<u>Object Oriented Programming</u>

Programming: Programming means giving instructions to a computer to perform any task.

Programming language Types:

1. Machine Language

- This is the lowest-level programming language.
- It consists of binary code—just 0s and 1s—that the computer’s processor can execute directly.
- Machine code is unique to each CPU type and architecture, and programming in it is extremely difficult for humans since it is hard to read and prone to errors

2. Assembly Language

- Assembly language sits one step above machine language.
- Instead of binary, it uses mnemonics (short text codes like MOV, ADD, etc.) to represent instructions, which are easier for humans to read.
- Each assembly instruction corresponds closely to a single machine code instruction.
- Assembly programs must be translated into machine code by an assembler before they can run. Assembly language is specific to a particular computer architecture.

A basic program to sum two values in assembly language:

```
LDA 2050H    Load the first number from memory location 2050H into the Accumulator (A)
MOV B, A     Move the content of Accumulator (A) to Register B
LDA 2051H    Load the second number from memory location 2051H into the Accumulator (A)
ADD B        Add the content of Register B to the Accumulator (A). Result is stored in A.
STA 2052H    Store the content of the Accumulator (A) into memory location 2052H (the sum)
HLT          Halt the program execution
```

3. High-Level Language

- High-level languages are designed to be easy for humans to read, write, and understand.
- They use English-like syntax (e.g., print, if, while) and abstract away hardware details.
- Common examples: Python, Java, C++, Ruby, JavaScript.
- Programs written in these languages need to be compiled or interpreted into lower-level code (assembly/machine language) before a computer can execute them. High-level languages are usually portable across different types of computers

Procedural Programming

- Procedural Programming is a programming paradigm based on the concept of procedures, also known as routines, subroutines, or functions.
- Procedures/Functions are blocks of code that perform specific tasks, and the program executes in a step-by-step, top-down manner.
- A program in Procedural Language is a list of functions.

Core Principles

- **Modularity**
  The program is divided into smaller units called procedures or functions, each responsible for a specific task. This improves code readability and reusability.

- **Top-Down Design**
  The development starts from the overall task and breaks it down into sub-tasks, each handled by a separate function.

- **Control Flow**
  Procedural programs use three basic control structures:
  - **Sequence** (executing instructions one after another)
  - **Selection** (decision-making using if, else, switch)
  - **Iteration** (repeating tasks using for, while, do-while loops)

- **Variable Scope and State**
  Variables can be local or global. Data is passed between functions via parameters and return values.

- **Function Calls**
  Functions are called from the main program or other functions to perform specific actions, allowing code reuse.

Advantages of Procedural Programming

- Simple and easy to learn.
- Clear structure makes it easy to debug and maintain.
- Code reuse through functions.
- Good for small to medium-sized programs.

Limitations/Flaws of Procedural Programming

- **Global variable access:**
  Since functions have unrestricted access to global variables, a new or careless programmer may unintentionally modify or corrupt data, leading to hard-to-find bugs.
- **Lack of data protection:**
  There is no concept of data hiding — any function can access or modify any data, which reduces security and modularity.
- **Poor traceability in large programs:**
  In large codebases, it becomes difficult to track which functions use or modify which data, making debugging and maintenance more challenging.
- **Tight coupling of functions and data:**
  When new data elements are added, multiple functions may need to be modified to handle the new data, making the system less flexible and more error-prone.
- **Inadequate real-world modeling:**
  Procedural programming focuses on actions (functions), not objects or entities. As a result, it is not suitable for modeling real-world problems where data and behavior should be grouped together (e.g., in simulations or business applications).

Object Oriented Programming

- Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects rather than functions and logic.
- Objects are instances of classes, which are user-defined data types that bundle data (attributes) and functions (methods) that operate on that data.

- A Problem is divided into a number of objects. Which means a problem is solved by identifying and modeling real-world entities as objects, which interact with each other to perform tasks.
- It overcome the flaws of procedural programming language.
- Each object encapsulates its data and the functions that operate on that data.
- **objects communicate** with each other by **sending messages**; usually in the form of **method calls**.
- OOP follows a bottom-up design approach, where individual objects (representing real-world entities) are created first and then integrated to build the complete system.

Real-World Example: "Car" Object

Attributes (Data Members):

- color → "Red"
- model → "Honda City"
- speed → 80 km/h

Methods (Functions):

- startEngine()
- accelerate()
- applyBrakes()

Another Example: "Student" Object

🔹 Attributes:

- name → "Rahul"
- rollNumber → 101
- branch → "CSE"

🔹 Methods:

- attendClass()
- submitAssignment()
- writeExam()

Object

- Objects are run-time entities. An object is the instance of class.
- Combines attributes and operations.

