## Foreword

This isn't a pure form of my writings, most of it is being pulled from the free sources available over web. However I have gone through the entire content line by line and edited it to make it compact and to the point. While the entire work will give you deep insights on the subject topic wise, a few which are highlighted in a different color fall into the zone of frequently asked question and answers in most of the interviews. I recommend the readers to strictly practice the concepts with the assignments listed within while taking the example programs as reference

Wishing you good luck!


[Uday]

# INTRODUCTION TO SOFTWARE

**What is software?**

- Software is a set of instructions used to operate a computer and execute specific tasks. For example, a web browser like chrome or Edge is a software application that allows users to access the internet. An operating system (OS) is a software program that serves as the interface between other applications and the hardware on a computer or mobile device.

**Types of Software**

**Application software -** It is software that helps an end user complete tasks such as doing research, taking notes, setting an alarm, designing graphics, or keeping an account log. Application software lies above the system software and is different from system software in that it's designed for the end use and is specific in its functionality. This type of software is sometimes referred to as non-essential software because it's installed and operated based on the user's needs. Most of the applications on a mobile phone are examples of application software.

- Examples – Microsoft Office, Web Browsers [ Chrome, Safari ]

**System software** - It helps the user, hardware, and application software interact and function with each other. System software acts as a mediator or middle layer between the user and the hardware. It's essential in managing the whole computer system, when a computer is first turned on, it's the system software that is initially loaded into memory. Unlike application software, system software isn't used by end users. Instead, it runs in the background of a device. The most well-known example of system software is the OS, which manages all other programs in a computer. Aside from the OS, other examples of system software include:

- Basic input/output system (BIOS): the built-in firmware that determines what a computer can do without accessing programs from a disk.
- Boot: loads the OS into the computer's main memory or RAM.
- Assembler: Takes basic instructions and converts them into a pattern of bits that the processor can use to perform basic operations.
- Device driver: Controls a particular type of device attached to the computer, such as a keyboard or mouse.

**Programming software –** It's a type of system software, programming software isn't used by the end user. It's used by programmers who are writing code. Programming software is a program that is used to write, develop, test, and debug other software, including application and system software. These programs serve as a sort of translator. It takes programming languages such as Python or C++ and translates it into something a computer will understand, known as machine language code. Besides simplifying code, it Programming languages define and compile a set of instructions for the CPU (Central Processing Unit) for performing any specific task. Every programming language has a set of keywords along with syntax- that it uses for creating instructions.

Till now, thousands of programming languages have come into form. All of them have their own specific purposes. All of these languages have a variation in terms of the level of abstraction that they all provide from the hardware. A few of these languages provide less or no abstraction at all, while the others provide a very high abstraction. On the basis of this level of abstraction, there are two types of programming languages:

- Low-level language
- High-level language

The primary difference between low and high-level languages is that any programmer can understand, compile, and interpret a high-level language feasibly as compared to the machine. The machines, on the other hand, are capable of understanding the low-level language more feasibly compared to human beings.

What are High-Level Languages?

- One can easily interpret and combine these languages as compared to the low-level languages.
- They are very easy to understand.
- Such languages are programmer-friendly.
- Debugging is not very difficult.
- They come with easy maintenance and are thus simple and manageable.
- One can easily run them on different platforms.
- They require a compiler/interpreter for translation into a machine code.
- A user can port them from one location to another.
- It consumes more memory than the low-level languages.
- They are very widely used and popular in today's times.
- Java, C, C++, Python, etc., are a few examples of high-level languages.

What are Low-Level Languages?

- They are also called machine-level languages.
- Machines can easily understand it.
- Debugging is very difficult.
- These are not very easy to understand.
- All the languages come with complex maintenance.
- These are not portable.
- These languages depend on machines. Thus, one can run it on various platforms.
- They always require assemblers for translating instructions.

**List out the differences between High level language and low level language**

| Parameter | High-Level Language | Low-Level Language |
|---|---|---|
| Basic | These are programmer-friendly languages that are manageable, easy to understand, debug, and widely used in today's times. | These are machine-friendly languages that are very difficult to understand by human beings but easy to interpret by machines. |
| Ease of Execution | These are very easy to execute. | These are very difficult to execute. |
| Process of Translation | High-level languages require the use of a compiler or an interpreter for their translation into the machine code. | Low-level language requires an assembler for directly translating the instructions of the machine language. |
| Efficiency of Memory | These languages have a very low memory efficiency. It means that they consume more memory than any low-level language. | These languages have a very high memory efficiency. It means that they consume less energy as compared to any high-level language. |

