OOP concept in Java

give me the deep and detailed expalination of the oops concept in java.

so that i am able to understand it well and do not have ant doubts so explain it in deep.

I'll explain OOP in Java from **zero to deep**, step by step, with simple examples.
If you read carefully and maybe type the examples in IntelliJ, you'll be very comfortable with OOP.

---

1. What is OOP in simple words?

OOP (Object-Oriented Programming) is a way of writing programs where you think in terms of:

- **Objects** → real-world entities (Student, Car, BankAccount, Employee)
- **Classes** → blueprints/templates to create those objects

OOP in Java is mainly based on **4 pillars**:

1. **Encapsulation** – data hiding + bundling data and methods together
2. **Inheritance** – one class reusing properties/behaviors of another
3. **Polymorphism** – one name, many forms (same method name behaves differently)
4. **Abstraction** – show only necessary details, hide internal complexity

Before the 4 pillars, you must be very clear about **Class & Object**.

---

2. Class and Object

2.1 Class

A **class** is a blueprint or template.

Example: Blueprint of a `Car` – it describes **what properties and behaviors a car should have**, but it is not an actual car.

In Java:

```java
class Car {
    String color;       // property / field / attribute
    String model;
    int speed;
    void start() {      // method / behavior
        System.out.println("Car started");
    }

    void accelerate(int value) {
        speed = speed + value;
        System.out.println("Car speed: " + speed);
    }
}
```

Here:

- `Car` is a **class**
- `color`, `model`, `speed` → **fields / instance variables**
- `start()`, `accelerate()` → **methods**

2.2 Object

An **object** is a **real instance** of a class.
Using the blueprint (`class`), you build a real car (`object`).

```java
java

public class Main {
    public static void main(String[] args) {
        Car c1 = new Car();    // object 1
        c1.color = "Red";
        c1.model = "Swift";
        c1.start();
        c1.accelerate(20);

        Car c2 = new Car();    // object 2
        c2.color = "Black";
        c2.model = "Creta";
        c2.start();
        c2.accelerate(30);
    }
}
```

- `c1` and `c2` are **two separate objects**.
- Both are created from the same class `Car`, but they have **different data** (`color`, `model`, `speed`).

Key idea:

- **Class** = definition of **what it is & what it can do**
- **Object** = **actual thing** in memory

---

3. Encapsulation (Data Hiding + Bundling)

3.1 What is encapsulation?

Encapsulation means:

1. **Bundling**: keeping data (variables) and methods together in one unit → a class
2. **Hiding**: restricting direct access to internal data, and providing controlled access through methods

Why?

- To **protect data** from being changed in an invalid way
- To **control** how outside code interacts with your object
- To make your code **safe and maintainable**

3.2 How we do encapsulation in Java?

We use:

- **`private`** for fields (variables)
- **`public`** methods (**`getters`** / **`setters`**) to access/update fields

Example without encapsulation:

```java
class Account {
    public int balance;
}
```

Anyone can do:

```java
Account a = new Account();
a.balance = -1000000000; // invalid value - but compiler allows this
```

Problem: No control.

3.3 Encapsulated version

```java
class Account {
    private int balance;      // hidden

    public int getBalance() { // getter
        return balance;
    }

    public void deposit(int amount) { // setter-style method
        if (amount > 0) {
            balance += amount;
        } else {
            System.out.println("Invalid deposit amount");
        }
    }

    public void withdraw(int amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        } else {
            System.out.println("Invalid withdraw amount");
        }
    }
}
```

Now usage:

```java
public class Main {
    public static void main(String[] args) {
        Account acc = new Account();
        acc.deposit(500);              // valid
        acc.withdraw(1000);            // prints error message
        System.out.println(acc.getBalance());
    }
}
```

We are **not directly touching `balance`**, we are using methods.
This is **encapsulation**.

3.4 Access modifiers recap

- **`private`** – accessible only inside the same class
- **`default`** (no keyword) – same package
- **`protected`** – same package + subclasses (even in different packages)
- **`public`** – accessible from everywhere

Encapsulation usually = **`private` fields + `public` methods**.

---

4. Inheritance (Reusability, IS-A)

4.1 What is inheritance?

Inheritance means: one class (child) **inherits** properties and methods from another class (parent).

- Parent class: **superclass / base class**
- Child class: **subclass / derived class**

Syntax:

```java
class Parent {
    // fields, methods
}

class Child extends Parent {
    // extra fields, extra methods
}
```

4.2 Why use inheritance?

- To **reuse** code instead of writing same things again and again
- To model **IS-A** relationships

Example:

`Dog` is a `Animal`

`Car` is a `Vehicle`

4.3 Example

```java
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}
```

Usage:

```java
public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();   // inherited from Animal
        d.bark();  // Dog's own
    }
}
```

Because `Dog` extends `Animal`, `Dog` has both **eat()** and **bark()**.

