural, sign and others. Most programming languages try to solve specific problems rather than general ones. That is to say, comparing programming languages is a rather hard task. One would say that C is better than Java because it is faster and more optimized and other would say that Java is better than C because it is easier to handle memory or to develop commercial software. Many programming languages exist because, as any other kind of language, they evolve and are improved with time. With better processors, we are now able to have better abstractions in our programming environment (remembering the the more abstractions the farther from the bare metal we are and therefore, longer we take to solve a given problem). Not all people program in Java because people have different needs and they are trying to, many times, solve different problems. Type checking can be handy when you want to eliminate certain types of errors before running a program. Imagine that you have a function $f$, that takes an parameters of type **int** and returns an **int**. Type-checking will guarantee that $f$ is only operating on **ints**. So before you run the program, you will catch any case where you call function $f$ on the wrong type, which will cause a compiler error. The most prevalent programming languages on the industry are **imperative languages**, such as Javascript, Python, PHP, Java and C#. A domain-specific language (DSL) are languages with very specific goals in design and implementation. Unlike a general-purpose language such as UML, a domain-specific language is designed to solve problems in a particular domain or universe. Each DSL is much better than a general-purpose language for describing operations on its own domain, but much worse for describing ideas that are outside its own scope. It is very hard to compare languages and have precise benchmarks regarding the **fastest** and the **best** programming languages. That happens because such benchmarks cannot compare or cannot create general topics in which one could use as a mean of measurement. However, the fastest languages are always the closest to the ”metal”. Thus, Assembly would be the fastest (not considering how hard it is to complete a task). Likewise, the best language varies from person to person, considering different backgrounds. Side effects are a very likely to happen when a program interacts with the external structures and/or global variables, e.g on IO procedures, file handling, networking and others. Side effects are nothing but simply the modification of some kind of state. Side effects are often related to the programming paradigms. The imperative programming paradigm, for example, is known for its frequent utilization, and as said on question #1, functional languages avoid to use them, as it can lead to many problems (bugs and such). **Concurrent Programming:** can be outlined as many (at least two) processes running throughout the same period of time. Differently from parallel programming, the execution does not need to happen at the same instant, thus, it consists of processes that overlap their lifetime. The main objective of concurrent programming is to model processes that happen concurrently. Many clients fetching data in the same time span from a server (not necessarily in parallel) would be a good example of how that works. **Event-driven Programming:** when a computer program control flow is determined by event listeners, the occurrence of these events are observed (or monitored) and executed by an, so called, event handler, which is typically a callback (delegate function) or method that determines the course of the program through user actions such as clicks or key presses or through messages sent by other threads and/or programs.