| | | |
|---|---|---|
| Portability | These are portable from any one device to another. | A user cannot port these from one device to another. |
| Comprehensibility | High-level languages are human-friendly. They are, thus, very easy to understand and learn by any programmer. | Low-level languages are machine-friendly. They are, thus, very difficult to understand and learn by any human. |
| Dependency on Machines | High-level languages do not depend on machines. | Low-level languages are machine-dependent and thus very difficult to understand by a normal user. |
| Debugging | It is very easy to debug these languages. | A programmer cannot easily debug these languages. |
| Maintenance | High-level languages have a simple and comprehensive maintenance technique. | It is quite complex to maintain any low-level language. |
| Usage | High-level languages are very common and widely used for programming in today's times. | Low-level languages are not very common nowadays for programming. |
| Speed of Execution | High-level languages take more time for execution as compared to low-level languages because these require a translation program. | The translation speed of low-level languages is very high. |
| Abstraction | High-level languages allow a higher abstraction. | Low-level languages allow very little abstraction or no abstraction at all. |
| Need of Hardware | One does not require a knowledge of hardware for writing programs. | Having knowledge of hardware is a prerequisite to writing programs. |
| Facilities Provided | High-level languages do not provide various facilities at the hardware level. | Low-level languages are very close to the hardware. They help in writing various programs at the hardware level. |
| Ease of Modification | The process of modifying programs is very difficult with high-level programs. It is because every single statement in it may execute a bunch of instructions. | The process of modifying programs is very easy in low-level programs. Here, it can directly map the statements to the processor instructions. |
| Examples | Some examples of high-level languages include Perl, BASIC, COBOL, Pascal, Ruby, etc. | Some examples of low-level languages include the Machine language and Assembly language. |

What Is an Assembly Language?
An assembly language is a type of low-level programming language that is intended to communicate directly with a computer's hardware. Unlike machine language, which consists of binary and hexadecimal characters, assembly languages are designed to be readable by humans.
Low-level programming languages such as assembly language are a necessary bridge between the underlying hardware of a computer and the higher-level programming languages—such as Python or JavaScript—in which modern software programs are written. Assembly language is designed to understand the instruction and provide it to machine language for further processing. It mainly depends on the architecture of the system, whether it is the operating system or computer architecture.

- An assembly language is a type of programming language that translates high-level languages into machine language.
- It is a necessary bridge between software programs and their underlying hardware platforms.
- Today, assemble languages are rarely written directly, although they are still used in some niche applications such as when performance requirements are particularly high.

```
global  _main
extern  _printf
section .text
_main:
push    message
call    _printf
add     esp, 4
ret
message:
db  'Hello, World!', 10, 0
```

1. Save the file with any name example XYZ.asm; the extension should be ".asm".
2. The above file needs to compile with the help of an assembler that is NASM (Netwide Assembler).
3. Run the command nasm –f win32 XYZ.asm
4. After this, Nasm creates one object file that contains machine code but not the executable code that is XYZ.obj.
5. To create the executable file for windows, Minimal GNU is used that provides the GCC compiler.
6. Run the command gcc –o XYZ.exe XYZ.obj
7. Execute the executable file now "XYZ."
8. It will show the output as "Hello, world".

Why should you learn Assembly Language?
The learning of assembly language is still important for programmers. It helps in taking complete control over the system and its resources. By learning assembly language, the programmer can write the code to access registers and retrieve the memory address of pointers and values. It mainly helps in speed optimization that increases efficiency and performance. Assembly language learning helps in understanding the processor and memory functions. If the programmer is writing any program that needs to be a compiler that means the programmer should have a complete understanding of the processor. Assembly language helps in understanding the work of processors and memory. It is cryptic and symbolic language.

**Machine Language** - Sometimes referred to as machine code or object code, machine language is a collection of binary digits or bits that the computer reads and interprets. Machine language is the only language a computer is capable of understanding

Machine language example

Below is an example of machine language (binary) for the text "Hello World."

```
01001000 01100101 01101100 01101100 01101111 00100000
01010111 01101111 01110010 01101100 01100100
```

Below is another example of machine language (non-binary), which prints the letter "A" 1000 times to the computer screen.

```
169 1 160 0 153 0 128 153 0 129 153 130 153 0 131 200
208 241 96
```

## What are Compilers?

A compiler is computer software that readily translates programming language into machine code or assembly language or low-level language. It translates every program to binary (1's and 0's) that a computer feasibly understands and does the task that corresponds to the code. One condition that a compiler has to follow is the syntax of the programming language that is used. Thus, if the syntax of the program does not match the analysis of the compiler, an error arises that has to be corrected manually in the program written. The main work of the compiler is to translate the program into machine code and let the programmer know if there are any errors, ranges, limits, etc., especially the syntactical errors in the program. It analyses the entire program and converts it into machine code.

1. It reads the source code and provides an executable code.
2. Translates programs written in a high-level language to a language that the CPU can understand.
3. The process is relatively complicated and takes time for analysis.
4. The executable code will be in machine-specific binary code.
5. Total run time is more and occupies a large part of the memory.

