OOPs (Object Oriented Programming)

Java is not a pure Object-Oriented Programming (OOP) language, but it's **mostly** object oriented. Because it supports primitive data types. However, Java provides **wrapper classes** (Integer, Character, Double, etc.) to use primitives as objects when needed.

What is OOP's ?

OOP's is a way of writing programs by creating objects that combine data and actions, making the code organized, reusable, and similar to real-world things.

- Data = variables (what the object has)
- Actions = methods (what the object does)

A **class** is like a **blueprint** or **template** that defines the **properties** (attributes) and **behaviors** (methods) that objects of that class will have.

An **object** is an **instance** of a class. It is a **specific thing** created based on the class template, containing real values for the properties and can perform the behaviors defined by the class.

Class: A **Phone** has attributes like brand and model, and actions like makeCall(). **Object:** A **Samsung Galaxy S21** is an instance of the Phone class that can make a call.

```
// Class definition
class Phone {
    String brand; // Phone brand
    String model; // Phone model
    // Constructor to initialize brand and model
    Phone(String b, String m) {
    brand = b; model = m;
    }
    // Method to make a call
    void makeCall() {
    System.out.println("Making a call on " + brand + " " + model);

    }}

    public class Main {

    public static void main(String[] args) {
    // Create a Phone object
    Phone myPhone = new Phone("Samsung", "Galaxy S21");
    // Call the makeCall method
    myPhone.makeCall(); // Output: Making a call on Samsung Galaxy
S21
    }}
```

Constructor : A constructor is a special type of method in Java that is used to create and initialize objects. It is automatically called when an object is created and is used to set initial values for the object's attributes.

- A constructor has the **same name** as the class it belongs to.
- It does **not have a return type**, not even `void`.
- It allows objects to be **initialized** with specific values when they are created.

The **constructor** Phone(String b, String m) has the **same name as the class** Phone. This is why it works.

When you create the object Phone myPhone = new Phone("Samsung", "Galaxy S21");, the **constructor**
is called automatically to initialize the brand and model.

2 Types of Constructor :

- **Default Constructor**: Automatically created by Java if we don't provide a constructor. It sets the fields to default values like `null`, `0`, etc.

  Car myCar = new Car(); // Uses default constructor

- **Parameterized Constructor**: You define this constructor to **accept values** when creating an object. This allows you to **initialize** the object with specific values.

- **Constructor**: Initializes an object with initial values when created. It doesn't return anything and is automatically called.
- **Method**: Defines behaviors or actions that the object can perform, and is explicitly called after the object is created.