Class

- A class is a blueprint, template, or design from which objects are created.
- It defines:
  - ✓ Data members (also called attributes or fields)
  - ✓ Member functions (also called methods or behaviors)
- A class itself doesn’t occupy memory — objects created from the class do.

Real-World Analogy:

Class = Blueprint, Object = Actual House

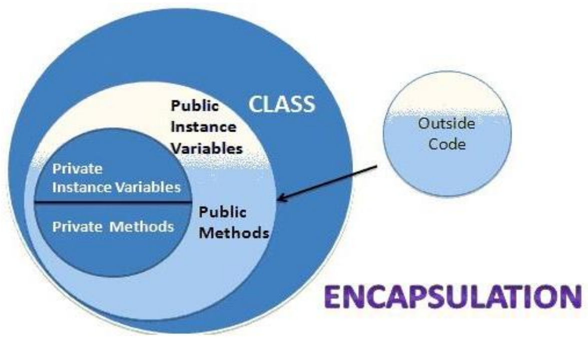
- A **blueprint** of a house defines the structure (rooms, doors, windows), but it’s not a real house**.**
- When you use the blueprint to build a house, that’s like creating an **object** from a **class**.

```
class Car {
  // Attributes (data)
  String color;
  int speed;
  // Method (behavior)
  void accelerate() {
    speed += 10;
    System.out.println("Speed is now: " + speed);
  }}}
```

OOP Principles

1. Encapsulation

- Encapsulation is the concept of binding data and methods together inside a class and restricting direct access to some of the object’s components.
- Data is hidden using access specifiers (like private, public, protected).
- Prevents accidental data modification and increases data security.



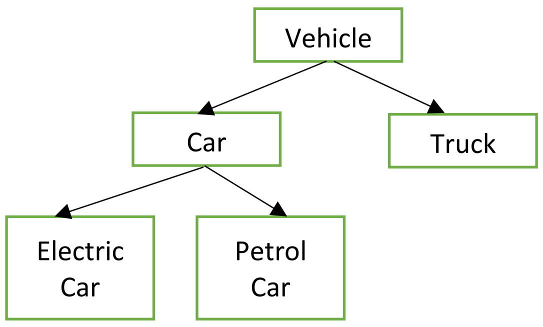
2. Abstraction

- Abstraction is the process of **hiding complex implementation details** and showing **only the essential features** of the object.
- Like with a **car**, you don’t need to know how the engine works to drive — you just use the steering, pedals, etc.
- In programming, **objects** work the same way; they expose only what you need (methods) and hide internal code.
- Big systems are managed using **hierarchical abstraction** — breaking them into smaller parts (like a car → music system → CD player).
- Achieved using **abstract classes**, **interfaces**, or simply method hiding.
- Reduces complexity for users of the class.

3. Inheritance

- Inheritance allows a **new class** (called *subclass* or *derived class*) to **acquire properties and behaviors** (data and methods) of an **existing class** (called *superclass* or *base class*).

- Promotes **reuse of existing code**, avoiding duplication.



4. Polymorphism

- "Polymorphism" comes from Greek: **"poly" = many**, **"morph" = forms**.
- It means the **same function name or symbol behaves differently** in different situations

Types of Polymorphism

1.**Compile-Time Polymorphism (Static Binding)**

- Achieved through method overloading or operator overloading.
- The decision is made at compile time.

Example – Method Overloading:

2. **Run-Time Polymorphism (Dynamic Binding)**

- Achieved through method overriding.
- The decision is made at runtime, based on the object.

Example – Method Overriding:

5. Message Passing

Message passing is the process by which objects communicate with each other by sending and receiving information (messages), typically by calling methods.

- In OOP, objects don't access each other's data directly.
- Instead, one object **sends a message** to another, usually by calling one of its **public methods**.
- This promotes **encapsulation** because the internal data is hidden.

Example

Main class	Car class	Driver class

public class Main {
 	
    public static void main(String[] args) {
 	
        Car Nexon = new Car();
 	
        Driver john = new Driver();
 	
        john.drive(Nexon); // Message passing from Driver to Car
 	
    }
 	
}
	public class Car {
	    void startEngine() {
	        System.out.println("Car engine started.");
	    }
	
	    void stopEngine() {
	        System.out.println("Car engine stopped.");
	    }
	
	    void accelerate() {
	        System.out.println("Car is accelerating.");
	    }
	}
		public class Driver {
		    void drive(Car car) {
		        car.startEngine(); // Message sent to Car
		        car.accelerate(); // Another message
		        car.stopEngine(); // One more message
		    }
		}