Lecture Two: The Need for Programming Languages

In our daily lives, we communicate with people in multiple languages—Arabic, French, English… Yet when we want to communicate with a computer, these languages are of no use, because the computer understands only strict logic and precise instructions. Here, programming emerges as a means of communication with the machine: it is the way we express our ideas in a language the computer can understand, enabling it to perform a specific task without errors or ambiguity.

It is important, however, to distinguish between **programming** and a **programming language**. Programming is the act itself: the logical process of thinking and formulating the instructions we want the computer to execute. A programming language, on the other hand, is the tool we use to write these instructions in a way the system can interpret. Just as writing requires a language, programming requires an artificial, precise language that transforms ideas into executable commands.

These languages are not used for decoration or literary expression, but for organizing logic, setting conditions, and defining steps with accuracy. From this necessity arises the importance of understanding programming languages—the central theme of this lecture.

First: The Computer Does Not Guess – Programming as a Logical Communication Tool with the Machine

The computer is often perceived as an intelligent device capable of understanding and inference, but this view contradicts the essence of its operation. A computer does not "understand" what it is told; it executes literally what is written for it, with no ability to interpret or improvise.

It resembles an employee receiving a vague instruction such as: *"Put the envelope in the drawer if it is not open."* At first glance, this sentence seems logical, but in reality, it is not executable—because if the envelope is not open, it is already closed, and the instruction becomes redundant. This kind of linguistic ambiguity confuses humans and paralyzes computers.

Programming languages provide a decisive solution to this problem. They compel the programmer to formulate instructions with logical precision, leaving no room for ambiguity. For example: *"If the envelope is not open, place it in the drawer. If it is open, check its contents first."* With this formulation, execution becomes possible, clear, and unambiguous—exactly what the computer requires to function efficiently.

Second: Machine Language and the Dilemma of Understanding – Why Do We Need Programming Languages?

At the core of the relationship between humans and computers lies a profound linguistic dilemma: how can a device that neither understands words nor perceives meanings execute human instructions? A computer does not process language as we do; it only interprets electrical signals internally translated into sequences of binary digits: zero (0) and one (1). This is **machine language**—a strict binary system that recognizes only two states: signal present or absent.

Since humans cannot think directly in this binary language, programming languages emerged as an artificial intermediary, enabling us to write instructions comprehensible to us but convertible into the binary form the computer understands.

Take, for example, the simple word *"Hi"*:

1. Each character is converted into its ASCII code:
    - H = 72
    - i = 105
2. These numbers are then converted into binary form:
    - 72 = 01001000
    - 105 = 01101001
3. The final representation understood by the computer:
    - "Hi" = 01001000 01101001

This example shows that the computer does not know the meaning of *"Hi"*, nor does it sense the greeting. It merely receives a sequence of zeros and ones and executes tasks based on these digital representations.

Thus, programming languages are not just technical tools; they are logical languages that compel us to think with precision, translating our ideas into instructions that can be converted into the binary system driving everything inside the machine.

Third: Levels of Programming Languages – From Machine Language to Human Language

Programming languages are classified according to their closeness to machine logic or human logic into **low-level** and **high-level** languages.

Low-level languages, such as **Assembly**, bring the programmer closer to the computer's internal architecture, granting direct control over the processor, memory, and registers. However, they require precise technical knowledge and detailed specification of every step. For example, to print the character **"A"** on the screen, the programmer must write:

```
mov ah, 0x0E   ; specify the print function
mov al, 'A'    ; load the character into the register
int 0x10       ; call the screen interrupt
```

This type of programming resembles speaking directly in the machine's language, where there is no room for abstraction or simplification, and every operation must be explicitly written.

High-level languages, such as **Python** and **Java**, emerged to simplify programming and bring it closer to human thinking. They hide technical details behind user-friendly interfaces, allowing programmers to express program logic without worrying about the computer's internal architecture. For the same task—printing the character "A"—in Python it suffices to write:

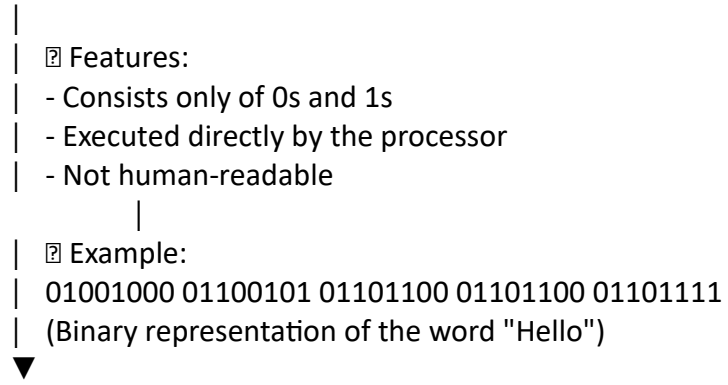
```
print("A")
```

Here, there is no need to know about interrupts, registers, or memory addresses. The language handles all of that, focusing instead on what the programmer wants to achieve, not how it is executed inside the machine.

This simplicity makes high-level languages ideal for education, application development, and data analysis, where smooth expression is preferred over precise hardware control. Thus, the difference between the two types illustrates the transition of programming from **machine language to human language**, and from literal execution to logical expression.