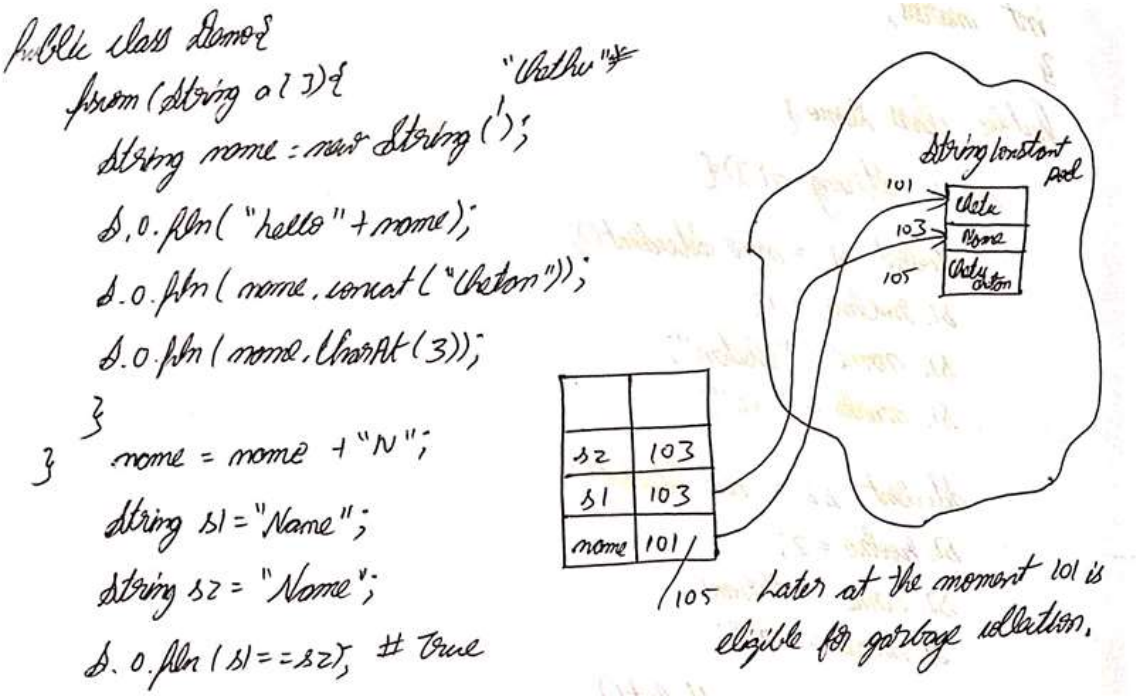
Feature	Constructor	Method
Purpose	Initializes the object when it's created.	Defines the actions/behaviors of the object.
Return Type	No return type.	Must have a return type (e.g., `void`, `int`).
Name	Must be the same as the class name.	Can have any valid method name.
Called	Automatically when the object is created.	Explicitly called using the object.
Parameters	Can have parameters (in parameterized constructors).	Can have parameters (optional).

Types of Memory in Java

1. Stack Memory
Used for storing method calls and local variables. Memory is cleared automatically after the method ends.The stack follows the **Last-In, First-Out (LIFO)** order, meaning the last method to be called is the first to finish. Stack contains reference of the object.

2. Heap Memory
Stores all objects and instance variables created using `new`. This memory is shared by all threads. Java's Garbage Collector manages and clears unused objects.

Garbage Collection (GC) in Java is a perform automatic memory management removes unused or unreferenced objects from heap memory to free up space and improve performance.



```
class Student{
    int rollno;
    String name;
    int marks;
}
public class StudentArray{
    public static void main(String args[]){
        Student s1=new Student();
        s1.rollno=1;
        s1.name="Chetan";
        s1.marks=10;

        Student s2=new Student();
        s2.rollno=2;
        s2.name="Syed";
        s2.marks=12;

        Student s3=new Student();
        s3.rollno=3;
        s3.name="Raghu";
        s3.marks=9;

        Student students[]=new Student[3];
        students[0]=s1;
        students[1]=s2;
        students[2]=s3;
        System.out.println("Name" + " " + "Rollno" + " " + "Marks");
        System.out.println();

        for(int i=0;i<students.length;i++){
            System.out.println(students[i].name+" :
"+students[i].rollno+" : "+students[i].marks);
       }}
```

Encapsulation : refers to a integrating variables and methods into single unit. Class's variables are hidden from other classes and can only be accessed or modified through getter and setter methods. **Example**: **Bank Account.**

Why Encapsulation?

- To **protect data** from unauthorized access.
- To **control how data is modified**.
- To **hide the internal implementation** and expose only what is necessary (data hiding).

How It's Achieved in Java :

1. Declare variables as `private`.
2. Provide **public getter and setter** methods to access and update them.

```
class Encapsulation{
    private String name;
    private int age;

    public String getname(){
        return name;        }

    public void setname(String n){
        name = n;                   }
    public int getage(){
        return age;     }

    public void setage(int a){
        age = a;                }}
public class EncapsulationCode{
    public static void main(String args[]){
        Encapsulation obj = new Encapsulation();
        obj.setname("Chetan");
        obj.setage(25);

        System.out.println(obj.getname()+":"+obj.getage());
}}
```

- `name` and `age` are **private** variables — cannot be accessed directly.
- **Getters** (`getname()`, `getage()`) — used to **read** the values.
- **Setters** (`setname()`, `setage()`) — used to **set/update** the values.
- In `main()`, we:

