ANALYSIS AND MACHINE LEARNING TECHNIQUES WITH BASICS OF COMPUTER SCIENCE AND STATISTICAL LEARNING - ANALYSIS AND MACHINE LEARNING TECHNIQUES WITH BASICS OF COMPUTER SCIENCE AND STATISTICAL LEARNING

Prof. Loriano Storchi
loriano@storchi.org
https://www.storchi.org/

Contents

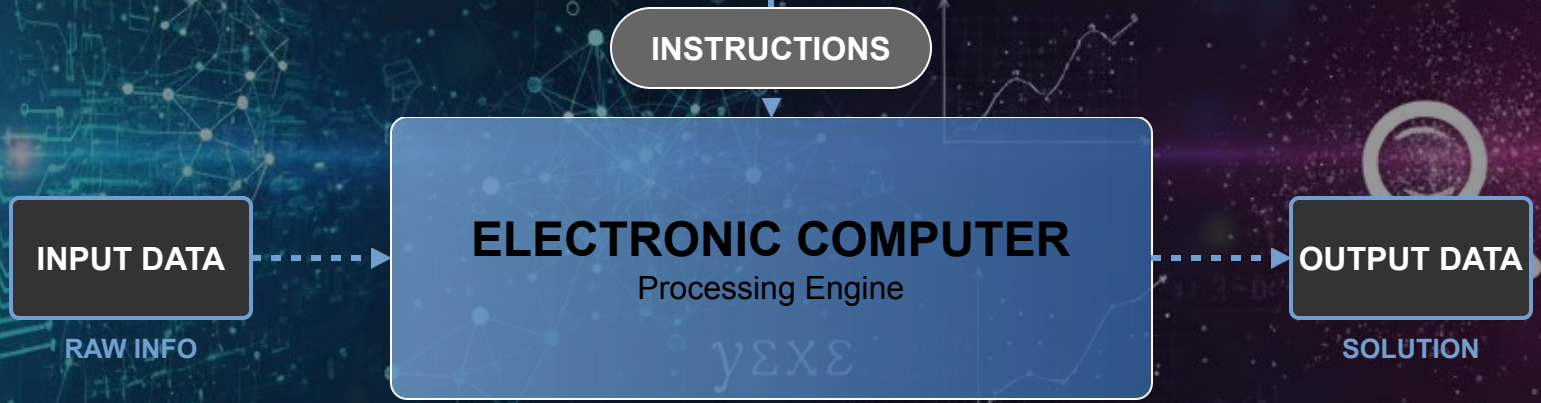
Brief introduction to programming languages and computer complexity

Algorithms

- Algorithms describe how information is transformed. Computer science deals with their theory, analysis, design, efficiency, implementation, and application.
- An algorithm is a formal process that solves a given problem through a finite number of steps. The term derives from the Latin transcription of the name of the Persian mathematician al-Khwarizmi, who is considered one of the first authors to have referred to this concept. Algorithm is a fundamental concept in computer science, primarily because it underlies the theoretical notion of computability: a problem is computable when it can be solved using an algorithm. (Wikipedia)

Programming: From Algorithm to Execution

Programming is the activity of instructing a computer to execute specific actions on input data to solve problems and produce outputs. It is the practical implementation of an algorithm.



"Implementation of a given algorithm to solve a problem."

LANGUAGES AND LAYERS

Level L0 represents the actual computer and the machine language it can directly execute. Each higher level represents an abstract machine. The programs (instructions) of each higher level must either be translated into instructions from one of the lower levels, or interpreted by a program running on an abstract machine of a strictly lower level.

MACHINE LANGUAGE AND ASSEMBLY

Machine Language

Every computer is capable of interpreting a low-level language called machine language. Instructions (OPCODEs) are simple sequences of bits that the processor interprets and follows a precise series of operations. Each instruction at this level consists of extremely basic operations.

```
00000000000000000000000001000000
00000000001000000000000001000100
00000010000000010000000000000000
00000001000000000000000000111100
```

Assembly Language

To make programming more immediate, the first step was the introduction of ASSEMBLY, where binary code is replaced by mnemonic instructions that are more easily usable by humans.

```
PSEUDO-ASSEMBLY
z : INT;
x : INT 8;
y : INT 38;
LOAD R0,x;
LOAD R1,y;
ADD R0,R1;
STORE R0,z;
```

Understanding Language Types

Programming languages today are diverse and purpose-built for specific computational tasks.

Natural Languages

Evolved spontaneously and highly expressive.

"la vecchia porta la sbarra"

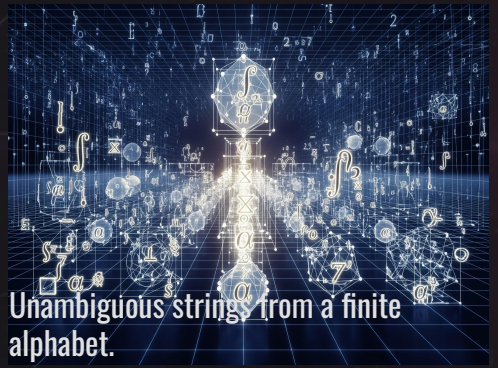
Inherently ambiguous due to context-dependent meanings.

Artificial Languages

Created with a specific birth date and defined authors.

Can be categorized as either **formal** or **non-formal** based on their structure.

Formal Languages



Unambiguous strings from a finite alphabet.

Defined by precise **Syntax** and **Semantics**, allowing algorithmic verification.

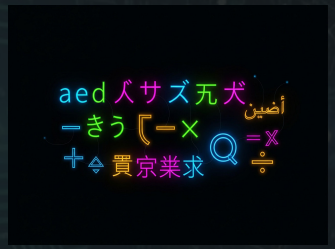
LANGUAGES

- Today, there are numerous programming languages. Generally, each language is more or less suited to a specific purpose.
- **Natural languages** : they are given spontaneously, they are extremely expressive but ambiguous: **“la vecchia porta la sbarra”**
- **Artificial languages** : These are languages that have a specific birth date and a list of authors. Artificial languages can be **formal** or non-formal.
- **Formal languages** : unambiguous languages consisting of a finite set of strings constructed from a finite alphabet. It is a language for which the form of sentences (**syntax**) and their meaning (**semantics**) are precisely and unambiguously defined. It is therefore possible to define an algorithmic procedure capable of verifying the **grammatical correctness** of sentences.

Defining Language Rigorously

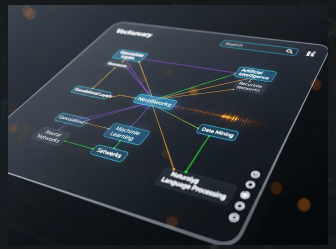
To formally establish a language, four fundamental conceptual tools are required:

Alphabet



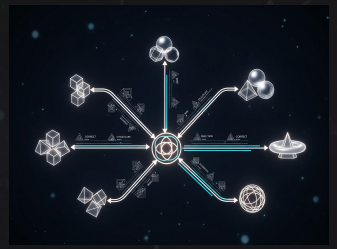
The set of **basic symbols** required to construct the fundamental units or words of a language.

Lexicon



The complete **vocabulary** and rules for word formation within the language system.

Syntax



Grammatical rules that determine if a specific sequence of words (a sentence) is structurally valid.

Semantics

Defines the **actual meaning**. Example: `int a[5];` in C reserves memory for 5 integers.

LANGUAGES

- To rigorously define a language, some basic tools are needed:
  - **Alphabet** : the set of basic symbols needed to make up words
  - **Lexicon** : set of rules needed to write the words of a language (vocabulary)
  - **Syntax (grammatical** rules): set of rules that allow us to establish whether a sentence (set of words) is correct
  - **Semantics** : defines the meaning of a syntactically correct "sentence" for example: **int a[5];** in C language it allows to reserve space in memory necessary to contain 5 integers

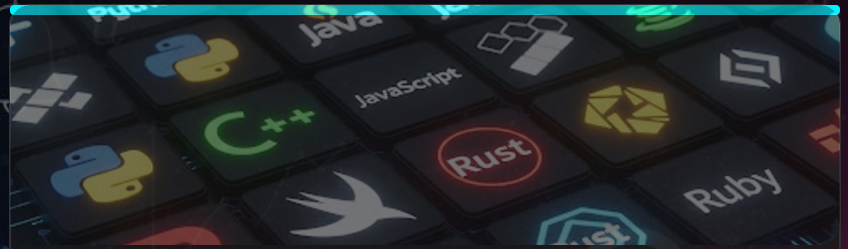
High-Level Programming Languages

Moving away from processor logic towards human readability and machine independence.

Evolution from Hardware

Unlike Low-Level (Assembly) or Machine Languages, high-level languages are built for efficiency and readability.

They bridge the gap between human logic and binary execution through abstraction.



Modern Classification

Thousands of languages exist, yet only about ten dominate the industry for specific paradigms.

Common Languages:

PYTHON JAVA C++ C# FORTRAN

PASCAL BASIC OBJECTIVE-C

LANGUAGES

- We have already mentioned Artificial Languages, Machine Language and Low-Level Language (ASSEMBLY)
- High-level languages move away from processor logic and are built to be simple, efficient, and readable, as well as machine-independent.
- There are many programming languages available today, although only about ten are actually used.
- Below we provide a brief classification of these languages (C, C++, C#, JAVA, PYTHON, FORTRAN, PASCAL, BASIC, Objective-C, and many others). It's clear that each programming paradigm is more or less suited to a specific purpose.


LANGUAGE CLASSIFICATION

LANGUAGE CLASSIFICATION

- Imperative: Do this, then do that, and change this state.
  - Procedural: Group the "doing" into reusable functions.
  - Object-Oriented: Group the "doing" and the "state" together into objects.

- Declarative: Here is the truth, now give me the answer.
  - Functional: The truth is defined by mathematical functions.
  - Logic: The truth is defined by facts and logical rules.


IMPERATIVE

Imperative programming is a paradigm that uses a step-by-step sequence of commands to explicitly define how a program changes its state to achieve a desired result

IMPERATIVE LANGUAGES

- The fundamental component of the program is the instruction, and each instruction indicates the operation to be performed. The individual instructions operate on the program's data.
- The instructions are executed one after the other
- Every program is made up of two fundamental parts: the data declaration and the algorithm, understood as a sequence of operations.
- Each statement is an order (declarative programming the program is a series of statements)
- In fact, from a syntactic point of view, many imperative languages use imperative verbs (i.e. PRINT, READ, ...)

IMPERATIVE LANGUAGES

```
1   PROGRAM SimpleAddition
2       ! 1. Initialize state / Declare variables
3       ! We tell the computer to reserve memory for three real (decimal) numbers.
4       REAL :: A, B, C
5       ! Prompt the user so they know what to do
6       PRINT *, "Please enter two numbers separated by a space:"
7       ! 2. Read the input and change the state of A and B
8       READ *, A, B
9       ! 3. Explicit calculation and mutation
10      ! We calculate A + B and store the result in the memory location for C
11      C = A + B
12      ! 4. Output the final state
13      PRINT *, "The sum (C) is: ", C
14  END PROGRAM SimpleAddition
```

So a series of instructions read A and B, calculate C as the sum of A plus B and finally print the result

PROCEDURAL PROGRAMMING

- We can consider it a sub-paradigm of imperative programming.
- The concept of subroutines or functions is introduced.
- This introduces the possibility of creating portions of source code useful for performing specific functions.
- These subprograms can receive input parameters and return output values.