▲

```
|
|  High-Level Languages (Close to Human Language)
| ————————————————————
|  - Python
|  - Java
|  - C++
|  - JavaScript
          |
|  Features:
|  - Easy to read and write
|  - Resemble natural language
|  - Hide machine details
|  - Suitable for general applications
|
|  Example:
|  print("Hello World")
 |
          |
|  Low-Level Languages (Close to Machine Language)
| ————————————————————
|  - Assembly
|
|  Features:
|  - Closer to machine language
|  - Require precise technical knowledge
|  - Used for hardware control
|
|  Example:
|  mov ah, 0x0E
|  mov al, 'H'
|  int 0x10
 |
      |
|  Machine Language
| ————————————————————
|  - Binary Code
```



This pyramid illustrates how programming languages progress from machine language at the base—understood only by the computer—to high-level languages at the top, where humans can easily express their ideas. The higher we move up the pyramid, the greater the human readability of the language, and the lesser the need for precise technical knowledge.

Fourth: Compiler vs. Interpreter – How Does the Machine Execute Our Instructions?

In programming, understanding how code is executed is fundamental. Here arises the distinction between the **compiler** and the **interpreter** as two main tools for translating source code into executable instructions.

A **compiler** translates the entire source code at once into an independent executable file before execution begins. This means that errors are detected during compilation, and the resulting program runs quickly since it does not need to be re-translated each time. Famous languages that rely on compilers include **C** and **C++**, where the code is converted into a `.exe` file (or equivalent) that can be run directly without the original source code.

An **interpreter**, on the other hand, processes the source code line by line, translating and executing each instruction immediately without producing a standalone executable file. This approach makes experimentation and modification easier but slows down execution because translation occurs every time. Languages such as **Python** and **JavaScript** use interpreters, allowing programmers to test instructions instantly and observe results without prior compilation. While interpreters are more flexible in educational and rapid development environments, compilers remain the preferred choice for applications requiring high performance and execution stability.

Fifth: Classification of Programming Languages – General-Purpose vs. Domain-Specific

As programming evolved, two major categories of languages emerged:

- **General-Purpose Programming Languages (GPPLs):**

These are designed to be flexible and applicable across a wide range of domains. Examples include **Python**, **Java**, and **C++**. They are used in web development, artificial intelligence, database management, and even embedded systems. Their strength lies in comprehensive structures, extensive libraries, and adaptability to diverse needs. Since they do not assume a predefined task, they give programmers great freedom in design and implementation, making them the first choice in education, research, and enterprise development.

- **Domain-Specific Languages (DSLs):**

These are tailored to solve problems within a narrow field. Examples include **SQL** for database management, **HTML** for web page description, and **MATLAB** for numerical and engineering analysis. DSLs are not intended to build complete systems but to perform one function with high efficiency. They resemble specialized tools in a workshop: not suitable for everything, but unmatched in their domain. Often, they are used alongside general-purpose languages, integrated into larger systems to provide precise functionality. Their specialization makes them indispensable in institutions that rely on repetitive, well-defined tasks such as data analysis or user interface design.

Sixth: The Fundamental Elements of Any Programming Language

To understand programming in a practical way, it is not enough to know the theoretical concepts; we must also become familiar with the fundamental elements that shape any programming language.

1. Integrated Development Environment (IDE)

In the world of programming, a programmer does not work in isolation but relies on intelligent tools that help transform ideas into executable instructions. One of the most important of these tools is the Integrated Development Environment (IDE), a platform that combines a code editor, compiler or interpreter, debugger, and instant execution capabilities. This transforms programming from a complex technical process into an organized creative experience. Some IDEs are general and support multiple languages, such as Visual Studio Code and Eclipse, while others are specialized for a specific language, such as PyCharm for Python and IntelliJ IDEA for Java. These provide precise tools that simplify writing, package management, and performance analysis. With the rise of mobile devices, lightweight IDEs such as PyDroid3 have appeared, enabling programmers to write and run Python code directly on smartphones, expanding learning and experimentation beyond the traditional computer. Choosing the right environment is like selecting the right tool in a workshop: the more compatible it is with the language and project, the more effective and precise the work becomes, allowing the programmer to focus on program logic rather than execution details.

2. Code

Code is a sequence of logical instructions written by the programmer in a specific programming language to direct the computer to perform a task. Code is not just writing—it is a precise way of thinking that transforms an idea into executable steps. Each line represents a clear command, such as “calculate,” “print,” “check condition,” or “repeat.”

Code is written in artificial languages such as Python or Java, and it resembles a cooking recipe: if the instructions are clear, the computer executes them accurately; if they are vague or illogical, execution stops or errors appear. In code, there is no room for ambiguity—the computer does not understand "maybe" or "if appropriate," but only clear conditions such as:

```
if temperature > 100:
    print("Too hot")
```

Thus, code is the logical translation of an idea, the medium through which humans express their intent to perform tasks inside a machine that understands only binary logic.

3. **Instructions**

Instructions in programming are the commands we write inside a program for the computer to execute precisely, one after another, without interpretation or guesswork. The computer does not understand intentions or infer meanings—it only executes what is written in a clear format. These instructions form the code, a sequence of logical steps representing execution. Each instruction corresponds to a specific action: printing, calculating, testing a condition, or repeating a process. For example, an instruction may tell the computer to print the word "Bonjour" on the screen. A program is therefore a set of ordered instructions executed according to strict logic, making code the medium through which humans express ideas in a language the machine understands.

4. **Syntax**

In programming, it is not enough for an idea to be correct—it must also be written correctly. This is where syntax comes in: the rules that define how instructions must be written inside code. Each programming language has its own syntax, similar to grammar rules in human language. If we break these rules, even if the idea is logical, the computer will not understand and errors will occur. For example, in Python we cannot write:

```
print "Hello"
```

Instead, we must write:

```
print("Hello")
```

The parentheses here are not optional but part of the language's syntax. Learning programming therefore requires not only logical thinking but also respect for each language's writing rules. Syntax teaches discipline and precision, helping programmers communicate with the computer in a language it fully understands.

Together, these elements form the foundation of any program and represent the starting point for understanding how ideas are transformed into executable instructions within a digital environment.