1. Create object of `Encapsulation`.
2. Set values using setters.
3. Print values using getters.

```
class BankAccount{
    private String accountHolderName;
    private int accountNumber;
    private double accountBalance;

    public String getaccountHolderName(){
        return accountHolderName;           }
    public int getaccountNumber(){
        return accountNumber;       }
    public double getaccountBalance(){
        return accountBalance;          }
    public void setaccountHolderName(String a){
        accountHolderName = a;                  }
    public void setaccountNumber(int a){
        accountNumber = a;              }
    public void setaccountBalance(double a){
        if (a>=0){
        accountBalance = a;
        }else{
            System.out.println("Account Balance cannot be
NEGATIVE");
        }}
    public void deposit(double amount){
        accountBalance += amount;
        System.out.println("Deposited: "+amount);
        System.out.println("Updated Balance: "+accountBalance);
    }
    public void withdraw(double amount){
        accountBalance -= amount;
        System.out.println("Withdraw: "+amount);
        System.out.println("Updated Balance: "+accountBalance);
    }
}
public class EncapsulationExample{
    public static void main(String args[]){
        BankAccount ba = new BankAccount();
        ba.setaccountHolderName("Chetan N Revankar");
        ba.setaccountNumber(123456789);
        ba.setaccountBalance(0.0123);
        System.out.println(ba.getaccountHolderName()+" :
"+ba.getaccountNumber()+" : "+ba.getaccountBalance());
        System.out.println();
        ba.deposit(500);
        System.out.println();
        ba.withdraw(500);           }}
```

Inheritance : means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and field of that class. It promotes **code reuse.** creating a relationship between parent and child classes. **Example**: **Vehicle → Car → ElectricCar.**

1. **Single Inheritance,** a **child class** can **inherit** properties and methods from **one parent class**.

```
class Animal{
    void eat(){
        System.out.print("This Animal eats
Food...!");          }}
class Dog extends Animal{
    void bark(){
        System.out.println("The Dog is
Barking...!");           }}

public class Inheritance
{
    public static void main(String[] args) {
        Dog animal = new Dog();
        animal.eat();
        System.out.println();
        animal.bark();
     }}
```

2. **Multilevel Inheritance,** a class is derived from another class, which itself is derived from another class.

```
class Animal{
    void eat(){
        System.out.print("This Animal eats
Food...!");      }}
class Dog extends Animal{
    void bark(){
        System.out.println("The Dog is Barking...!");       }}
class Huskey extends Dog{
    void looks(){
        System.out.println("The Huskey looks like a
FOX...!");  }}

public class MultiLvlInheritance
{
    public static void main(String[] args) {
        Huskey animal = new Huskey();
        animal.eat();
        System.out.println();
        animal.bark();
        animal.looks();                    }}
```

3. **Hierarchical Inheritance,** multiple classes inherit from a single parent class.

```
// Parent class (Super class)
class Animal {
    void eat() {
        System.out.println("This animal eats food.");    }}

// Child class (Sub class) 1
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");           }}

// Child class (Sub class) 2
class Cat extends Animal {
    void meow() {
        System.out.println("The cat meows.");        }}

public class HierarchicalInheritance {
    public static void main(String[] args) {
        // Creating objects of Dog and Cat class
        Dog myDog = new Dog();
        Cat myCat = new Cat();

        myDog.eat();  // Inherited from Animal
        myDog.bark(); // Defined in Dog

        myCat.eat();  // Inherited from Animal
        myCat.meow();               }}
```

4. ***(Multiple Inheritance)*** – via **interfaces** only. Java does not supports multiple inheritance because of ambiguity problem of the methods. Which is solved using interfaces.

```
class Printer {
    void print() {
        System.out.println("Printing document...");      }}

class Scanner {
    void print() {
        System.out.println("Scanning document...");      }}

class MultiFunctionDevice extends Printer, Scanner {  // Error:
Cannot inherit from both Printer and Scanner
}
```

Ambiguity Problem (Diamond Problem):

- When a class inherits from multiple classes that have the same method, Java won't know which method to use.
- This causes **ambiguity** in the code.