PROCEDURAL PROGRAMMING

functions.c — THE PROCEDURES

```c
// Reusable block 1
float calculateTax(float amount) {
   return amount * 0.20;
}

// Reusable block 2
void printReceipt(float base, float tax) {
   float total = base + tax;
   printf("Total: $%f", total);
}
```

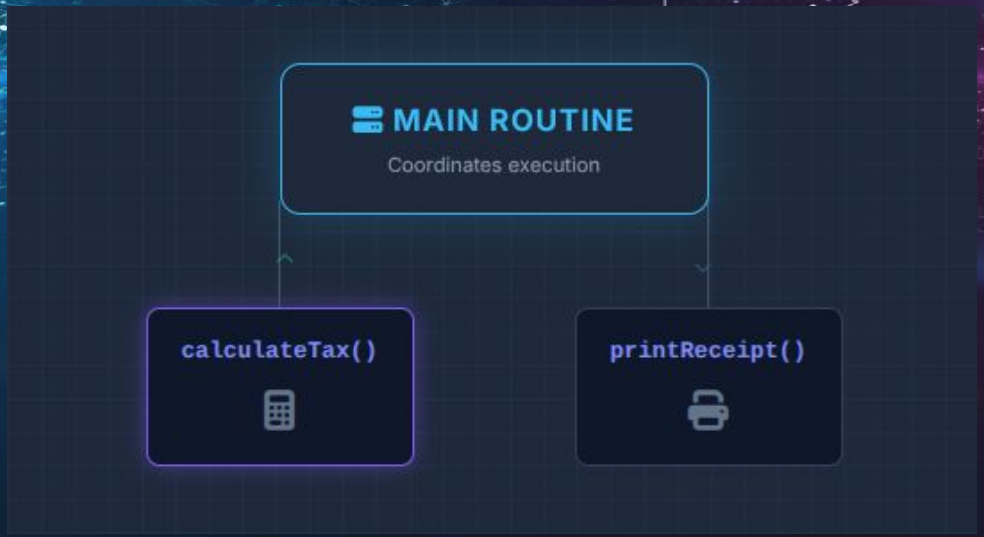
main.c — THE COORDINATOR

```c
int main() {
   float cartValue = 100.0;

   // Call Procedure 1
   float myTax = calculateTax(cartValue);

   // Call Procedure 2
   printReceipt(cartValue, myTax);

   return 0;
}
```

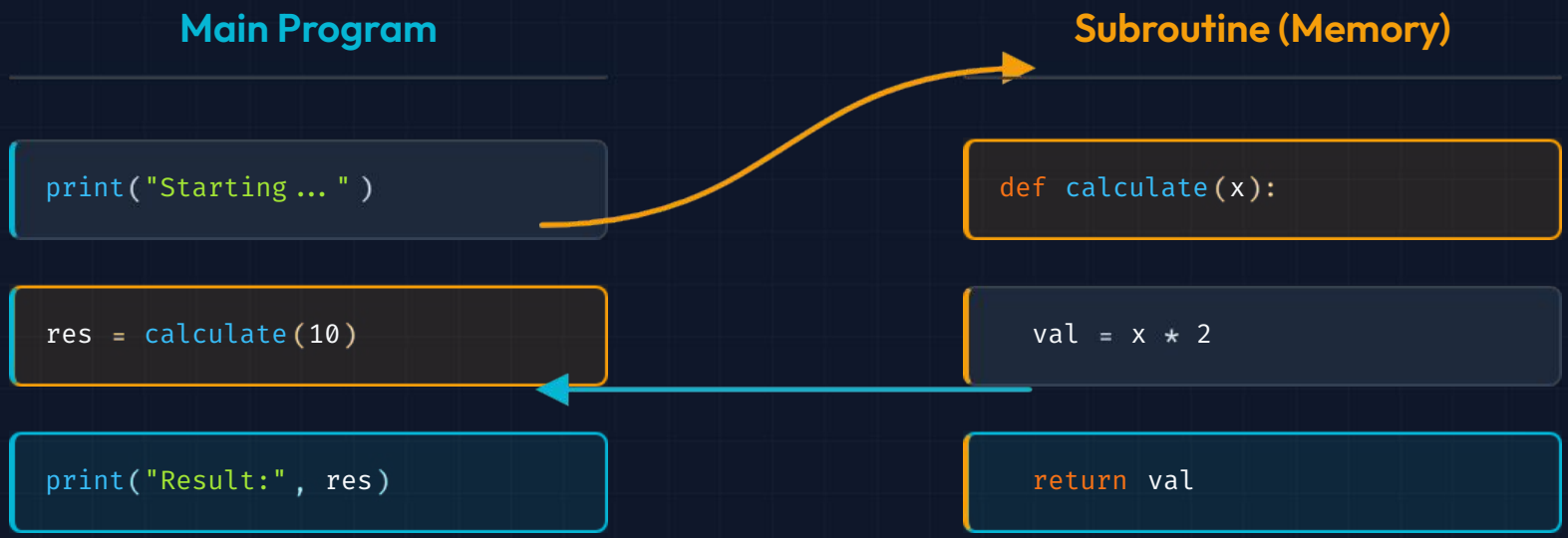
Using subroutines and functions (code reusability, function libraries)
Instead of writing every step in the main execution block, we define isolated functions that take inputs, perform a specific task, and return an output. The main() block then acts as a coordinator.

PROCEDURAL PROGRAMMING

The concept: The Main program doesn't do all the heavy lifting. It sends arguments (data) down to specific procedures. The procedure processes the data and returns the result back up to Main, which then decides what to do next. This makes the code highly modular and reusable.



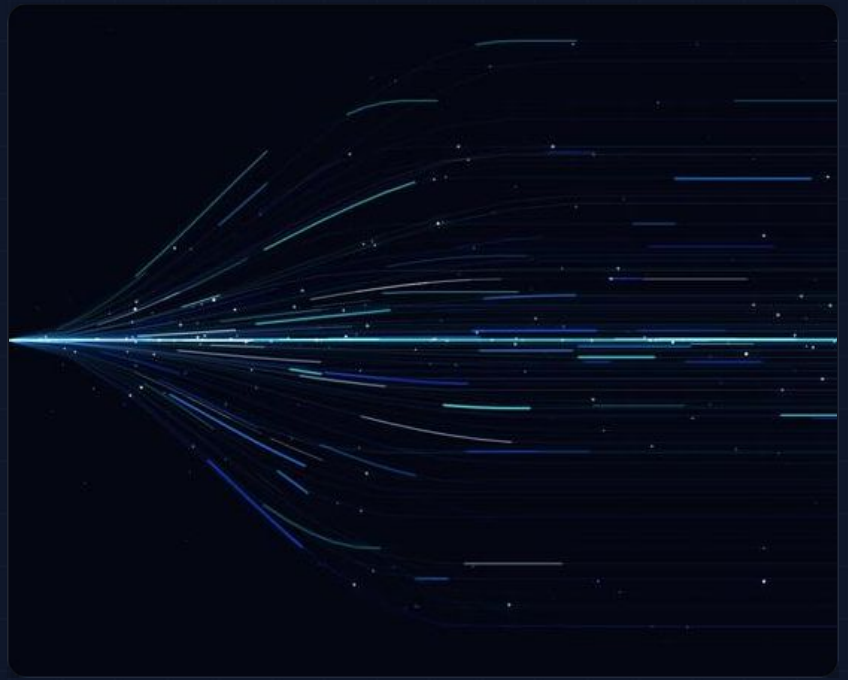
Visualizing the Jump & Return



How does it remember where to return?

When a program jumps to a subroutine, it must pause its current work and remember exactly where it left off. It does this using the **Call Stack**.

- **The Push:** Before jumping, the computer pushes a "Return Address" (a bookmark) onto a hidden stack in the system memory.
- **The Execution:** The CPU completely diverts its attention to the new function, allocating new local variables.
- **The Pop (Return):** When the function hits the return keyword, the CPU pops that address off the stack, retrieves the data, and resumes perfectly.



STRUCTURED PROGRAMMING

- We can consider it a sub-paradigm of imperative programming.
- In practice, the programmer is restricted to using only canonical control structures that do not include unconditional branch (GOTO) instructions. Therefore, the language syntax prohibits the use of structures that do not meet certain constraints. (Not only that.)
- Using the GOTO statement inevitably leads to poor code readability (spaghetti-code)

STRUCTURED PROGRAMMING

- Example from Wikipedia

```
10 dim i
20 i = 0
30 i = i + 1
40 if i <= 10 then goto 70
50 print "Programma terminato."
60 end
70 print i & " al quadrato = " & i * i
80 goto 30
```

```
function square(i)
    square = i * i
end function
dim i
for i = 1 to 10
    print i & " al quadrato = " & square(i)
next
print "Programma terminato."
```

OBJECT-ORIENTED PROGRAMMING

- We can consider it a sub-paradigm of imperative programming.
- This programming paradigm allows us to define Software **Objects** that can interact with each other.
- Organizing software as objects allows for easier reuse and better organization of large projects.
- OOP languages involve grouping part of the source code into classes. Each class contains data and methods (functions) that operate on that data. Classes are abstract models that are invoked at runtime to create or instantiate software objects.
- An object-oriented language allows you to implement three basic mechanisms using the language's native syntax: **encapsulation, polymorphism, and inheritance** .

OBJECT-ORIENTED PROGRAMMING

Attributes (Variables)

The variables that hold the geometric shape's properties.

```
double    sideLength ;

String    color ;

int       x_position ;

int       y_position ;
```

Methods (Functions)

The mathematical functions or actions the shape can perform.

```
double getArea() {
   return this.sideLength *
   this.sideLength;
}
```

```
void resize(double newSide) {
   this.sideLength = newSide;
}
```

OBJECT-ORIENTED PROGRAMMING

Geometry.java

```java
public static void main(String[] args) {

    // 1. Instantiate Object 1 (Small Red Square)
    Square boxA = new Square(5.0, "Red");
    double areaA = boxA.getArea(); // Returns 25.0

    // 2. Instantiate Object 2 (Large Blue Square)
    Square boxB = new Square(10.0, "Blue");
    boxB.resize(12.0); // Change its side length
    double areaB = boxB.getArea(); // Returns 144.0

}
```

OBJECTS IN MEMORY

boxA (Instance)

sideLength: 5.0 color: "Red"

x_position: 0 y_position: 0

boxB (Instance)

sideLength: 12.0 color: "Blue"

x_position: 0 y_position: 0

OBJECT-ORIENTED PROGRAMMING

Encapsulation: A precise separation between the implementation and the class interface. Users of the class (object) do not need to know the implementation details. They use the class through public methods and data, interacting with the object without knowing the implementation details.

OBJECT-ORIENTED PROGRAMMING

Square.java (Bad) — WITHOUT ENCAPSULATION

```
public class Square {
   // Danger: Direct access allowed!
   public double sideLength;
}

// Somewhere in Main:
Square box = new Square();
box.sideLength = -10.0; // Math is broken!
Squares can't be negative.
```

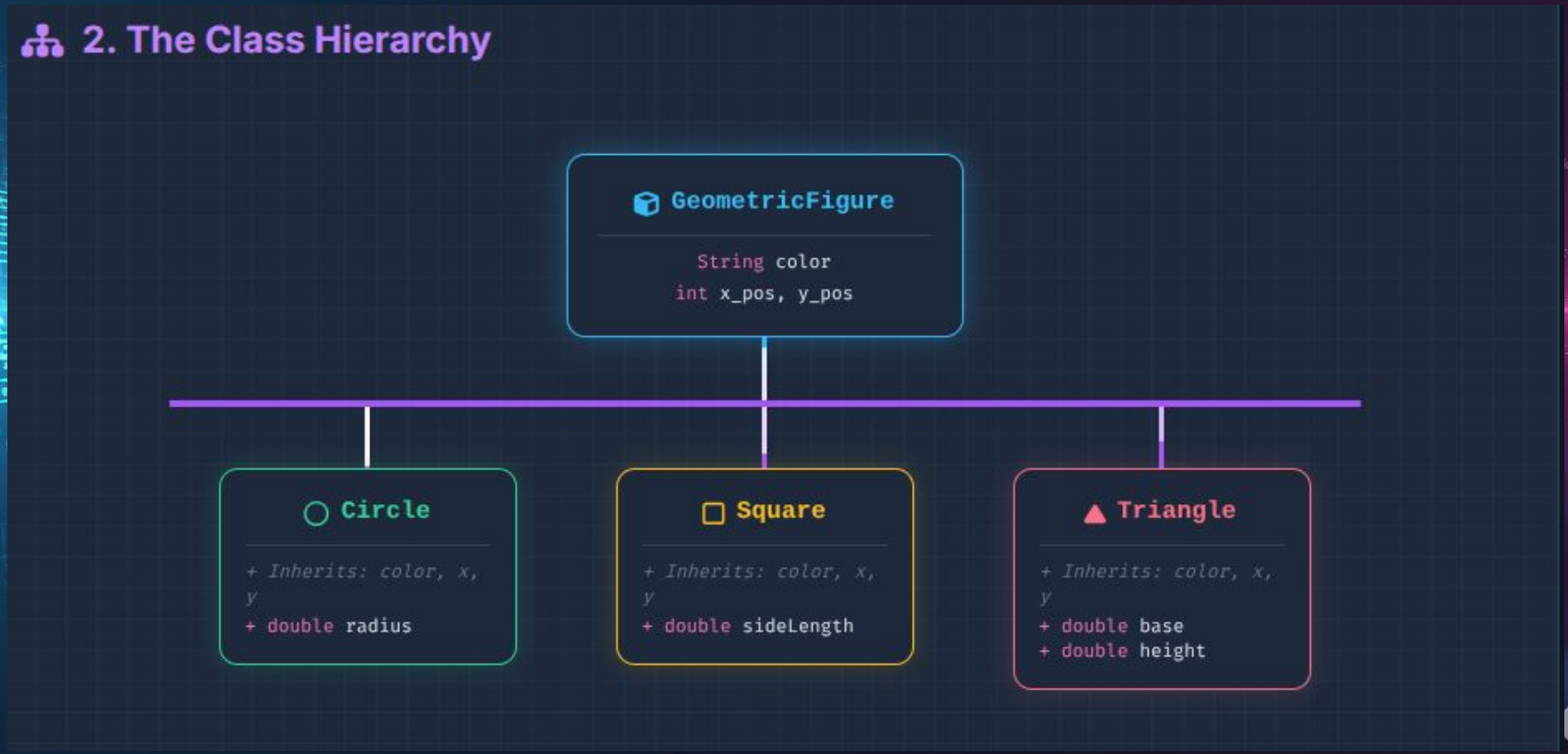
Square.java (Good) — WITH ENCAPSULATION

```
public class Square {
   // Safe: Hidden inside the object
   private double sideLength;

   // The "Gatekeeper" method (Setter)
   public void setSideLength(double s) {
      if (s > 0) {
        this.sideLength = s;
      } else {
        print("Error: Invalid size!");
      }
   }
}
```