4.4 Types of inheritance in Java

By concept:

1. **Single inheritance**

   ```java
   class A {}
   class B extends A {}
   ```

2. **Multilevel inheritance**

   ```java
   class A {}
   class B extends A {}
   class C extends B {}
   ```

   `C` inherits from `B`, and indirectly from `A`.

3. **Hierarchical inheritance**

```java
class A {}
class B extends A {}
class C extends A {}
```

`B` and `C` both inherit from `A`.

> Java **does NOT support multiple inheritance of classes** like:
> `class C extends A, B {}` ❌
> Instead, we use **interfaces** for that (we'll see in abstraction).

4.5 `super` keyword

Used to:

1. Call **parent class constructor**
2. Call **parent class method** when overridden

```java
class Animal {
    Animal() {
        System.out.println("Animal constructor");
    }

    void eat() {
        System.out.println("Animal eats");
    }
}

class Dog extends Animal {
    Dog() {
        super(); // calls Animal()
        System.out.println("Dog constructor");
    }

    void eat() {
        System.out.println("Dog eats");
    }

    void printParentEat() {
        super.eat(); // calls Animal's eat()
    }
}
```

---

5. Polymorphism (One Name, Many Forms)

Polymorphism: “poly” = many, “morph” = forms.

In Java:

- **Compile-time polymorphism** → method overloading
- **Run-time polymorphism** → method overriding + dynamic method dispatch

5.1 Method Overloading (Compile-time polymorphism)

Same method name, different parameter list in same class.

```
java

class MathUtil {
    // 1
    int add(int a, int b) {
        return a + b;
    }

    // 2
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // 3
    double add(double a, double b) {
        return a + b;
    }
}
```

Here, `add()` is **overloaded**.

Selection happens at **compile time**, based on:

- number of parameters
- types of parameters
- order of parameters

5.2 Method Overriding (Run-time polymorphism)

Same method signature (name + parameters) in **parent and child** classes.

```
java

class Animal {
    void sound() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Bark");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Meow");
    }
}
```

Usage:

```
java

public class Main {
    public static void main(String[] args) {
        Animal a1 = new Dog();  // upcasting
        Animal a2 = new Cat();
        a1.sound(); // Bark
        a2.sound(); // Meow
    }
}
```

- Variable type: `Animal`
- Actual object: `Dog` / `Cat`
- Method called is based on **actual object at runtime**, not reference type.

This is called **dynamic method dispatch / run-time polymorphism**.

5.3 Upcasting and dynamic dispatch

```java
Animal a = new Dog(); // upcasting
a.sound();            // Dog's sound() called
```

Even though the reference type is `Animal`, the JVM sees the **object is Dog**, so it calls **Dog's overridden method**.
This is **run-time polymorphism**.

---

6. Abstraction (Show Only What's Needed)

6.1 What is abstraction?

Abstraction means:

- Show only necessary details
- Hide complex internal logic

Real life example:
You use a **mobile phone** – you just click on icons. You don't care about internal circuits or hardware working.

In Java, abstraction is implemented using:

1. **Abstract classes**
2. **Interfaces**

6.2 Abstract Classes

- Declared with `abstract` keyword
- Can have:
  - **Abstract methods** (no body)
  - **Non-abstract methods** (with body)

  - Fields, constructors
- Cannot be instantiated directly

```java
abstract class Shape {
    abstract double area(); // abstract method

    void printShape() {     // normal method
        System.out.println("I am a shape");
    }
}

class Circle extends Shape {
    double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * radius * radius;
    }
}
```

Usage:

```java
public class Main {
    public static void main(String[] args) {
        // Shape s = new Shape(); // ❌ cannot create object of abstract class

        Shape s = new Circle(5); // ✅ allowed
        s.printShape();
        System.out.println("Area: " + s.area());
    }
}
```

Abstract class gives **partial implementation** and leaves **some parts (abstract methods) to be implemented by child classes**.

6.3 Interfaces

Interface is a completely abstract type (contract).