5. ***(Hybrid Inheritance)*** – also via **interfaces.** Java doesn't support hybrid inheritance (a mix of multiple inheritance via classes and interfaces) directly, but you can achieve this using **multiple interfaces**. Only achieved through interfaces.

Polymorphism : Is the ability of an object to take on many forms. It allows methods to behave differently based on the object that calls them, enabling a single method or class to work in different ways depending on the context. **Example**: **Shape Drawing**

- Same method name, different behaviors based on the object.

Types of Polymorphism:

1. **Compile-time Polymorphism** (also called **Static Polymorphism** or **Method Overloading**):
    - This occurs when the method to be called is determined **at compile time**.
    - Example: **Method Overloading** (same method name, but with different parameters).
2. **Runtime Polymorphism** (also called **Dynamic Polymorphism** or **Method Overriding**):
    - This occurs when the method to be called is determined **at runtime**.
    - Example: **Method Overriding** (child class overrides the method of the parent class).
    - **Dynamic method dispatch or Runtime Polymorphism**, Is a mechanism by which a call to an overridden method is resolved at the runtime, rather than compile time.

Method Overloading and Method OverRiding

Feature	Method Overloading	Method Overriding
Class Level	Same class	Involves parent and child classes
Method Signature	Must be **different**	Must be **same**
Return Type	Can be same or different	Must be same or covariant
Polymorphism Type	Compile-time polymorphism	Run-time polymorphism
Annotation Used	Not required	`@Override` is recommended

Method Overloading (Compile-Time Polymorphism**) :** Method overloading occurs when multiple methods in the same class have the same name but different parameters (type, number, or order).

- Happens **within the same class.**
- Based on **different method signatures.**
- JVM decides which method to call **at compile time.**

```
class MethodOverloading_Calculator{
    void add(int a, int b) {
        System.out.println(a + b);
    }

    void add(double a, double b) {
        System.out.println(a + b);
    }
}
```

```
public class MethodOverloading {
    public static void main(String[] args) {
        MethodOverloading_Calculator calc = new
MethodOverloading_Calculator();
        calc.add(2, 3); // Calls the int version
        calc.add(2.0, 3.0); // Calls the double version
    }
}
```

Method OverRiding (Run-Time Polymorphism**) :** Method overriding occurs when a **child class provides its own version** of a method that is already defined in its **parent class**.

- Happens in **inheritance,** Because **overriding always happens between a parent class and its child class** — so inheritance is **required**
- Method signature must be **exactly the same**
- JVM decides which method to call **at runtime**

Compile Time: The time when the Java compiler checks for syntax errors and translates the code into bytecode. Errors like **syntax errors**, **missing semicolons**, and **incorrect types** are caught during compile time.

Run Time: The time when the program is executed, and runtime errors like division by zero or null pointer exceptions occur. Errors like **dividing by zero**, **null pointer exception**, or **array index out of bounds** happen at runtime.

```
class Animal {

    void sound() {
        System.out.println("Animal makes a sound");
    }}

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }}
public class MethodOverRiding {
    public static void main(String[] args) {
        Animal obj = new Animal();
        Animal obj1 = new Dog();
        obj.sound();
        obj1.sound();
    }}
```

```
class Calculator{
    void add(int a , int b){
        System.out.println(a+b);
    }
    void add(int a , int b , int c){
        System.out.println(a+b+c);
    }
    void add(double a , double b){
        System.out.println(a + b);
    }
}
class AdvancedCalculator extends Calculator{
    void add(int a,int b){
        System.out.println("The addition of two numbers are :" +
(a+b));
    }
}
```

```
public class MethodOverloading_Overriding{
    public static void main(String args[]){
        Calculator obj1 = new Calculator();
        obj1.add(2,3);
        obj1.add(1,2,3);
        obj1.add(1,2);

        Calculator obj2 = new AdvancedCalculator();
        obj2.add(3,5);
    }
}
```

Abstraction : Abstraction is the process of hiding the internal implementation details and showing only the functionality to the user. Example : When you drive a car, you just use the steering, accelerator, and brakes without knowing how the engine works internally.