OBJECT-ORIENTED PROGRAMMING

- **Inheritance** : A class can inherit from a base class and evolve or specialize its functionality.
  - For example, I can imagine a base class geometric_figure from which classes like triangle, circle, square... derive.
- Classes that derive from a base class inherit all the methods and properties of the base class, but can specialize by defining their own methods and data.
  - For example, if B is a subclass (or more generally a subtype) of A, any program/function that can use A can also use B.

OBJECT-ORIENTED PROGRAMMING



OBJECT-ORIENTED PROGRAMMING

GeometricFigure.java — PARENT / BASE

```java
public abstract class GeometricFigure {
    // Shared attributes for ALL shapes
    protected String color;
    protected int x_pos;
    protected int y_pos;

    // A contract: Every shape MUST have an area
    public abstract double getArea();
}
```

Shapes.java — CHILDREN / DERIVED

```java
// Circle inherits color, x, y
public class Circle extends GeometricFigure
{
    private double radius; // Specialization

    public double getArea() {
        return Math.PI * (radius * radius);
    }
}

// Square inherits color, x, y
public class Square extends GeometricFigure
{
    private double sideLength; // Specialization

    public double getArea() {
        return sideLength * sideLength;
    }
}
```

OBJECT-ORIENTED PROGRAMMING

- **Polymorphism** : We can try to exemplify this concept by saying: multiple definitions of the same function (overloading), classes, and functions that are parametric with respect to the data type. A simple example is function overloading.
- We can formally distinguish at least four types of polymorphism: inclusion, parametric, overloading, and coercion.
- We will only give a few simple examples to help clarify the general concept.

OBJECT-ORIENTED PROGRAMMING

Let's imagine the usual geometric_figure class from which we will have derived two classes: circle and triangle

When the user calls the calculate area method it will perform a certain action (calculating the area in both cases even though they have the same name)

OBJECT-ORIENTED PROGRAMMING

Shapes.java — THE SETUP

```java
// Each shape implements its own math
class Circle extends GeometricFigure {
   public double getArea() {
      return Math.PI * (radius * radius);
   }
}

class Triangle extends GeometricFigure {
   public double getArea() {
      return (base * height) / 2.0;
   }
}
```

Main.java — POLYMORPHISM IN ACTION

```java
// 1. Store different shapes in ONE generic
array
GeometricFigure[] myShapes = {
   new Circle(5.0),
   new Triangle(4.0, 6.0)
};

// 2. Loop through them blindly
for (GeometricFigure shape : myShapes) {
   // The EXACT SAME line of code ...
   // ... does two completely different
   things!
   double area = shape.getArea();
   System.out.println(area);
}
```


DECLARATIVE

FUNCTIONAL PROGRAMMING

- As the name suggests, the execution flow takes the form of evaluating a series of mathematical function evaluations. The program is therefore a set of functions.
- In pure functional languages, there is no concept of assignment, or explicit memory allocation.
- Values are not found by changing the state of the program—there is no value assignment—but by constructing the new state using functions starting from the previous state.
- Used in AI (little or no use in industrial settings)
- Functions can be passed as parameters and returned as a “result” from other functions.

FUNCTIONAL PROGRAMMING

Instead of writing `for` loops that manually change a `total` variable (Imperative), we chain together pure functions that take an array and return a brand new array, never modifying the original data.

functional_sum.js

PURE PIPELINE

```
// 1. Immutable Data (Never changes)
const numbers = [1, 2, 3, 4, 5];

// 2. The Data Transformation Pipeline
const result = numbers
  // WHAT to do (Keep evens)
  .filter(n => n % 2 === 0)
  // WHAT to do (Double them)
  .map(n => n * 2)
  // WHAT to do (Sum them up)
  .reduce((sum, n) => sum + n, 0);

console.log(result); // Outputs 12
console.log(numbers); // Still [1, 2, 3, 4, 5]!
```

FUNCTIONAL PROGRAMMING

Imperative Approach (Manual State)

Instead of writing `for` loops that manually change a `total` variable, we maintain a mutable state.

imperative_sum.js

```
// 1. Manually initialize the state
let result = 0;

// 2. Explicit control flow (HOW to loop)
for (let i = 0; i < numbers.length; i++) {
    // 3. The "filter" step
    if (numbers[i] % 2 === 0) {
        // 4. The "map" step
        let doubled = numbers[i] * 2;
        // 5. The "reduce" step (mutating state)
        result = result + doubled;
    }
}

console.log(result); // Outputs: 12
```

Functional Approach (Pure Pipeline)

We chain together pure functions that take an array and return a brand new array, never modifying the original data.

functional_sum.js

```
// 1. Immutable Data (Never changes)
const numbers = [1, 2, 3, 4, 5];

// 2. The Data Transformation Pipeline
const result = numbers
  .filter(n ⇒ n % 2 === 0)        // Keep evens
  .map(n ⇒ n * 2)                 // Double them
  .reduce((sum, n) ⇒ sum + n, 0); // Sum them up

console.log(result); // Outputs: 12
```

FUNCTIONAL PROGRAMMING

The concept: Think of a water purification plant. The water (data) flows through different filters and chemical processors (functions). The pipes don't change the water's original source, they just output a transformed version of what passed through them.

DECLARATIVE (OR LOGICAL) LANGUAGES

- Compared to the imperative paradigm, the program consists of a series of statements and not of commands.
- The program specifies WHAT you want to achieve, not HOW. The how is left to the executor.
- In practice, the program (or its execution) can be considered as a demonstration of the truth of a statement.

DECLARATIVE (OR LOGICAL) LANGUAGES

Imperative (The HOW)

You provide explicit, step-by-step instructions on how the computer must achieve the goal, carefully managing loops and mutating state.

```
let evens = [];
for (let i = 0; i < arr.length; i++) {
  if (arr[i] % 2 === 0) {
    evens.push(arr[i]);
  }
}
```

Declarative (The WHAT)

You declare the result you want. The underlying engine determines the most efficient way to achieve that outcome.

```
const evens = arr.filter(n => n % 2 === 0);


// No loops, no manual mutation.
// The intent is clear immediately.
```

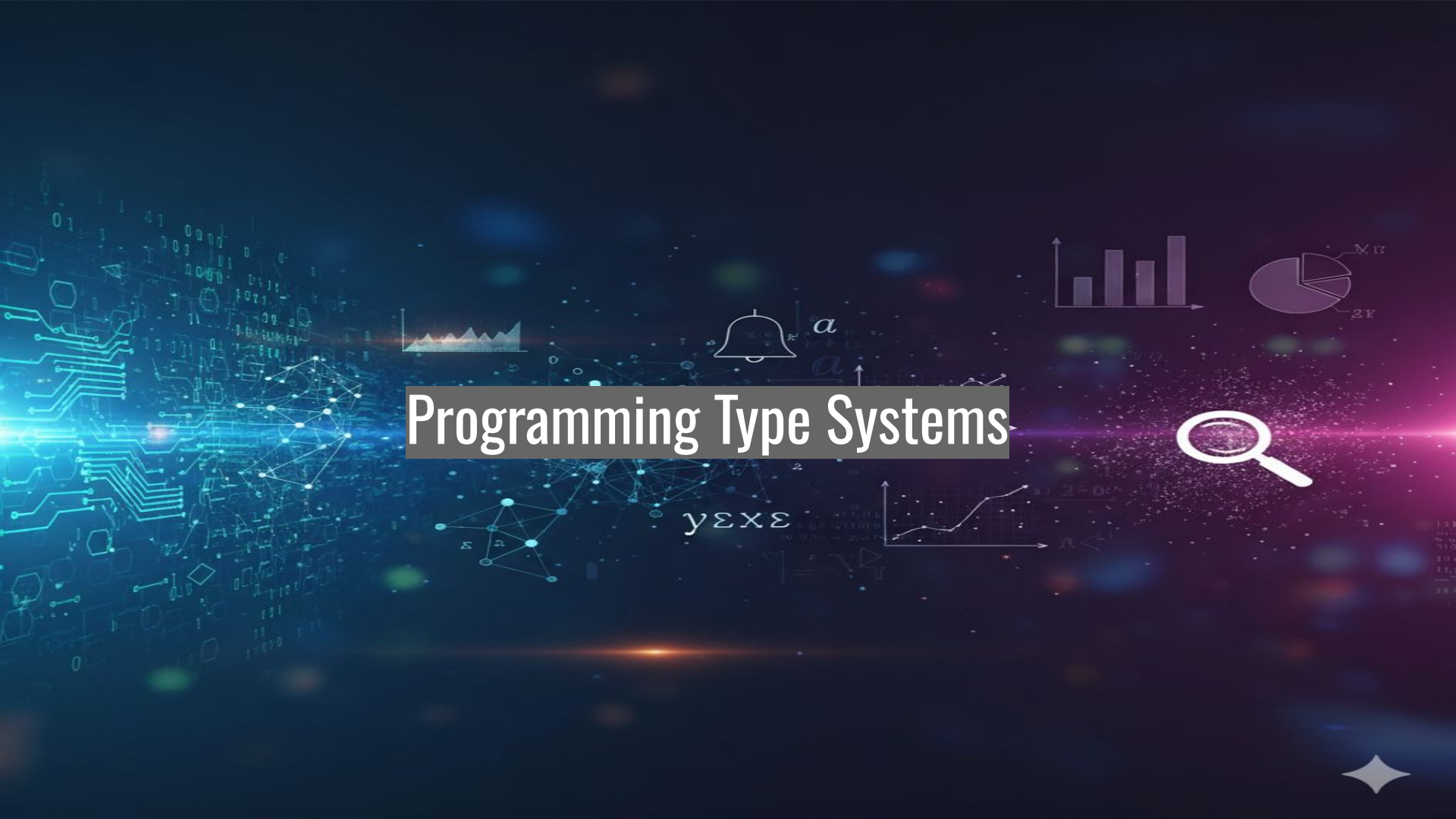
SQL: The Pioneer

Structured Query Language (SQL) is arguably the most famous and widely used declarative language in history.

When you write a query, you never specify the search algorithms, memory allocation, or indexing strategies.

```sql
SELECT name, email
FROM users
WHERE status = 'active'
ORDER BY created_at DESC;
```

The database engine acts as the "chef", figuring out the most optimal execution plan to retrieve your data.


