Object-Oriented Programming (OOP) in Java

Object-Oiented Programming (OOP) is a programming style that organizes a program into **objects**.

- **Object** = real-world entity (example: Student, Car, Employee).
- Each object has:
    - **Attributes (data/properties)** → what it has
    - **Methods (functions/behavior)** → what it does

Principles of OOP

1. **Encapsulation**
    - Wrapping data and methods together inside a class.
    - Provides **data hiding** → outside world can't directly access sensitive data.
    - Access through getters & setters.
2. **Inheritance**
    - One class can acquire properties and behaviors of another class.
    - Promotes **code reusability** and represents **IS-A relationship** (Dog IS-A Animal).
3. **Polymorphism**
    - "Many forms".
    - A single method/object behaves differently in different situations.
    - Two types:
        - Compile-time (Method Overloading)
        - Runtime (Method Overriding)
4. **Abstraction**
    - Hiding implementation details and showing only important features.
    - Achieved using **abstract classes** or **interfaces**.
    - Example: ATM machine shows buttons but hides internal working.

Creating a New Class in Java

1. What is a Class

- A **class** is a blueprint for creating objects.
- It contains **fields (variables)** and **methods (functions)**.

2. How to Create a Class

```
// Defining a new class
class Student {
    // Fields (variables)
    private String name;
    private int age;

    // Methods (functions)
    public void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

`Student` is the class, and inside it we have **variables** (`name`, `age`) and a **method** (`displayInfo`).

Getters and Setters in Java

1. Why use Getters and Setters?

- To follow **Encapsulation** (hiding data using `private` keyword).
- Direct access to variables is **not allowed**.
- We use **getters** to read values, and **setters** to update values safely.

2. Example: Getters and Setters

```
class Student {
    // Private fields (cannot be accessed directly)
    private String name;
    private int age;

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter for age
    public void setAge(int age) {
        this.age = age;
    }
}
```

3. Using the Class with Getters & Setters

```
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();

        // Setting values using setters
        s1.setName("Neeraj");
        s1.setAge(23);

        // Getting values using getters
        System.out.println("Name: " + s1.getName());
        System.out.println("Age: " + s1.getAge());
    }
}
```

Access Modifiers in Java

1. **public** → Accessible **from anywhere** (same class, same package, different package, subclasses).
2. **protected** → Accessible **within same package** and also **in subclasses (even if in different package)**.
3. **default (no modifier / package-private)** → Accessible only **within same package**.
4. **private** → Accessible only **within the same class**.

What is a Constructor?

- A **constructor** is a **special method** in Java that is used to **initialize objects**.
- It is automatically called when an object is created.
- Constructor ka naam **class ke naam jaisa hi hota hai** aur iska **koi return type nahi hota (not even void)**.

Key Points

1. Constructor name = Class name.
2. No return type.
3. Called automatically when `new` keyword is used.
4. Can be **overloaded** (multiple constructors with different parameters).
5. If you don't define a constructor → Java provides a **default constructor**.

Types of Constructors in Java

- **Default Constructor**

No parameters.
Provided by Java if we don't create any constructor.
Initializes objects with **default values**.

- **Example (conceptual):**

int = 0, float = 0.0, boolean = false, String/Object = null.

- **Parameterized Constructor**

Takes arguments to assign values to attributes.
Allows initialization of objects with specific values.

Constructor Overloading:

Having more than one constructor in the same class with different parameter lists.
Helps in creating objects in multiple ways.

Example:

```
// Class
class Student {
    String name;
    int age;

    // 1. Default Constructor
    Student() {
        name = "Unknown";
        age = 0;
        System.out.println("Default constructor called");
    }

    // 2. Parameterized Constructor
    Student(String n, int a) {
        name = n;
```

```
        age = a;
        System.out.println("Parameterized constructor called");
    }

    // Method to display student details
    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        // Object using default constructor
        Student s1 = new Student();
        s1.display();

        // Object using parameterized constructor
        Student s2 = new Student("Neeraj", 23);
        s2.display();
    }
}
```

Inheritance in Java

1. What is Inheritance?

- Inheritance is a concept where **one class (child class)** acquires the **properties and methods** of another class (parent class).
- It allows **code reusability** and supports **hierarchical relationships** in OOP.

In simple words: **"Child class can reuse parent's features."**

Example:

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

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

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();   // Inherited from Animal
        d.bark();  // Defined in Dog
    }
}
```

Output:

```
This animal eats food.
Dog barks.
```

Types of Inheritance in Java

1. **Single Inheritance** – One class inherits another.
   (A → B)
2. **Multilevel Inheritance** – A class inherits another, which itself is inherited by a third.
   (A → B → C)
3. **Hierarchical Inheritance** – Multiple classes inherit a single parent class.
   (A → B, A → C)

Multiple Inheritance using classes is **not allowed** in Java (to avoid ambiguity). But it can be achieved using **interfaces**.