## What is an Interpreter?

An interpreter is a computer program that converts program statements into machine code. Program statements include source code, pre-compiled code, and scripts. An interpreter works more or less similar to a compiler. The only difference between their working is that the interpreter does not generate any intermediate code forms, reads the program line to line checking for errors, and runs the program simultaneously.

1. It converts program statements, line by line, into machine code.
2. Allows modification of program while executing.
3. Relatively lesser time is consumed for analysis as it runs line by line.
4. Execution of the program is relatively slow as analysis takes place every time the program is run.

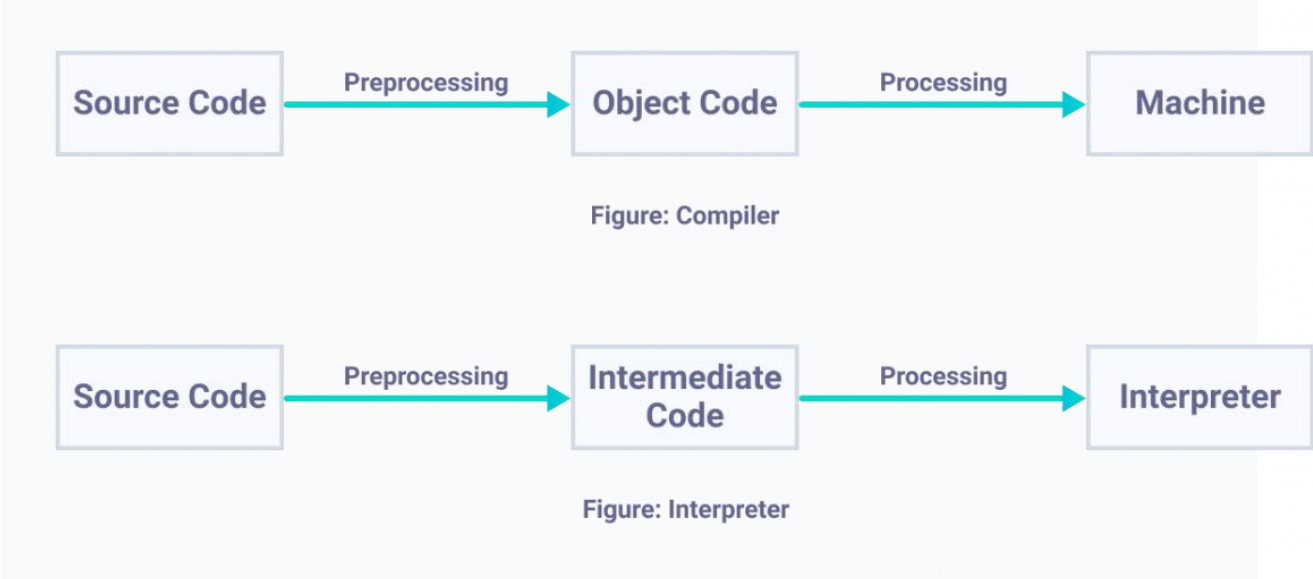
**Interpreter Vs Compiler: Differences between Interpreter and Compiler**

We generally write a computer program using a high-level language. A high-level language is one that is understandable by us, humans. This is called **source code**. However, a computer does not understand high-level language. It only understands the program written in **0**'s and **1**'s in binary, called the **machine code**. To convert source code into machine code, we use either a **compiler** or an **interpreter**.

Both compilers and interpreters are used to convert a program written in a high-level language into machine code understood by computers. However, there are differences between how an interpreter and a compiler works.

---

Working of Compiler and Interpreter




**Differences between compiler and Interpreter**

| Basis | Compiler | Interpreter |
|---|---|---|
| Analysis | The entire program is analyzed in a compiler. | Line by line of the program is analyzed in an interpreter. |
| Machine Code | Stores machine code in the disk storage. | Machine code is not stored anywhere. |
| Execution | The execution of the program happens only after the entire program is compiled. | The execution of the program takes place after every line is evaluated and hence the error is raised line by line if any. |
| Run Time | Compiled program runs faster | Interpreted program runs slower. |
| Generation | The compilation gives an output program that runs independently from the source file. | The interpretation does not give any output program and is thus evaluated on every execution. |
| Optimization | The compiler reads the entire program and searches multiple times for a time-saving execution. | No rigorous optimization takes place as code is evaluated line by line |
| Error and error execution | All the errors are shown at the end of the compilation and the program cannot be run until the error is resolved | Displays the errors from line to line. The program runs till the error is found and proceeds further on resolving. |
| Input | The compiler takes in the entire program for analysis. | The interpreter takes in lines of code for analysis. |
| Output | The compiler gives intermediate code forms or object code | The interpreter does not generate any intermediate code forms. |
| Programming languages | C, C++, C#, Java are compiler-based programming languages | PHP, PERL, Ruby are interpreter-based programming languages. |