Programming Type Systems

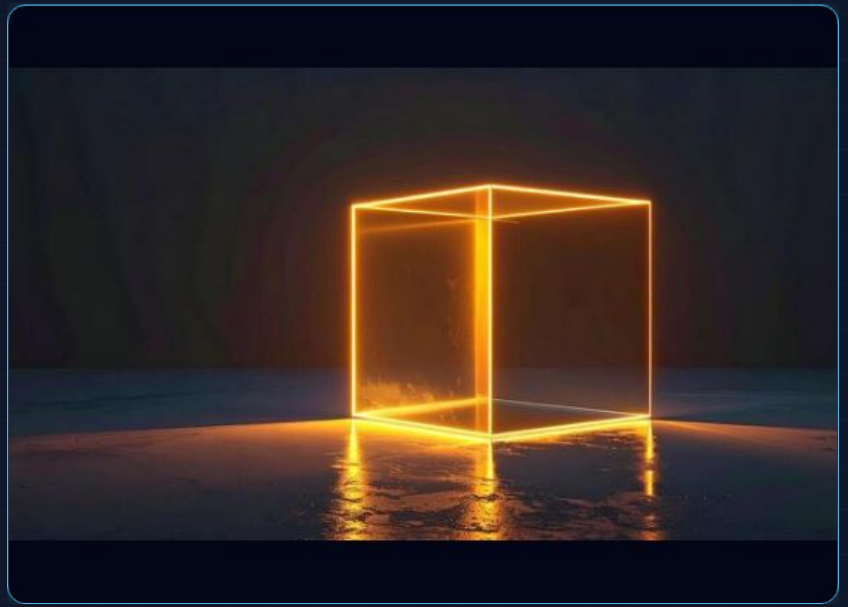
Understanding Variables

The fundamental building blocks of all computer programs. How we store, label, and manipulate data in memory.

The "Storage Box" Metaphor

The easiest way to understand a variable is to think of it as a **labeled storage box** in the computer's memory.

- **The Identifier (Name):** The label you write on the outside of the box so you can find it later (e.g., playerName).
- **The Value (Data):** The actual information you place inside the box (e.g., the text "Alex").
- **Mutability:** Just like a physical box, you can take the old data out and replace it with new data at any time while the program runs.



Anatomy of a Variable

1. Declaration & Assignment

To use a variable, you must first **declare** it (create the box) and then **assign** it a value (put data inside).
In modern languages, this is usually done in a single line of code.

> *The equals sign = is the **Assignment Operator**. It takes the data on the right and stores it in the variable on the left.*

2. The Code Syntax

```
// JavaScript Example
let userAge = 25;
let status = "Active";

// Python Example
score = 100
```

let	userAge	=	25
Keyword	Name	Operator	Value

Common Data Types (The Shape of the Box)

Strings

Text data, surrounded by quotation marks. Used for names, messages, and addresses.

```
"Hello World"
```

Numbers

Mathematical values used for calculations. Can be whole integers or decimals (floats).

```
42 | 3.14
```

Booleans

A simple logical switch that can only hold one of two possible states. Used for conditional logic.

```
true | false
```


ARRAY

Array

How can I easily handle information that is structured by nature? For example a complex number, number, or a vector or a matrix?

Typical structured variables are arrays

C Example: Vector

```
float v[10]; // Declaration

We can then access the i-th element of the vector:
v[i-1] = 3.5;
```

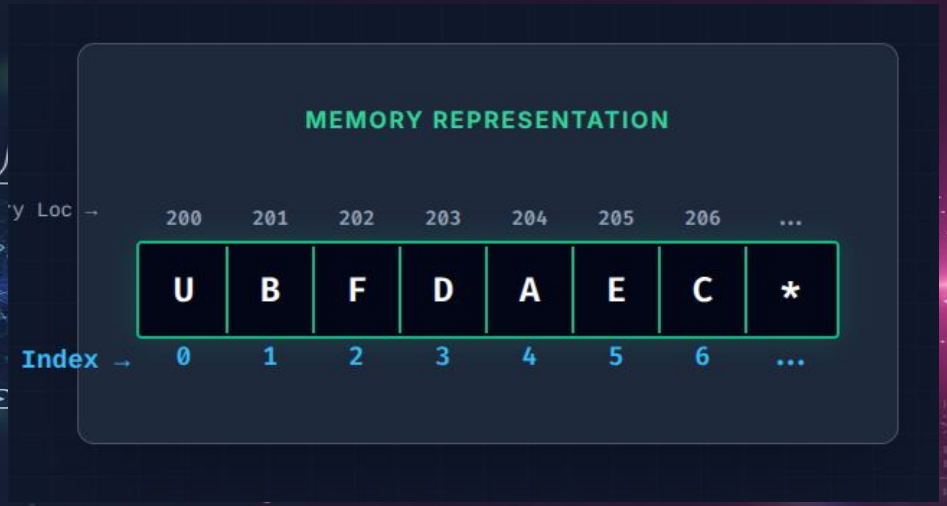
C Example: Matrix (2D Array)

```
Likewise I can define an array (two-dimensional array) as:
float m[10][10];
```

Array

If a normal variable is a single box, an array is a continuous row of boxes (a bookshelf) in the computer's memory.

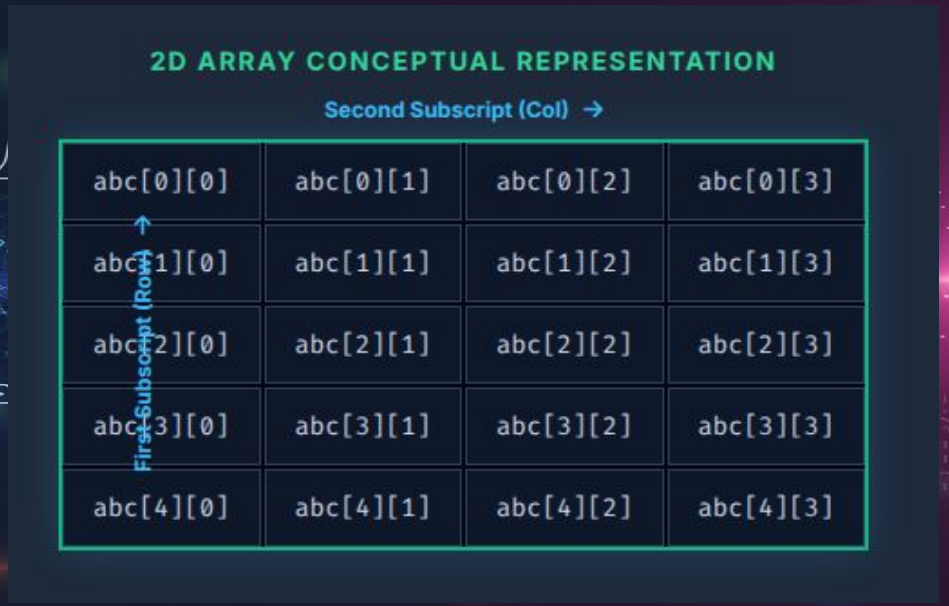
- Memory Location: The physical address in RAM (e.g., 200, 201). Arrays occupy contiguous space.
- Index (Subscript): The logical position we use to access the data.
- Zero-Indexed: In almost all languages, the first element is always at index `0`, not `1`.



Array

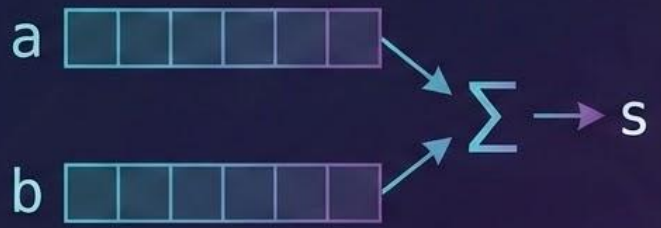
A 2D array is simply an "Array of Arrays". We conceptually view it as a grid or a matrix with rows and columns .

- "Here my array is abc[5][4], which can be conceptually viewed as a matrix of 5 rows and 4 columns. Point to note here is that subscript starts with zero, which means abc[0][0] would be the first element of the array."



Example

```
1  float a[N], b[N];
2  for (i=0; i<N; ++i)
3  {
4      a[i] = (float)rand()/(float)(RAND_MAX/N);
5      b[i] = (float)rand()/(float)(RAND_MAX/N);
6  }
7  s = 0.0;
8  for (i=0; i<N; ++i)
9  {
10     s = s + a[i]*b[i];
11 }
```



LANGUAGES

- Languages can also be classified according to their data type: **static typing** and **dynamic typing** .
- **Static typing:** the programmer is forced to explicitly specify the type of each syntactic element. For example, he must specify the type of a variable and the language will then guarantee that that variable will be used consistently with the declaration.
- **Dynamic typing** : for example, in this case the data will assume a type that varies at runtime based on assignments made (see Python)
- We can then also distinguish between **weak and strong typing**

LANGUAGES

1. WHEN are types checked?

Static Typing: Checked at Compile-Time.

Dynamic Typing: Checked at Run-Time.

2. HOW STRICTLY are they checked?

Strong Typing: Strict enforcement, no implicit conversions.

Weak Typing: Loose enforcement, language attempts to guess/coerce types.

LANGUAGES

Static Typing

The programmer explicitly declares the type of a variable. The compiler guarantees that variable will only ever hold that type of data. Errors are caught **before** the program runs.

Java / C++

```
int age = 25;
// The compiler locks 'age' as an integer

age = "Twenty Five";
^ ERROR: Incompatible types
```

Dynamic Typing

Variables do not have types; only the *values* do. A variable can hold a number right now, and a string on the next line. Types are evaluated **while** the program is running.

Python / JavaScript

```
age = 25
# 'age' points to an integer value

age = "Twenty Five"
✓ PERFECTLY VALID
```


Beyond the Mainstream

LANGUAGES

- **Parallel programming languages or paradigms** (for modern architectures) if you are curious you can see here for example http://www.storchi.org/lecturenotes/acr/index.html
- **Esoteric languages** : These are sometimes complex and unclear. Popular only among experienced users, they are used solely to test programming skills (essentially for entertainment purposes).
- **Scripting** : Initially developed for use in Unix shells, these languages are used to automate repetitive and time-consuming tasks.

LANGUAGES

Parallel Programming

Designed for **modern architectures** (Multi-core CPUs, GPUs, Supercomputers).

Instead of executing instructions one by one sequentially, parallel languages divide a large problem into smaller sub-problems that are solved **simultaneously**.

Use Cases:
Scientific simulations, Machine Learning training, Video rendering, Big Data processing.

Examples:
`CUDA (Nvidia GPUs), OpenMP, MPI, Erlang.`

LANGUAGES

Scripting Languages

backup_script.sh

```
#!/bin/bash
# Automating repetitive tasks
echo "Starting daily backup ..."
tar -czf backup_$(date +%F).tar.gz /var/www/html
scp backup_*.tar.gz admin@server:/backups/
echo "Done! Files secured."
```

Initially developed for Unix shells to **automate repetitive and time-consuming tasks**.

Today, they power the web and system administration. They are usually *interpreted* (run line-by-line) rather than compiled, making them fast to write and test.

System Admin
Bash, PowerShell

Web & General
Python, JavaScript, Ruby

LANGUAGES

🎭 Esoteric Languages (Esolangs)

Designed to be deliberately **complex and unclear**.

They are popular only among experienced programmers. They are never used for production software, but rather to test programming skills, push the boundaries of language design, or purely for **entertainment purposes**.

Famous Examples:

- **Brainfuck:** Uses only 8 simple characters (+ - < > . , [ ]).
- **Malbolge:** Designed to be the hardest language to write.
- **Shakespeare:** Code reads like a theatrical play.

BRAINFUCK: "HELLO WORLD" PROGRAM

Yes, that actually prints "Hello World".


Programming other concepts

From Source to Executable

- The source code is written in ASCII text files. The source code expresses the algorithm implemented in the chosen language. To write the source code, you can use simple text editors (VI, Emacs), or IDEs (integrated development environments with other tools, such as compilers, linkers, and debuggers).
- **Compilation** : The source code is translated (by the compiler) from a high-level language to executable code. The advantage is that execution is fast and the code is optimized for the specific platform. The disadvantage is that it will have to be recompiled for each different operating system or hardware.
- **Linking** : Each program generally uses one or more libraries, and the linker connects the libraries and the source program together. Linking can be either static or dynamic (for example, .so libraries in Linux/Unix-like programs or .dll libraries in Windows).

From Source to Executable

1. The Source Code

The source code is written in simple **ASCII text files**. It expresses the algorithm implemented in the chosen language.