- Before Java 8: only abstract methods + constants
- Now (Java 8+): can have **`default`** and **`static`** methods with body

Use **`interface`** keyword:

```java
interface Flyable {
    void fly(); // public abstract by default
}
```

Class implementing interface:

```java
class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Bird is flying");
    }
}

class Airplane implements Flyable {
    @Override
    public void fly() {
        System.out.println("Airplane is flying");
    }
}
```

Usage:

```java
public class Main {
    public static void main(String[] args) {
        Flyable f1 = new Bird();
        Flyable f2 = new Airplane();

        f1.fly();
        f2.fly();
    }
}
```

6.4 Difference: Abstract class vs Interface (simple)

- **Abstract class**
  - **`extends`**
  - Can have fields, constructors, normal methods, abstract methods
  - A class can extend **only ONE** abstract class (because Java has single inheritance of classes)
- **Interface**
  - **`implements`**
  - Mainly used for defining **capabilities / behavior contracts**
  - A class can implement **multiple interfaces** (this is how Java supports something like multiple inheritance)

Example:

```java
class A { }

interface I1 { }
interface I2 { }

class B extends A implements I1, I2 { }
```

---

7. “Has-A” vs “Is-A” (Composition vs Inheritance)

This is not one of the 4 pillars, but very important in OOP design.

7.1 IS-A (Inheritance)

`Dog` **is a** `Animal` → `Dog extends Animal`

Use inheritance when:

- Child is truly a **type of** parent
- You want to **reuse behavior** and also enable polymorphism

7.2 HAS-A (Composition)

`Car` **has an** `Engine` → `Engine` is a separate class, and `Car` has a field of type `Engine`.

```java
class Engine {
    void startEngine() {
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine; // HAS-A relation

    Car() {
        this.engine = new Engine();
    }

    void startCar() {
        engine.startEngine();
        System.out.println("Car started");
    }
}
```

Here:

- `Car` is **not an** `Engine`, so we don't use inheritance.
- We use **composition** → Car **has** Engine

Good OOP design often prefers **composition over inheritance**.

---

8. How all four pillars work together (Mini example)

Let's design a small system using all 4 concepts.

Requirement:

- We have different types of employees:
    - `FullTimeEmployee`
    - `PartTimeEmployee`
- All employees have:
    - `id`, `name`

  - `calculateSalary()` but formula differs

We’ll use:

- **Abstraction**: abstract class `Employee`
- **Encapsulation**: `private` fields + getters/setters
- **Inheritance**: `FullTimeEmployee` & `PartTimeEmployee` extend `Employee`
- **Polymorphism**: same method `calculateSalary()` behaves differently

Code:

```java
abstract class Employee {              // Abstraction
    private int id;                    // Encapsulation
    private String name;

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // getters only (read-only for now)
    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    // abstract method
    public abstract double calculateSalary(); // Polymorphism (overridden in subclasses)
}

class FullTimeEmployee extends Employee {     // Inheritance
    private double monthlySalary;

    public FullTimeEmployee(int id, String name, double monthlySalary) {
        super(id, name);
        this.monthlySalary = monthlySalary;
    }

    @Override
    public double calculateSalary() {         // Polymorphism (override)
        return monthlySalary;
    }
}

class PartTimeEmployee extends Employee {     // Inheritance
    private int hoursWorked;
    private double hourlyRate;

    public PartTimeEmployee(int id, String name, int hoursWorked, double hourlyRate) {
        super(id, name);
        this.hoursWorked = hoursWorked;
        this.hourlyRate = hourlyRate;
    }

    @Override
    public double calculateSalary() {         // Polymorphism (override)
        return hoursWorked * hourlyRate;
    }
}
```

Usage:

```java
public class Main {
    public static void main(String[] args) {
        Employee e1 = new FullTimeEmployee(1, "Piyush", 50000);
        Employee e2 = new PartTimeEmployee(2, "Rahul", 80, 300);

        System.out.println(e1.getName() + " salary: " + e1.calculateSalary());
        System.out.println(e2.getName() + " salary: " + e2.calculateSalary());
    }
}
```

Points:

- **`Employee`** is **abstract**: you cannot create **`new Employee()`**
- **`FullTimeEmployee`** and **`PartTimeEmployee`** **inherit** from **`Employee`**
- All fields are **private** = encapsulation
- Same method **`calculateSalary()`** is called on **`Employee`** type, but actual implementation depends on the **actual object** (**`FullTimeEmployee`** or **`PartTimeEmployee`**) → **polymorphism**

---

9. Very quick summary (for revision)

- **Class** – blueprint
- **Object** – real instance of a class
- **Encapsulation**
  - Hiding data with **`private`**
  - Using **`public`** methods (**`get`**/**`set`**) to control access
- **Inheritance**
  - **`extends`** keyword
  - Child gets properties/methods of parent
  - IS-A relationship
- **Polymorphism**
  - Overloading (compile-time) → same name, different parameters
  - Overriding (run-time) → same signature in parent & child
  - Dynamic dispatch via parent reference → child object
- **Abstraction**
  - Hide complexity
  - Using **abstract classes** and **interfaces**

If you want, next I can:

- Give you **small OOP questions** to practice (and then we can discuss your answers), or

- Take one **real-world mini project** (e.g., Library System / Banking System) and design full OOP structure in Java with you step-by-step.