Important Keywords

- `extends` → Used to inherit a class.
- `super` → Refers to the parent class (used to call parent methods/constructors).

Advantages

- **Code reusability** (no need to rewrite code).
- **Method overriding** support (Polymorphism).
- Creates a **hierarchy** of classes.

Constructor in Inheritance

- In inheritance, **parent class constructor** always runs before the **child class constructor**.
- Reason: Parent's part of object must be initialized first.
- We use `super()` to explicitly call parent's constructor (default or parameterized).

Case 1: Default Constructor

```
class Employee {
    String name;
    double salary;

    // Default constructor
    Employee() {
        name = "Unknown";
        salary = 0.0;
        System.out.println("Employee constructor called");
    }
}

class Manager extends Employee {
    Manager() {
        System.out.println("Manager constructor called");
    }
}

public class Main {
    public static void main(String[] args) {
        Manager m = new Manager(); // Object created
        System.out.println("Name: " + m.name + ", Salary: " + m.salary);
    }
}
```

Output:

```
Employee constructor called
Manager constructor called
Name: Unknown, Salary: 0.0
```

Case 2: Parameterized Constructor with super()

```
class Employee {
    String name;
    double salary;

    // Parameterized constructor
    Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
        System.out.println("Employee constructor called for " + name);
    }
}

class Manager extends Employee {
    String department;

    // Child constructor
    Manager(String name, double salary, String department) {
        super(name, salary); // Call parent constructor
        this.department = department;
        System.out.println("Manager constructor called for " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Manager m = new Manager("Neeraj", 50000, "IT");
        System.out.println("Name: " + m.name + ", Salary: " + m.salary + ",
Department: " + m.department);
    }
}
```

Output:

```
Employee constructor called for Neeraj
Manager constructor called for Neeraj
Name: Neeraj, Salary: 50000.0, Department: IT
```

1. `this` Keyword

- Refers to the **current object** of the class.
- Used to differentiate between **class variables** and **method/local variables** when they have the same name.

Uses of `this`

1. **Refers to current object**
2. **Differentiate variables** (when local variable & instance variable have same name)
3. **Call current class methods**
4. **Call current class constructor** (using `this()`)

Example with `name` and `salary`

```
class Employee {
    String name;
    double salary;
```

```
    Employee(String name, double salary) {
        this.name = name;       // "this" refers to instance variable
        this.salary = salary;
    }

    void display() {
        System.out.println("Name: " + this.name + ", Salary: " +
this.salary);
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e1 = new Employee("Neeraj", 50000);
        e1.display();
    }
}
```

Output:

```
Name: Neeraj, Salary: 50000.0
```

2. super Keyword

- Refers to the **immediate parent class object**.
- Used in **inheritance** to access parent's variables, methods, and constructors.

Uses of super

1. **Access parent class variables** (if same name exists in child class).
2. **Call parent class methods** (if overridden in child class).
3. **Call parent class constructor** (first line of child constructor).

Example with name and salary

```
class Employee {
    String name;
    double salary;

    Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
        System.out.println("Employee constructor called for " + name);
    }

    void display() {
        System.out.println("Employee: " + name + ", Salary: " + salary);
    }
}

class Manager extends Employee {
    String department;

    Manager(String name, double salary, String department) {
        super(name, salary); // calls parent constructor
        this.department = department;
        System.out.println("Manager constructor called for " + name);
    }

    @Override
```

```
    void display() {
        super.display(); // calls parent method
        System.out.println("Department: " + department);
    }
}

public class Main {
    public static void main(String[] args) {
        Manager m = new Manager("Rahul", 70000, "IT");
        m.display();
    }
}
```

Output:

```
Employee constructor called for Rahul
Manager constructor called for Rahul
Employee: Rahul, Salary: 70000.0
Department: IT
```