The Tools

- **Text Editors:** Lightweight tools like VI, Nano, Emacs, or VS Code.
- **IDEs (Integrated Development Environments):** Heavy-duty platforms (like Visual Studio or Eclipse) that bundle editors with built-in *compilers, linkers, and debuggers.*

app.c - Emacs

```c
#include <stdio.h>

int main() {
   printf("Hello, World!\n");
   return 0;
}
```

From Source to Executable

2. Compilation

The compiler acts as a translator. It reads the high-level human-readable code and translates it entirely into low-level **machine code (binary)** before it can be run.

The Advantage

Execution is extremely fast because the code is optimized directly for the specific hardware/CPU.

The Disadvantage

Lack of portability. You must recompile the code separately for Windows, Linux, Mac, ARM, x86, etc.

From Source to Executable

3. Linking

Your program rarely stands alone. It needs pre-written code (like math functions or UI drawing tools) from **Libraries**. The Linker stitches your compiled Object Code together with these libraries to create the final executable.

Static Linking

The library code is physically copied and baked right into your final executable file. (Bigger file size, easier to distribute).

Dynamic Linking

The executable just contains a "pointer" to the library. The OS loads the library into memory when the program runs.

`Windows: .dll | Linux/Unix: .so`

IThe Alternative

- **Interpretation** : To avoid the problem of program portability, the concept of interpreters has been used. In this case, the source code is not compiled or "translated," but rather executed by an interpreter. This introduces other problems, such as performance.
- **Bytecode, P-code** : We can define it as an intermediate approach in which the source program is "translated" into an intermediate code that is interpreted by a virtual machine. This combines the advantages of good execution speed with extreme program portability. JIT (Just in Time) compiles the intermediate code into machine code at execution time.

Interpretation (The Alternative)

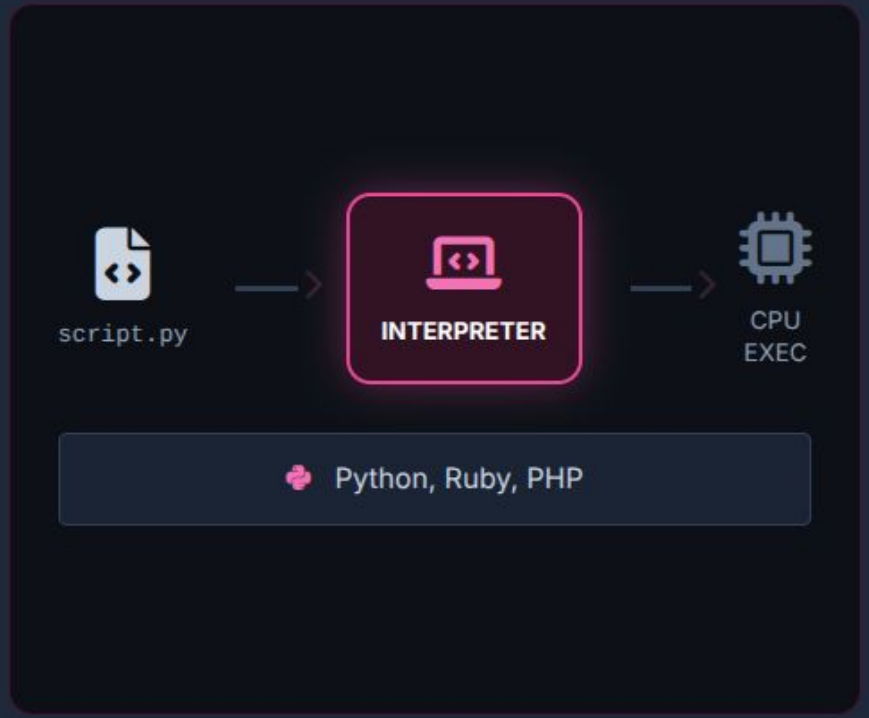
To solve the portability problem of compiled executables, the **Interpreter** model reads and executes the source code line-by-line at runtime, without creating a standalone executable file.

The Advantage

Extreme portability. The same source code file can run on Windows, Mac, or Linux, as long as the system has the Interpreter installed.

The Disadvantage

Lower performance. Translating high-level code to machine instructions on-the-fly during execution adds significant overhead.



The Hybrid Approach (Bytecode)

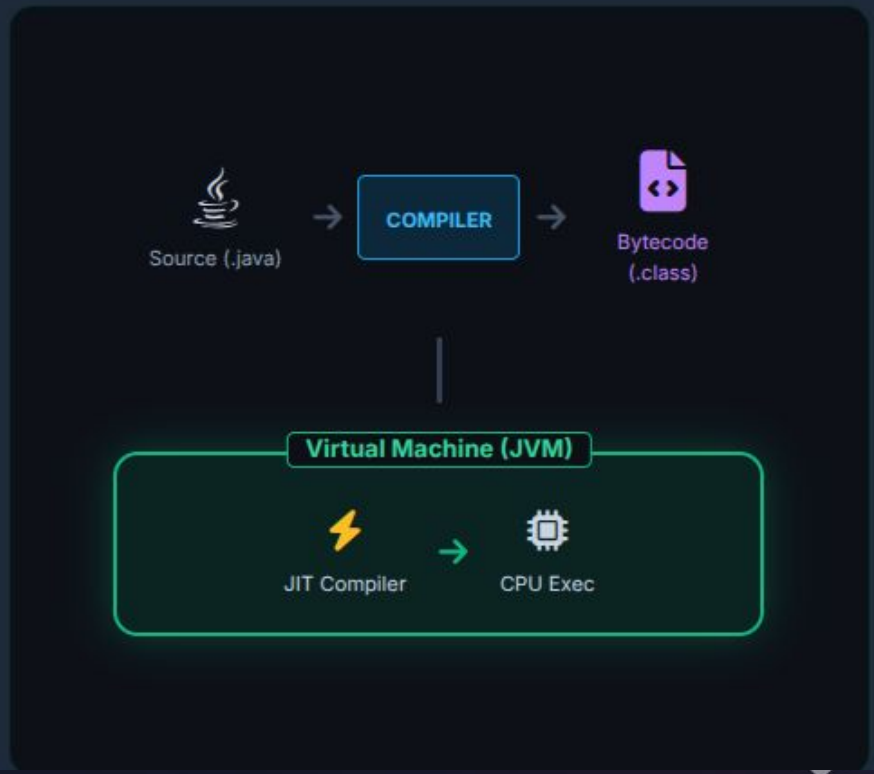
An intermediate approach combining the speed of compilation with the portability of interpretation.

1. Bytecode (P-Code)

The source code is first translated into a highly optimized, portable intermediate language called Bytecode. It is not machine-specific.

2. Virtual Machine & JIT

A Virtual Machine (VM) executes the bytecode. Modern VMs use **JIT (Just-In-Time) Compilation** to translate bytecode into native machine code exactly when it's needed during execution.




Computability

Programming other concepts

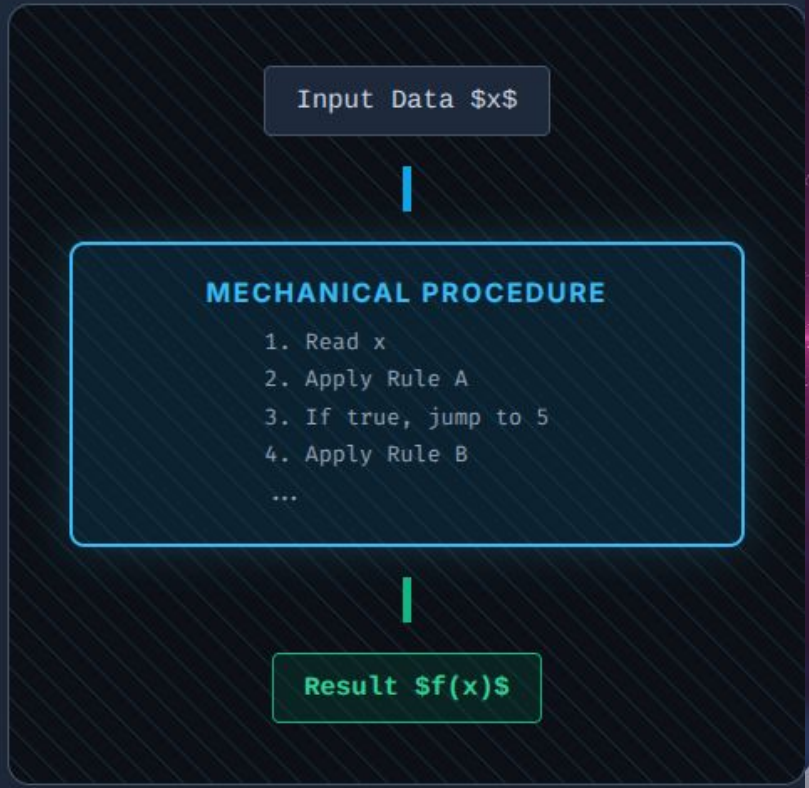
- **Computability:** Given a function, it is said to be computable if we can find an algorithm (i.e. a procedure that mechanically executes a finite number of steps) that computes it.
- Church-Turing thesis, the class of computable functions coincides with the class of functions computable by a Turing machine.
- All computing machines (computers) can be traced back to a Turing machine.

What is Computability?

Given a mathematical function or a problem, it is said to be **computable** if we can find an algorithm that calculates it.

What defines an Algorithm?

- **Finite Steps:** It must eventually terminate and produce a result.
- **Mechanical:** It must be a strict procedure that requires no human intuition or creativity to execute.
- **Unambiguous:** Every step must be exactly defined.



The Church-Turing Thesis

> *"The class of computable functions exactly coincides with the class of functions computable by a Turing machine."*

This is a **thesis**, not a mathematical theorem, because "intuitive computability" is a human concept. However, it is universally accepted.

It states that no matter how advanced, complex, or futuristic a computing system is, if a problem can be solved algorithmically, a simple Turing Machine can also solve it.

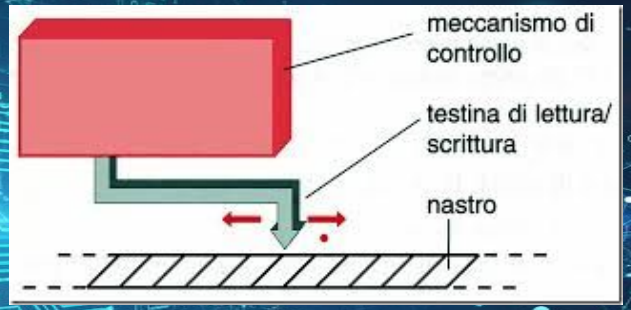
All computers trace back here

`Turing's Infinite Tape`	⇄ **Modern RAM / Storage**
`R/W Head + State Table`	⇄ **CPU (ALU & Control Unit)**
`Universal Turing Machine`	⇄ **Programmable Computer**

Every laptop, smartphone, and supercomputer is essentially a physical implementation of a Universal Turing Machine.

The Turing Machine

- **MdT: Turing machine**



Deterministic model with tape and 5-field instructions:

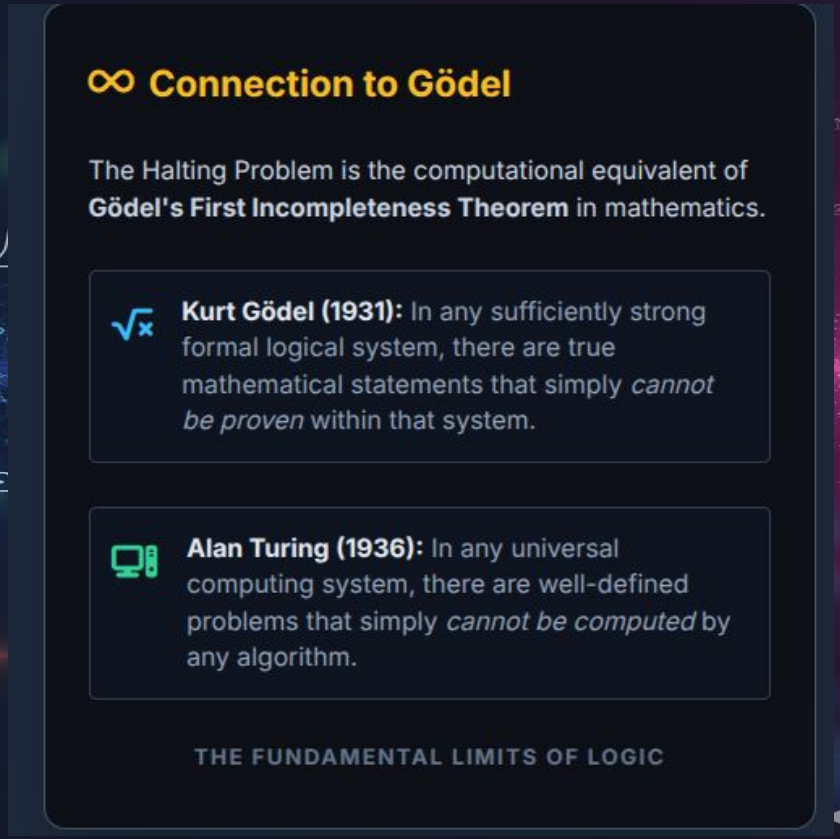
1 - A ribbon of any length that can contain characters or blank spaces
2 - reading and writing head/device with which to read and write on the tape and obviously the head can move the tape to the right or left
3 - The machine has an internal state

At each step, the MdT reads a symbol and, depending on its internal state, it can change state and then write a symbol on the tape and then move the tape to the right or left.
The behavior of the MdT is programmed by defining rules or quadruples of the type:

(internal state, read-symbol, new-state, written-symbol/direction)

Programming other concepts

MdT and the halting problem: given a certain program and a certain input, it is impossible to determine whether the program will halt or not. (This problem is closely related to Gödel's incompleteness theorem.)