- Abstract method is only belonging to abstract class.
- Abstract class **cannot** be instantiated.
- Abstract class is restricted class we can't use to create a objects.
- If we are defining abstract class , we must need to define it compulsory in the inherited class else it shows error.
- Its not mandatory to have abstract method in abstract class.
- Abstract class can have constructors, fields, and methods with implementations.

✓ **Abstract methods** (without a body) – that must be implemented in the subclass.

✓ **Concrete methods** (with a body) – that can be inherited as-is or overridden.

```
abstract class Animal {
    abstract void sound();  // Abstract method

    void sleep() {
        System.out.println("Sleeping...");      }}

class Dog extends Animal {
    void sound() {               // Concrete method
        System.out.println("Dog barks");        }}

public class Abstraction {
    public static void main(String args[]) {
        Dog d = new Dog();
        d.sound(); // Dog barks
        d.sleep(); // Sleeping...
        }}
```

Without abstract method :

```
abstract class Vehicle {
    void display() {
        System.out.println("This is a vehicle.");   }}

class Car extends Vehicle {
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.display();            }}
```

Example: **ATM Machine.**

```
abstract class ATM {
    abstract void withdraw();
    abstract void deposit();        }

class SBIATM extends ATM {
    void withdraw() {
        System.out.println("Withdrawing from SBI ATM...");      }

    void deposit() {
        System.out.println("Depositing to SBI ATM...");     }}

public class Main {
    public static void main(String[] args) {
        SBIATM atm = new SBIATM();
        atm.deposit();
        atm.withdraw();     }}
```

```
abstract class ATM {
    double balance = 0;

    abstract void withdraw(double amount);
    abstract void deposit(double amount);

    void checkBalance() {
        System.out.println("Current Balance: " + balance);      }}

class SBIATM extends ATM {
    void withdraw(double amount) {
        if (amount > 0 && amount + 5 <= balance) {
            balance -= (amount + 5.0);
            System.out.println("Withdrawn from SBI with 5rs charge.
Remaining Balance: " + balance);
        } else {
            System.out.println("Insufficient balance or invalid
amount in SBI ATM.");   }}
```

```
    void deposit(double amount) {
        if (amount > 2) {
            balance += (amount - 2.0);
            System.out.println("Deposited to SBI with 2rs charge.
Current Balance: " + balance);
        } else {
            System.out.println("Amount too low to deposit in SBI
ATM.");        }}}

class HDFCATM extends ATM {
    void withdraw(double amount) {
        if (amount > 0 && amount + 10 <= balance) {
            balance -= (amount + 10.0);
            System.out.println("Withdrawn from HDFC with 10rs
charge. Remaining Balance: " + balance);
        } else {
            System.out.println("Insufficient balance or invalid
amount in HDFC ATM.");      }}

    void deposit(double amount) {
        if (amount > 3) {
            balance += (amount - 3.0);
            System.out.println("Deposited to HDFC with 3rs charge.
Current Balance: " + balance);
        } else {
            System.out.println("Amount too low to deposit in HDFC
ATM.");       }}}

public class AbstractExample {
    public static void main(String args[]) {
        SBIATM sbiatm = new SBIATM();
        sbiatm.deposit(500);
        sbiatm.withdraw(100);
        sbiatm.checkBalance();

        System.out.println();

        HDFCATM hdfcatm = new HDFCATM();
        hdfcatm.deposit(300);
        hdfcatm.withdraw(100);
        hdfcatm.checkBalance();             }}
```

Types of Abstraction in Java:

1. **Partial Abstraction**
   - Achieved using **abstract classes**.
   - Can have both **abstract and concrete** methods.
2. **Full Abstraction**
   - Achieved using **interfaces**.
   - Only **abstract methods** (unless default/static).
   - All methods are **public & abstract** by default.

```
abstract class Animal {
    abstract void sound();      }
abstract class Bird {
    abstract void fly();        }
// ❌ Not allowed: Java doesn't support this
// class Bat extends Animal, Bird { ... }
```
(So to solve this issue we can choose interfaces)
```
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");    }}
```

- Have **many abstract classes** defined.
- Choose **one** to extend in a subclass.

