Classification of Programming Languages

Programming languages are classified into two main categories:

1. **Low-Level Languages**
2. **High-Level Languages**

---

1. Low-Level Languages

Low-level languages are closer to machine hardware and are difficult for humans to understand.

Types of Low-Level Languages:

1. **Machine Language:**
    - It is written in binary (0s and 1s).
    - It is directly understood by the computer.
    - It is platform-dependent.
    - Debugging and error correction are difficult.
2. **Assembly Language:**
    - Uses mnemonics (short codes) instead of binary.
    - Requires an assembler to convert it into machine language.
    - It is also platform-dependent but easier than machine language.

Characteristics of Low-Level Languages:

- Platform-dependent.
- Fast execution speed.
- Difficult error detection and correction.
- Requires knowledge of computer architecture.

---

2. High-Level Languages

High-level languages are closer to human language and easier to learn.

Types of High-Level Languages:

1. **Procedure-Oriented Programming (POP):**
    - Definition: A programming approach focused on procedures (functions).
    - Features:
        - Uses functions to divide tasks.
        - Follows a **top-down approach**.
        - Example languages: C, BASIC, FORTRAN.
2. **Object-Oriented Programming (OOP):**
    - Definition: A programming approach focused on objects and data.
    - Features:

        - Uses **objects** that contain both data and methods.
        - Follows a **bottom-up approach**.
        - Example languages: Java, Python, C++.
3. **Structure-Oriented Programming:**
    - Uses structured control flow (loops, conditionals, functions).
    - Example languages: Pascal, COBOL.

Translators of Programming Languages

A translator converts high-level language into machine code. The three main types of translators are:

1. **Compiler:**
    - Translates the entire code at once before execution.
    - Detects all errors at once
    - Faster execution
    - Example: C, C++.
2. **Interpreter:**
    - Translates and executes the code line by line.
    - Detects errors statement by statement.
    - Slower execution
    - Example: Python, JavaScript.
3. **Assembler:**
    - Converts assembly language into machine language.
    - Example: MASM (Microsoft Assembler).

Difference Between Compiler and Interpreter

Feature	Compiler	Interpreter
Translation	Entire code at once	Line by line
Speed	Faster execution	Slower execution
Error Detection	shows all errors at once	shows errors one by one
Example Languages	C, C++	Python, JavaScript

Difference Between POP and OOP

Feature	POP	OOP
Focus	Functions	Data & Objects
Approach	Top-down	Bottom-up
Reusability	Less	More
Example Languages	C, FORTRAN	Java, C++

Principles of OOP

1. **Encapsulation:**
    - The concept of binding data and methods together in a class.
    - It restricts direct access to some of an object's components.
    - Example: class (as it contains many variables, objects and functions)
    - **Benefits:**
        - Improves code maintainability.
        - Enhances security.
2. **Inheritance:**
    - The mechanism by which a new class (child class/ sub class/ derived class) derives properties and behaviour from an existing class (parent class/ base class/ super class).
    - It promotes code reusability and hierarchical relationships.
    - Example: A `Car` class inheriting from a `Vehicle` class.
    - **Types of Inheritance:**
        - Single Inheritance
        - Multiple Inheritance
        - Multilevel Inheritance
        - Hierarchical Inheritance
        - Hybrid Inheritance
    - **Benefits:**
        - Reduces redundancy by reusing existing code.
        - Helps in building a logical structure for programs.
3. **Polymorphism:**
    - The ability of a function, method, or object to take multiple forms.
    - It allows the same interface to be used for different underlying forms (data types).
    - **Types of Polymorphism:**
        - Compile-time Polymorphism (Method Overloading)
        - Runtime Polymorphism (Method Overriding)
    - **Benefits:**
        - Improves flexibility and scalability of code.
        - Enhances maintainability by allowing modifications without affecting existing code.
4. **Data Abstraction:**
    - The process of hiding implementation details and showing only necessary features.
    - Achieved through abstract classes and interfaces.
    - Example: A `BankAccount` class exposing methods like `deposit()` and `withdraw()`, but hiding internal implementation details.
    - **Benefits:**
        - Reduces code complexity.
        - Enhances code reusability and security.

Difference between High Level Language and Low Level Language

Feature	High-Level Language	Low-Level Language
Abstraction	Closer to human language, easier to understand.	Closer to machine code, harder to read.
Execution Speed	Slower due to abstraction and compilation.	Faster as it directly interacts with hardware.
Portability	Highly portable, runs on multiple platforms.	Less portable, depends on hardware architecture.
Examples	C, Java, Python, JavaScript.	Assembly Language, Machine Code.
Ease of Coding	Easier, with simpler syntax and built-in functions.	Complex, requires detailed hardware knowledge.
Error Fixing/ Debugging	Easier	Harder
Use Case	Software development, web development, AI, etc.	Embedded systems, OS development, hardware programming.

Advantages and Limitations of High-Level and Low-Level Languages

High-Level Languages

Advantages:

1. **Easy to Learn & Use** – Uses human-readable syntax, making it easier for programmers.
2. **Portability** – Can run on different hardware with little or no modification.
3. **Faster Development** – Comes with built-in libraries and functions, reducing coding effort.
4. **Better Debugging & Maintenance** – Easier to debug and maintain due to structured programming.

Limitations:

1. **Slower Execution** – Needs to be compiled or interpreted, making it slower than low-level languages.
2. **Less Control Over Hardware** – Cannot directly manipulate hardware resources efficiently.
3. **More Memory Usage** – Uses extra memory for abstraction, making it less efficient in resource-constrained environments.
4. **Dependency on Compilers/Interpreters** – Needs an extra layer (compiler/interpreter) to convert code into machine language.

Low-Level Languages

Advantages:

1. **Fast Execution** – Directly interacts with hardware, making it faster than high-level languages.
2. **Greater Hardware Control** – Suitable for system programming (e.g., OS, firmware, embedded systems).
3. **No Need for Compilation** – In machine code, no translation is needed; in assembly, minimal translation is required.

Limitations:

1. **Complex & Hard to Learn** – Requires deep knowledge of hardware architecture.
2. **Not Portable** – Code is specific to a particular processor or architecture.
3. **Difficult Debugging & Maintenance** – Harder to read, understand, and modify compared to high-level languages.

Source Code: The High level code written by user is known as source code

Object Code: The Machine language Code (binary) we get after compiling the high level code is known as Object Code

Some languages use compilers , others use interpreter, Java uses both:

In java, the user written source code (High Level Code) is compiled by java compiler and byte code is created , this byte code is again given to java interpreter which converts the byte code into object code (Machine Level Code)

Source Code → Compiler → Byte Cde → Interpreter → Object Code

** Extension defines type of a file, for example music files are .mp3 , video files are .mp4 , MS WORD is .docx , powerpoint has .ppt , **Just like that extension for java source code is .java**

and extension for byte code is .class

Example: if a program has a class name "SUM" then the source code's file name will be SUM.java and file name for byte code will be SUM.class

JVM: Java interpreter is known as JVM (JAVA Virtual Machine)