THE FUNDAMENTAL THEORY

The Böhm-Jacopini Theorem

The foundations of structured programming and how three simple logical rules govern all modern computer science.

The 1966 Intuition

Proven in 1966 by Italian computer scientists **Corrado Böhm** and **Giuseppe Jacopini**, this theorem forever changed how we write software (moving from unstructured programming with GOTOs to structured programming).

> **The Statement:**
> *"Any computable algorithm can be written using exclusively a mix of three fundamental control structures."*

Whether it's calculating the physics of light in a next-generation video game or having a neural network process millions of data points, under the hood everything reduces to these three structures.

The Three Fundamental Structures

1. Sequence

The execution of instructions in chronological order, strictly one after the other.

```
x = 10;
y = x + 5;
print(y);
```

2. Selection

The branching of the logical flow based on a condition (the famous if and else).

```
if (x > 10) {
print("Greater");
} else {
print("Lesser");
}
```

3. Iteration

The repetition of a block of instructions as long as a specific condition is true.

```
while (n > 0) {
print(n);
n = n - 1;
}
```

The Reality of Modern Programming

Although at the hardware level there are always these three structures, modern languages offer tools to hide the chaos.

Spaghetti Code

If we tried to write the entire code of a video game (millions of lines) using *exclusively* long sequences of raw ifs and loops, we would create unreadable and unmaintainable code.

```
if (player.x > enemy.x) {
  if (player.y == enemy.y) {
    while (enemy.hp > 0) { ... }
  } else { ... }
} else if ( ... ) { ... }
// ... x 1,000,000 lines
```

The Illusion of Complexity

To manage the enormous complexity of modern software, computer scientists invented **abstract concepts**.

These concepts "rest" upon those three base structures to group and organize them into sensible logical blocks that humans can actually read and manage effectively.

Abstractions to Manage Complexity

Object-Oriented Programming (OOP)

Groups variables and functions into logical entities called "Objects" (e.g., the player in a video game) to keep the code rigidly organized.

Functions and Modules

Allow you to "package" a complex sequence of ifs and loops, give it a descriptive name, and reuse it infinite times without rewriting it.

Parallelism and Concurrency

Instead of having only one sequence at a time, thousands of sequences, choices, and loops are run simultaneously on different processors.

Return to the Origins

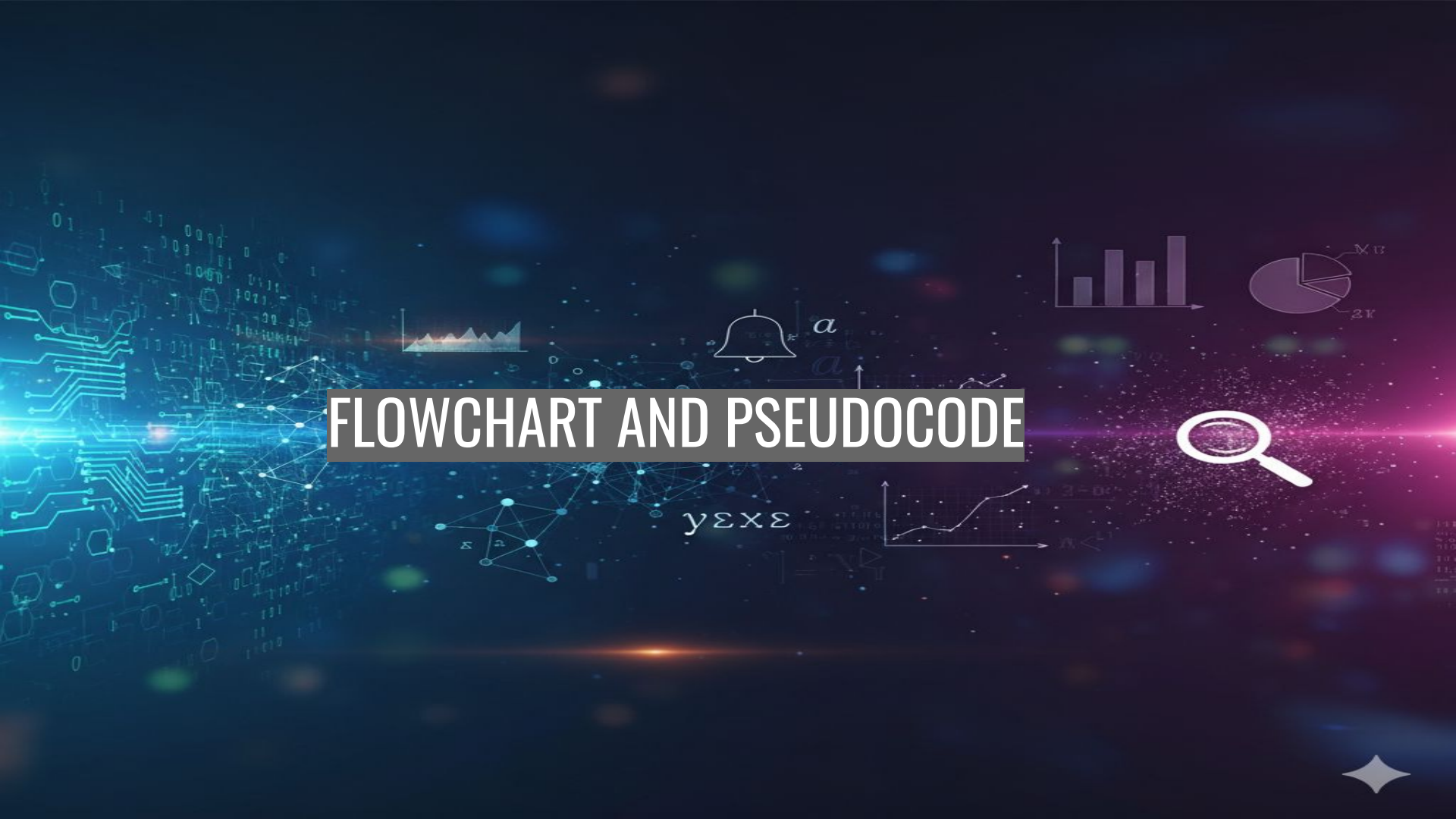
The truth under the hood.

In the end, when our elegant and abstract code passes through the **Compiler** to be translated into the machine language (Assembly/Binary) that the CPU must actually execute...

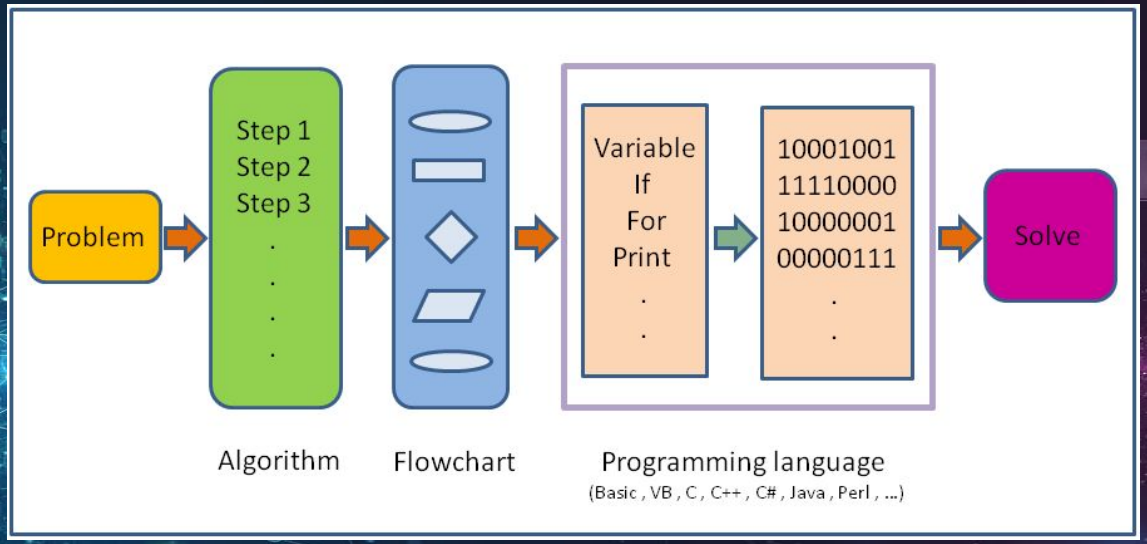
...all abstractions vanish.

The code returns to its purest form, exactly as described by the Böhm-Jacopini Theorem: a very long ribbon of sequential instructions, conditional jumps, and repeated cycles.

FLOWCHART AND PSEUDOCODE



FLOWCHART AND PSEUDOCODE



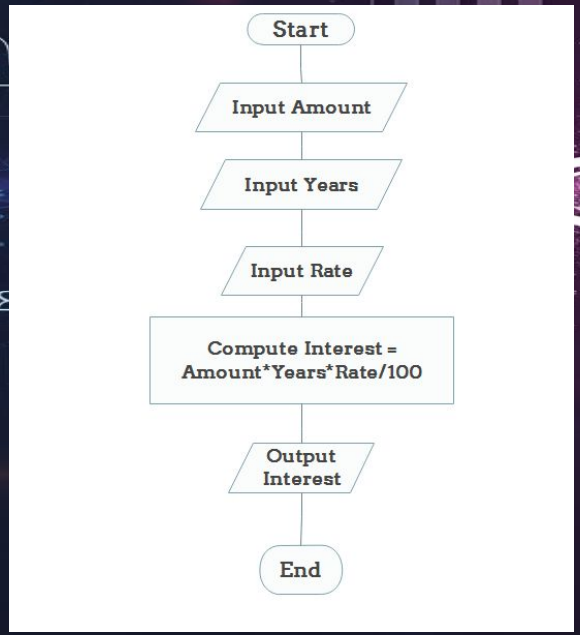
There are three fundamental structures that are used for the algorithmic resolution of problems: **selection, iterations and sequence** (sequence of instructions) (the GOTO present in machine languages since the 70s has been progressively discouraged / eliminated)

Examples: https://github.com/lstorchi/teaching

FLOWCHART AND PSEUDOCODE

- We will only go through some basic example: Calculate the Interest of a Bank Deposit

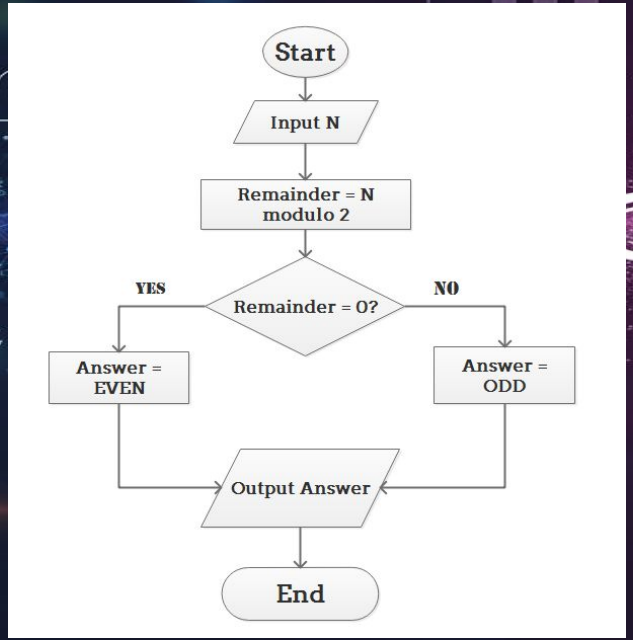
1. Read amount,
2. Read years,
3. Read rate,
4. Calculate the interest with formula "Interest=Amount*Years*Rate/100",
5. Print interest.



