Name/ Nouran Sameh Mohamed

Search

1. **Low level languages & high level languages**

	Low level	High level
Definition	• Languages that are close to machine language and interact directly with hardware • Require deep knowledge of computer architecture (e.g., processors, memory).	• Languages that are closer to human language and easier to understand. • Do not require deep knowledge of hardware
Ease of Use	• Difficult to understand and write due to their proximity to machine language • Require more effort to write simple programs	• Easy to understand and write. • Simple programs can be written quickly
Performance	• Faster execution because they are closer to hardware • Ideal for applications requiring high performance, such as operating systems and games	• Provide full control over hardware. • Allow direct access to memory and processors

Hardware Control	• Provide full control over hardware • Allow direct access to memory and processors.	• Provide an abstraction layer between the program and hardware. • Do not allow direct hardware control
Portability	• Not portable across different operating systems (e.g., Windows, Linux) because they are hardware-dependent • Example: A program written in Assembly will not work on a different processor	• Portable across systems, provided a suitable compiler or interpreter is available. • Example: A program written in Python can run on Windows, Linux, and Mac
Applications	• Operating systems. • Device drivers. • Embedded systems	• Web applications. • Mobile apps. • Artificial intelligence programs.
Examples	• Assembly language • Machine Code	• Python • Java

2. Interpreter & compiler

	Interpreter	compiler
Definition	A program that executes code line by line, translating and running each line immediately	A program that translates the entire source code into machine code or an intermediate code before execution

Execution Process	• Reads, translates, and executes the code line by line. • Stops execution if an error is encountered in a line.	• Translates the entire source code into an executable file (e.g., .exe) or intermediate code (e.g., Java bytecode). • Checks for errors in the entire program before execution
Speed of Execution	• Slower execution because each line is translated and executed at runtime. • Suitable for scripting and rapid development.	• Faster execution because the entire program is translated into machine code before execution. • Ideal for performance-critical applications.
Error Detection	• Detects errors line by line and stops execution at the point of error. • Easier for debugging during development.	• Detects errors in the entire program before execution. • Requires recompilation after fixing errors.
Portability	• Highly portable because the interpreter itself handles platform-specific details. • Example: A Python script can run on any system with a Python interpreter.	• Less portable because the compiled code is often platform-specific. • Example: A C program compiled for Windows won't run on Linux without recompilation.

Memory Usage	• Uses less memory during execution because it processes code line by line. • Suitable for systems with limited resources	• Uses more memory because the entire program is loaded into memory before execution. • Better for systems with sufficient resources
Development Cycle	• Faster development cycle because there's no need for a separate compilation step. • Example: Writing and testing Python scripts interactively.	• Slower development cycle due to the need for compilation before execution. • Example: Writing, compiling, and testing C programs.
Examples	• Python • JavaScript	• C • Java

3.programming & scripted

	Programming	scripted
Definition	Languages used to write software applications, where the code is compiled into machine code or intermediate code before execution	Languages used to write scripts, which are interpreted and executed line by line at runtime.
Execution Process	Code is compiled into an executable file or intermediate code before execution.	Code is interpreted and executed line by line at runtime.

Speed of Execution	• Faster execution because the code is compiled into machine code or intermediate code before execution. • Ideal for performance-critical applications	• Slower execution because the code is interpreted and executed line by line at runtime. • Suitable for tasks where speed is not critical
Error Detection	Errors are detected during the compilation phase before execution.	Errors are detected at runtime as the interpreter executes each line.
Portability	Less portable because the compiled code is often platform-specific	Highly portable because the interpreter itself handles platform-specific details.
Memory Usage	• Uses more memory because the entire program is loaded into memory before execution. • Better for systems with sufficient resources.	• Uses less memory during execution because it processes code line by line. • Suitable for systems with limited resources.
Development Cycle	• Slower development cycle due to the need for compilation before execution. • Example: Writing, compiling, and testing C programs	• Faster development cycle because there's no need for a separate compilation step. • Example: Writing and testing Python scripts interactively.
Examples	• C • Java	• JavaScript • python

3.Open Source & not Open Source

	Open Source	Not Open Source
Definition	Software whose source code is made available to the public, allowing anyone to view, modify, and distribute it.	Software whose source code is not shared with the public, and only the compiled version is distributed
Access to Source Code	• Source code is freely available to the public. • Users can modify and customize the software to suit their needs. • Example: The source code of Linux is available on platforms like GitHub	• Source code is not available to the public. • Users cannot modify or customize the software. • Example: The source code of Microsoft Windows is not publicly accessible
Cost	Generally free of charge, though some open-source software may offer paid support or premium features.	Usually requires a purchase or subscription fee.
Customization and Flexibility	• Highly customizable as users have access to the source code. • Developers can modify the software to add features or fix bugs. • Example: Customizing the Linux kernel for specific hardware.	• Limited customization as users do not have access to the source code. • Customization is restricted to the options provided by the vendor. • Example: Customizing Microsoft Office is limited to the features provided by Microsoft

Support and Community	• Support is often community-driven, with forums, documentation, and user contributions. • Large communities can provide extensive resources and help	Support is typically provided by the vendor, including official documentation, customer service, and professional support.
Security	• Security can be both a strength and a weakness. • The open nature allows many eyes to review and improve the code, potentially leading to faster identification and fixing of vulnerabilities	• Security relies on the vendor's ability to identify and fix vulnerabilities. • Users must trust the vendor to handle security issues promptly.
Development and Innovation	• Encourages collaboration and innovation as developers from around the world can contribute. • Rapid development cycles due to community involvement	• Development is controlled by the vendor, which can lead to slower innovation cycles. • Features and updates are determined by the vendor's roadmap
Examples	• Linux (Operating System) • Apache HTTP Server (Web Server) • Mozilla Firefox (Web Browser) • WordPress (Content Management System)	• Microsoft Windows (Operating System) • Adobe Photoshop (Graphic Design Software) • Microsoft Office (Productivity Suite) • Oracle Database (Database Management System)

5.support OOP & not support OOP

	Support OOP	Not support OOP
Definition	Languages that are designed to support the principles of Object-Oriented Programming, such as encapsulation, inheritance, and polymorphism	Languages that do not inherently support the principles of Object-Oriented Programming.
Core Concepts	• Encapsulation: Bundling data and methods that operate on the data within a single unit (class). • Inheritance: Creating new classes (derived classes) from existing ones (base classes). • Polymorphism: Allowing objects to be treated as instances of their parent class rather than their actual class.	These languages typically use procedural programming paradigms, focusing on functions and procedures rather than objects
Code Organization	Code is organized around objects and classes, making it easier to manage and reuse.	Code is organized around functions and procedures, which can lead to less modular and reusable code.
Reusability and Maintainability	• High reusability through inheritance and polymorphism. • Easier to maintain and extend due to encapsulation and modularity.	• Lower reusability as there is no inherent support for inheritance or polymorphism • Can be harder to maintain and extend, especially as the codebase grows.

Complexity and Learning Curve	• Can be more complex due to the additional concepts and syntax related to OOP. • Steeper learning curve for beginners	• Generally simpler and more straightforward, focusing on procedural logic. • Easier to learn for beginners
Performance	Can have some overhead due to the additional layers of abstraction (e.g., virtual functions in C++).	Typically more performant as they are closer to the hardware and have less overhead.
Examples	• Java • python	• Fortran • C