Interface : It is a mechanism in java to achieve abstraction & is defined as an abstract type used to specify the behaviour of class.

An interface is a way to declare that some group of behaviors (functions) exist. A type can then implement the interface and provide the implementation for the behaviors expected by the interface. This enables functions to operate on the interface instead of a specified type.

- Used to achieve full abstraction.
- We can support the functionality of multiple inheritance.
- Used to achieve loose coupling.
- Interface is not a class, By default every method in interface is public abstract.
- 1 Class can implement multiple interfaces.
- In interfaces By default the variables are final & static.
- Every variable is **public, static, and final** by default.

- Class – Class = extends
- Class – Interface = implements
- Interface – Interface = extends

Types of Interfaces :

1. Normal Interface

- Contains method declarations.
- Classes must implement all its methods.

```
interface Vehicle {
    void start();
    void stop();        }

class Car implements Vehicle {
    public void start() {
        System.out.println("Car started");      }
    public void stop() {
        System.out.println("Car stopped");      }}
```

---

2. Marker Interface

- **No methods** at all.
- Used to "mark" or "tag" a class (e.g., `Serializable`, `Cloneable`).

```
interface MarkerInterface {}  // No methods

class Product implements MarkerInterface {
    int id = 101;                           }
```

---

3. Functional Interface

- Contains **only one abstract method**.
- Used in **lambda expressions** and **functional programming**.

```
@FunctionalInterface
interface Calculator {
    int operation(int a, int b);        }

class LambdaExample {
    public static void main(String[] args) {
        Calculator add = (a, b) -> a + b;
        System.out.println("Sum: " + add.operation(5, 3));      }}
```

Interface Example :

```
interface Animal {
    void sound();  // abstract method
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}


public class Interface {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();                  }}
```

```
interface Vehicle{
    void start();
    void stop();
}
class Car implements Vehicle{
    public void start(){
        System.out.println("Car is Starting...!");
    }
    public void stop(){
        System.out.println("Car is Stoping...!");
    }
}

class Bike implements Vehicle{
    public void start(){
        System.out.println("Bike is Starting...!");
    }
    public void stop(){
        System.out.println("Bike is Stoping...!");
    }
}

public class InterfaceExample {
    public static void main(String args[]) {
        Car car = new Car();
        car.start();
        car.stop();

        System.out.println();

        Bike bike = new Bike();
        bike.start();
        bike.stop();
    }}
```

- Always use **public** when implementing methods from an interface.
- Otherwise, Java won't recognize it as a valid override.

```java
interface Flyable {
    void fly();    }

interface Swimmable {
    void swim();       }

class Duck implements Flyable, Swimmable {
    public void fly() {
        System.out.println("Duck is flying!");      }

    public void swim() {
        System.out.println("Duck is swimming!");     }}

public class MultipleInterfaces {
    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.fly();
        duck.swim();            }}
```

Feature	Abstract Class	Interface
Keyword Used	`abstract class`	`interface`
Methods	Can have both **abstract** and **concrete** methods	Only **abstract methods** (until Java 7), **default & static methods** allowed (from Java 8)
Variables	Can have instance variables	Variables are **public**, **static**, and **final** by default
Inheritance	Can **extend only one abstract class**	Can **implement multiple interfaces**
Access Modifiers	Can have any access modifiers	Methods are **public** by default, must be public when implemented
Constructor	Can have constructors	Cannot have constructors
Use Case	When classes share common base logic	To define **contract** (set of behaviors) for unrelated classes
Level of Abstraction	**Partial Abstraction** (can have implementation)	**Full Abstraction** (no implementation until Java 8)