FLOWCHART AND PSEUDOCODE

- Determine and Output Whether Number N is Even or Odd

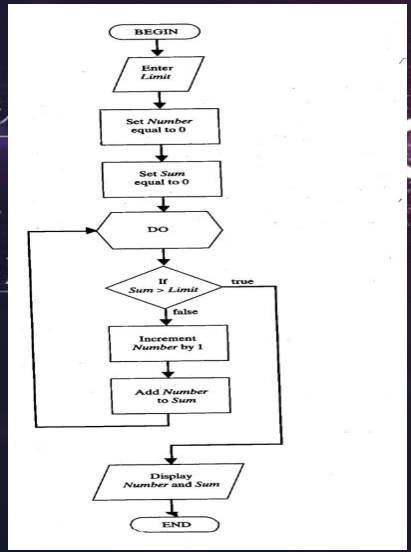
Step 1: Read number N,
Step 2: Set remainder as N modulo 2,
Step 3: If remainder is equal to 0 then number N is even, else number N is odd,
Step 4: Print output.

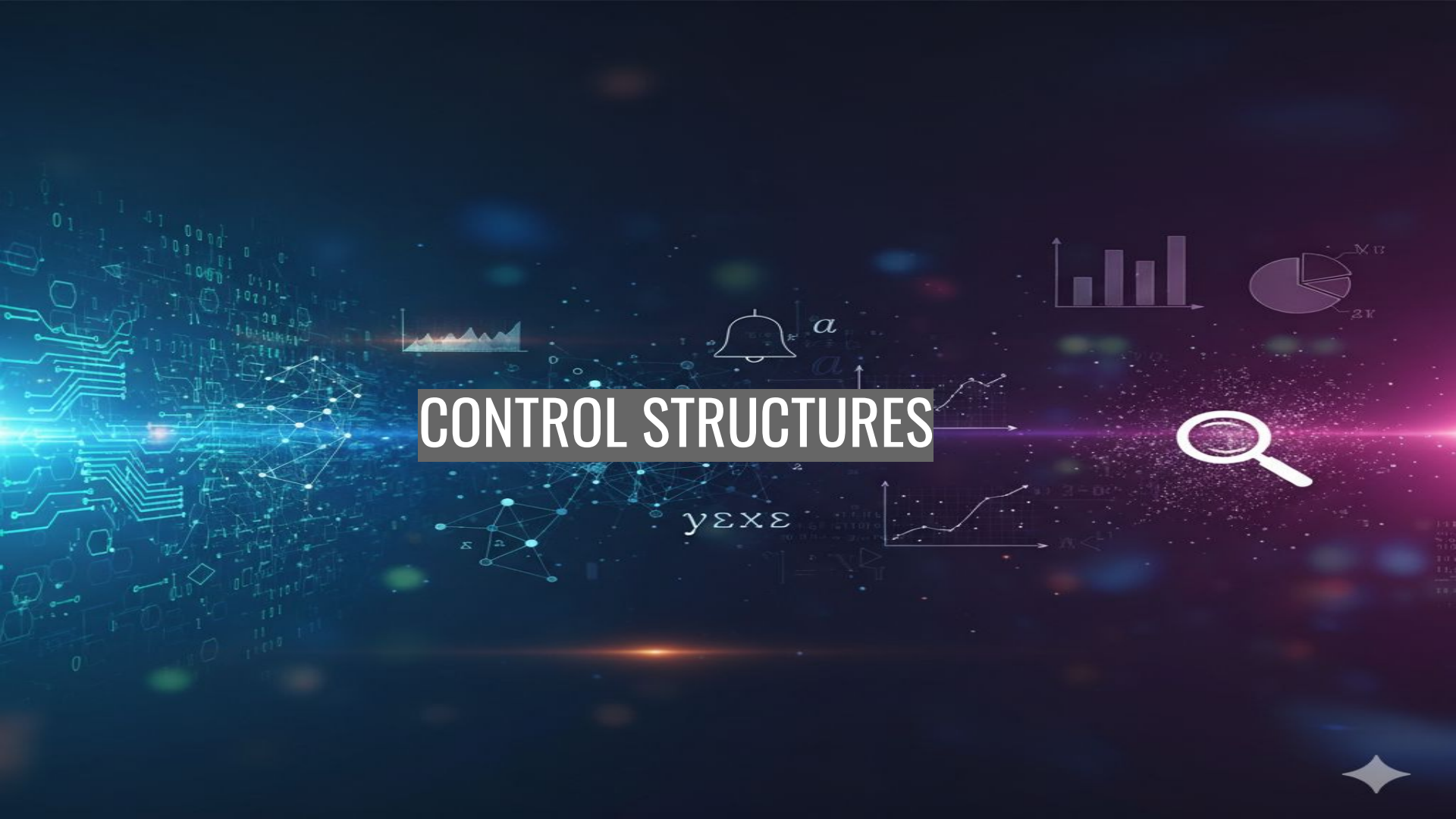


FLOWCHART AND PSEUDOCODE

- For a given value, Limit, what is the smallest positive integer **Number** for which the sum Sum = 1 + 2 + ... + Number **is greater than Limit. What is the value for this Sum?**

1. Enter **Limit**
2. Set **Number** = **0**.
3. Set **Sum** = **0**.
4. Repeat the following:
    a. If **Sum** > **Limit**, terminate the repitition, otherwise.
    b. Increment **Number** by one.
    c. Add **Number** to **Sum** and set equal to **Sum**.
5. Print **Number** and **Sum**.




CONTROL STRUCTURES


SEQUENCES

Sequences

Sequence control structure refers to the line-by-line execution by which statements are executed sequentially, in the same order in which they appear in the program. They might, for example, carry out a series of read or write operations, arithmetic operations, or assignments to variables.

```
Step 1: Read amount,
Step 2: Read years,
Step 3: Read rate,
Step 4: Calculate the interest with formula
            "Interest=Amount*Years*Rate/100
Step 5: Print interest,
```


SELECTION

Selection Control Structure

General logic for conditional execution:

```
if (condition 1)
   statements 1
else if (condition 2)
   statements 2
   ...
else
   statements N
endif
```

- Flexibility: **else if** can be repeated indefinitely.
- Minimalism: Structures can exist as a single `if` block.

Nesting Selection Logic

Complex decisions use nested instructions:

```
if (condition 1)
  statement 1
  if (condition 2)
    statements 2
  end if
else
  statements 3
end if
```

- Hierarchical Control: Inner blocks only execute if parent conditions are met.
- Logical Depth: Allows for granular filtering of data and program flow.

Operators

- In all programming languages, I can use relationship operators to compare numbers and variables, for example:

Description	Java, C, C++	Fortran
Greater than	>	.GT.
Greater than or equal	>=	.GE.
Less than	<	.LT.
Less than or equal	<=	.LE.
Equal	==	.EQ.
Not Equal	!=	.NE.

```
Int a;

a = 4 // operatori di  assegnazione

If  (a == 5)
{
   cout << "Hello" << std::endl;
}
else if (a > 5)
{
    cout <<  a << " > 5 " << std::endl;
}
```

Logical Operators

- I take for granted the logical operators AND, OR and NOT

Logical Operators		
Operator	**Description**	**Example**
&&	AND	x=6 y=3 x<10 && y>1 Return True
||	OR	x=6 y=3 x==5 || y==5 Return False
!	NOT	x=6 y=3 !(x==y) Return True

Examples the following expression

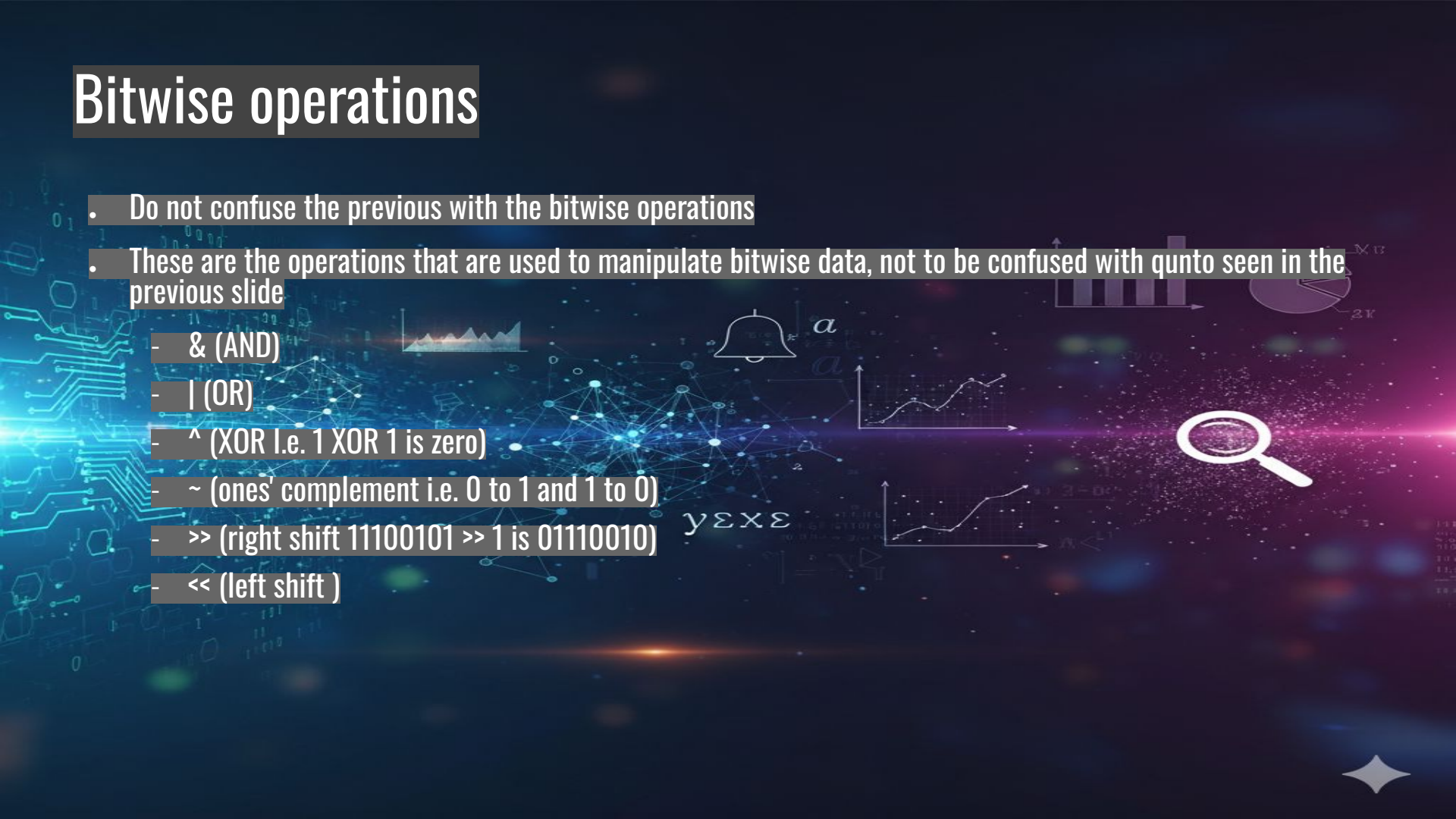
$5 < a < 7$

In programming languages it is broken into two elementary expressions linked by the operator AND:

$(a > 5)$ AND $(a < 7)$

Bitwise operations

- Do not confuse the previous with the bitwise operations
- These are the operations that are used to manipulate bitwise data, not to be confused with qunto seen in the previous slide
  - & (AND)
  - | (OR)
  - ^ (XOR I.e. 1 XOR 1 is zero)
  - ~ (ones' complement i.e. 0 to 1 and 1 to 0)
  - >> (right shift 11100101 >> 1 is 01110010)
  - << (left shift)



Bitwise operations

- A simple test just to clarify

```
└─ $ python3
Python 3.6.9 (default, A
[GCC 8.4.0] on linux
Type "help", "copyright"
>>> a = 60
>>> b = 13
>>> print(a&b)
12
>>> exit()
```

```
a = 60
b = 13

print(a&b)
```

```
12
```

```
a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100
```

Case structure

Basically a series of if-then-else with some constraint. In practice, the choice between the blocks of instruction is guided by the value of a certain variable:

```
switch variable: {
   val1 0: "!= min";
   case 1: "or inear";
   case 2: "== text";
   case 3: "== imast";
   break;
}
```

```
If (variable == val1) {
    // Code for val1
} else if (variable == 1) {
    // Code for case 1
} else if (variable == 2) {
    // Code for case 2
} else if (variable == 3) {
    // Code for case 3
}
```


LOOPS

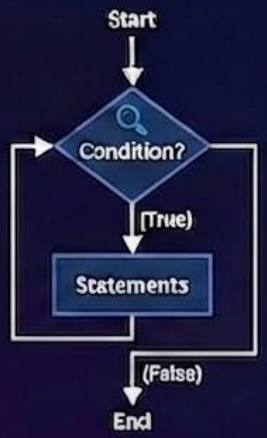
Loops

- There are three different types: **while..do** , **do..while** e **for**
- **Not all languages necessarily have all three of these structures**
- **These structures allow to repeat a block of instructions until a condition occurs**
- Also in this case it is possible to **nest** more loops one inside the other

While..do e Do..while

WHILE..DO LOOP

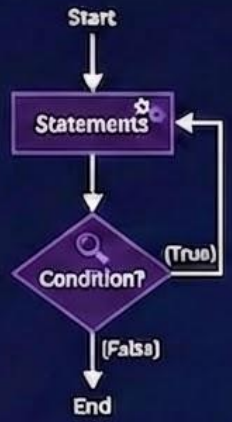
The block of instructions can also never be executed, since the condition is checked at the beginning, as long as the condition condition is true, the block of instructions is executed




```
WHILE (condition)
    statements
END DO
```

DO..WHILE LOOP

do..while instead executes the block of instructions until the condition is false

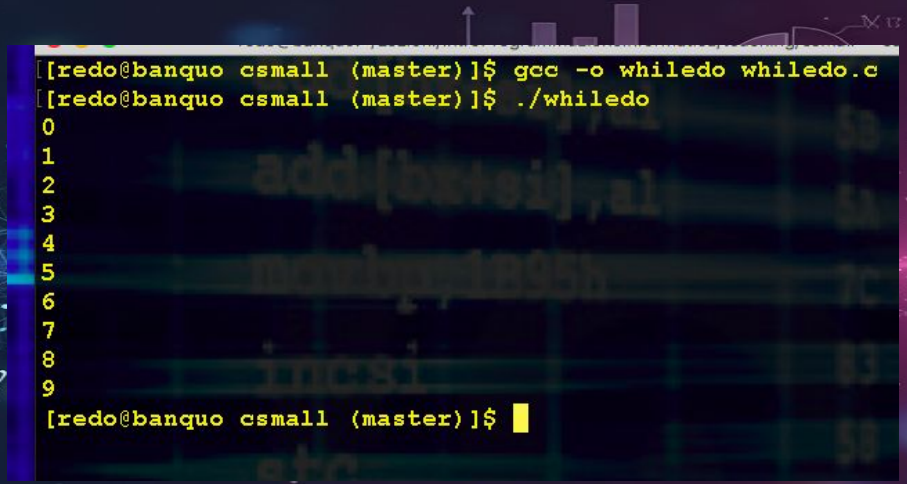



```
DO
    statements
WHILE (condition)
```

Example

A simple C example:

```
int i = 0, N = 10;
while (i != N)
{
    printf("%d \n", i);
    i++;
}
```



Example

```
Type "help", "copyright"
>>> while True:
...     print("here")
...
```

```
while True:
  print("here")
```

```
here
here
here
here
here
here
here
here
here
here
```

Example

- Int i = 0, N = 10;

```
do
{
  printf ("%d \n", i);
  i++; // i = i + 1
} while (i >= N);
```

For loop

- Executes a block of instructions a known number of times.
- Often uses an integer counter variable, changed step-by-step.
- End condition determined by comparing counter with a value.

```
for (counter = startingValue TO endValue STEP = stepValue) {
    statements
} // end for
```

Example

```
int i;
for (i=0; i<10; ++i)
{
    printf (“%d \n”, i);
}
```

```
[redo@banquo csmall (master)]$ gcc -o forloop forloop.c
[redo@banquo csmall (master)]$ ./forloop
0
1
2
3
4
5
6
7
8
9
[redo@banquo csmall (master)]$
```

Example

```
for i in range(10):
    print(i)
```

```
└─ $ python loop.py
0
1
2
3
4
5
6
7
8
9
```

Algorithmic complexity

Algorithmic complexity

- **Algorithmic complexity:** is the measure of the difficulty of a calculation (algorithm + input)
- The quality of an algorithm is evaluated based on the time and space required for its execution, and therefore generally based on the resources required.
- Clearly, execution time depends on the type of input as well as the type of hardware used, so it makes no sense to classify algorithms based on the number of seconds required for their execution.
- The computation time is therefore expressed as the number of elementary operations as a function of the size N of the input data.

Algorithmic complexity

- Example of calculating the efficiency of an algorithm in which we search for the minimum **m** within a set of **N** numbers {x1, x2, ..., xN}
- Let's imagine we tackle the problem as follows:
  - I choose x1 as the minimum possible
  - I compare it with x2, then x3 and so on
  - If I find a smaller xi I continue the comparisons with that one as I did with x1
  - At the end I will have found the minimum
- To do everything I will have done N comparisons so the efficiency of the algorithm is directly proportional to the size N of the input

Algorithmic complexity

- How many comparisons do we make to find the minimum?
- Step 1: Initialization
  - We set the first element as the minimum (1 operation).
- Step 2: Iteration
  - We loop through the remaining elements. Since there are N elements total, we check N - 1 elements.
- Therefore, the number of comparisons is exactly N - 1.

Algorithmic complexity

find_minimum.py

```python
def find_min(numbers):
    # 1. Assume the first is the smallest
    current_min = numbers[0]

    # 2. Iterate through the rest
    for i in range(1, len(numbers)):
        # 3. Compare and update
        if numbers[i] < current_min:
            current_min = numbers[i]

    # 4. Return the result
    return current_min
```

Algorithmic complexity

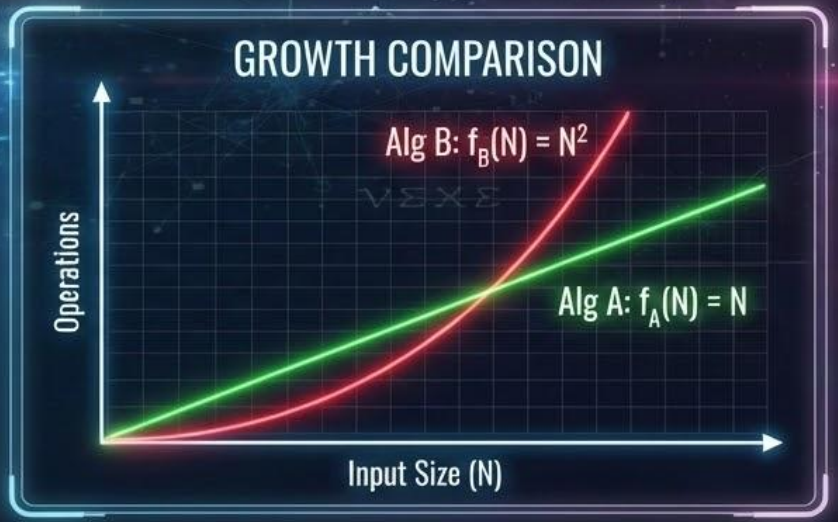
Algorithmic complexity

- **The efficiency of a given algorithm can therefore be expressed as a function f(N), therefore a function of the variable N which represents the size of the input data.**
  - This function therefore expresses the number of elementary operations necessary to solve the problem using the given algorithm as a function of the size of the input.
  - It therefore represents the computational complexity
  - **Given N, an algorithm A is more efficient than another B if as N increases $f_A(N)$ is less than or equal to $f_B(N)$**
- The execution time of a program therefore depends on the complexity of the algorithm, on the size of N and obviously on the "speed" of the machine on which it is executed.

Algorithmic complexity

Given input N, Algorithm A is more efficient than Algorithm B if, as N increases:

$$f_A(N) \leq f_B(N)$$



Algorithmic complexity

- To divide algorithms into complexity classes, the following criterion is used:
- **A function f(N) is said to be of order g(N) and is denoted by f(N) = O(g(N)) if there exists a constant K such that , unless for a finite number of values of N the following inequality is always true: f(N) ≤ K*g(N) . One can also write f(n) / g(N) ≤ K**
- For example 2*N + 5 = O(N) in fact 2*N + 5 ≤ 7*N for every N greater than zero

Algorithmic complexity

Prove that: $2N + 5 = O(N)$

We need to find a constant K where $2N + 5 \leq K * N$.

Let's assume $N \geq 1$.

If $N \geq 1$, then $5 \leq 5N$.

Therefore, $2N + 5 \leq 2N + 5N$.

$2N + 5 \leq 7N$.

The inequality is true for $K = 7$ for every $N > 0$.


P vs NP

Class P (Polynomial Time)

The Definition

- P stands for Polynomial time. It contains all decision problems that can be solved by a Deterministic Turing Machine using a polynomial amount of computation time.

Easy" Problems

- If a problem is in P, it is considered "tractable" or relatively fast to solve, even for large inputs. The execution time **f(n) grows as $O(n^k)$**.
  - Sorting a list of numbers.
  - Finding the shortest path on a map.
  - Multiplication of large numbers.

Class NP (Nondeterministic Polynomial)

The Definition

- NP stands for Nondeterministic Polynomial time. It contains problems where, if you are given a potential solution, you can verify if it is correct in polynomial time. (Equivalently, solvable by a Nondeterministic Turing machine in polynomial time).

"Hard to Solve, Easy to Check"

- Think of a Sudoku puzzle. Solving it from scratch is hard. But if someone hands you a completed grid, checking if all rows/columns add up to 1-9 is very easy and fast!
  - Sudoku / Crosswords.
  - Traveling Salesman Problem.
  - Cracking cryptographic passwords.

NP-Complete Problems

The "Hardest" in NP

- NP-Complete problems are the most difficult problems in the NP class. They share a unique, magical property: they are all mathematically interconnected.

You can "convert" (reduce) any NP problem into an NP-Complete problem in polynomial time. Therefore:

- If one is easy to solve, THEY ARE ALL EASY.
- If one is difficult to solve, THEY ARE ALL DIFFICULT.

Example: Prime Factorization

The Problem (NP problem but quite surely not NP-Complete)

- **Given an integer N, find the prime numbers a and b such that: N = a x b**

Why is it in NP?

- **Verification is instant: If someone claims a=13 and b=17 for N=221, you just multiply them. Fast!**
- **Solving is slow: For a 500-digit number N, no known fast (polynomial) algorithm exists to find a and b from scratch.**

The Million Dollar Question

$$P = NP\ ?$$

The Clay Mathematics Institute offers $1,000,000 to anyone who can prove whether P = NP or P ≠ NP.

If it is easy to check that a solution to a problem is correct, is it also inherently easy to find that solution from scratch?

The Million Dollar Question

$$P \neq NP(Expected) \quad P = NP(Miracle)$$

If P = NP: Modern cryptography breaks instantly. AI can easily solve perfect logistics, find cancer cures, and optimize everything globally. We just haven't found the algorithm yet.

Algorithmic complexity

- Complexity classes P and NP, whether P equals NP or not is still an open question today (Clay Math Institute million dollar problem)
- Class P problems solvable in a deterministic Turing machine in polynomial time.
- Class NP problems for which no algorithm with polynomial complexity is known. However, they can be verified quickly.
- NP-complete problems: the simplest way to describe it is that if one of the problems is easy, then they all are, since I can "convert" the solution of one into the other. Likewise, if one is difficult, they are all difficult.
- EXAMPLE: Factoring an integer into prime numbers is a NP problem (most likely it is not NP-complete, we cannot say yet) given c find the prime factors a and b such that a * b = n