🔹 Q1. What is Object-Oriented Programming (OOP)?

📖 Definition

OOP (Object-Oriented Programming) is a **programming paradigm** where software is designed around **objects** instead of just functions and logic.

- An **object** is something that represents a **real-world entity**.
- Each object has **state** (data/attributes) and **behavior** (functions/methods).
- We create **classes** to define objects, and then we use objects to perform tasks.

✅ Why OOP?

- Helps organize complex code in a structured way.
- Makes code **reusable** (you can reuse classes in different programs).
- Provides **scalability** (easy to add new features).
- Improves **maintainability** (bugs can be fixed in one place).

🔹 Real-world Analogy

- Think of a **Car**:
  - **State** → color, brand, model.
  - **Behavior** → drive, brake, honk.
- In OOP, we model this as a class → create multiple objects (e.g., `Car c1`, `Car c2`).

🔹 Example (Java)

```
class Car {
   String color;   // attribute (state)

   void drive() {  // method (behavior)
      System.out.println("Car is driving...");
   }
}

public class Main {
```

```
    public static void main(String[] args) {
        Car car1 = new Car();   // object
        car1.color = "Red";
        car1.drive();           // Output: Car is driving...
    }
}
```

🎯 Interview Tip

Answer like:
"OOP is a programming style based on real-world objects. It helps bundle **data and behavior** into classes, making software **modular, reusable, and easier to maintain**."

🔹 Q2. What are the four main pillars of OOP?

📖 The 4 Pillars

1. **Encapsulation** → Wrapping variables and methods into a single unit (class) and controlling access.
2. **Abstraction** → Showing only essential features and hiding background details.
3. **Inheritance** → Acquiring properties and behaviors from another class.
4. **Polymorphism** → Same method behaving differently in different situations.

🔹 Real-world Analogies

- **Encapsulation** → Medicine capsule (contents hidden inside, only the effect is visible).
- **Abstraction** → Driving a car (you only use steering, brakes → internal engine mechanism hidden).
- **Inheritance** → Child inherits traits from parents.
- **Polymorphism** → The word "run" → a person runs, a program runs, a machine runs.

🎯 Interview Tip

Say: *"The 4 pillars are Encapsulation, Abstraction, Inheritance, Polymorphism. They allow OOP to build secure, reusable, and flexible systems."*

🔹 **Q3. What is a class and what is an object?**

📖 **Class**

- A **blueprint** or **template** that defines attributes (data) and methods (behavior).
- A class itself does not occupy memory until we create an object.

📖 **Object**

- A **real-world instance** of a class.
- Occupies memory.
- Has actual values for the attributes.

🔹 **Real-world Analogy**

- **Class** → Blueprint of a house.
- **Object** → The actual house built using the blueprint.
- We can build multiple houses (objects) from one blueprint (class).

🎯 **Interview Tip**

Keep it short:

- *"Class is a design, Object is an instance. Class defines, Object realizes."*

🔹 **Q4. What is the difference between class and object?**

📖 **Detailed Difference**

Class	Object
A **blueprint/template** for creating objects	A **real-world instance** created from a class
Does not occupy memory (logical entity)	Occupies memory (physical entity)

Defines properties (variables) and methods	Has actual values for properties
Example: `Car` class defines model, color, drive()	`Car c1 = new Car()` is an object

🔹 **Example**

```
class Car {
   String model;
}
public class Main {
   public static void main(String[] args) {
      Car c1 = new Car();  // object
      c1.model = "Honda";  // object has actual value
   }
}
```

🎯 **Interview Tip**

Say:
"A class is a definition, an object is a realization. Class is like a recipe, object is the dish prepared from it."

🔹 **Q5. What is Encapsulation? Give an example.**

📖 **Definition**

Encapsulation = **Wrapping data and methods together in a single unit** (class).

- It hides internal data from the outside world (**data hiding**).
- Access is given only through **methods (getters/setters)**.

🔹 **Why Encapsulation?**

- Protects data from unauthorized access.
- Maintains control over how data is modified.
- Increases code **security and flexibility**.

🔹 **Real-world Analogy**

- Think of a **bank ATM**:
    - You can only deposit/withdraw money using provided options.
    - You can't directly change the bank's database.
    - The sensitive data (balance) is **encapsulated** and protected.

🔹 **Example (Java)**

```
class BankAccount {
    private double balance;  // private variable (hidden)

    // public method to modify value
    public void deposit(double amount) {
        balance += amount;
    }

    // public method to read value
    public double getBalance() {
        return balance;
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount();
        acc.deposit(500);  // balance modified safely
        System.out.println(acc.getBalance()); // 500
    }
}
```

Here:

- `balance` is **hidden (private)**.
- Access is given only through **methods (public)**.

🎯 **Interview Tip**

Say:
"Encapsulation means data hiding + controlled access. It improves security and prevents misuse of data."

Perfect 👌 Let's continue with **Q6–Q10** in the same **detailed, structured way (definition + why + analogy + code + interview tip)**.

🔹 **Q6. What is Abstraction? Give an example.**

📖 **Definition**

- **Abstraction** = Showing only the **essential features** and hiding unnecessary details.
- It focuses on **what an object does** instead of **how it does it**.
- Achieved in:
  - **Java** → `abstract classes` & `interfaces`.
  - **Python** → Abstract Base Classes (ABC).
  - **C++** → Pure virtual functions.

🔹 **Why Abstraction?**

- Reduces complexity.
- Increases code flexibility (implementation can change without affecting the user).
- Improves security (internal details hidden).

🔹 **Real-world Analogy**

- Using a **TV remote** → You just press buttons to change channels, but you don't know/care about the electronic circuits inside.

🔹 **Example (Java)**

```
abstract class Vehicle {
   abstract void start();  // abstract method (no body)
}

class Car extends Vehicle {
   void start() {
      System.out.println("Car starts with a key");
   }
}

class Bike extends Vehicle {
   void start() {
```

```
        System.out.println("Bike starts with a kick");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle v1 = new Car();
        Vehicle v2 = new Bike();
        v1.start();  // Car starts with a key
        v2.start();  // Bike starts with a kick
    }
}
```

🎯 Interview Tip

Say:
"Abstraction hides implementation and only exposes essential features. It helps focus on functionality rather than internal details."

🔹 Q7. What is Inheritance? Explain with example.

📖 Definition

- **Inheritance** = Mechanism where one class (child/subclass) **acquires properties and methods** of another class (parent/superclass).
- Promotes **code reusability** and hierarchy.

🔹 Types of Inheritance

- **Single** → One child inherits one parent.
- **Multilevel** → Class inherits from a class that is itself a child.
- **Hierarchical** → Multiple classes inherit from the same parent.
- **Multiple** (not directly supported in Java, but possible via interfaces).

🔹 Real-world Analogy

- A **child inherits traits** from parents.

🔹 **Example (Java)**

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

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(); // Dog's own method
    }
}
```

🎯 **Interview Tip**

Say:
"Inheritance allows reusing code and creating a hierarchy of classes. It models real-world 'is-a' relationships (Dog is an Animal)."

🔹 **Q8. What is Polymorphism? Types of Polymorphism.**

📖 **Definition**

- **Polymorphism** = "Many forms."
- A method/action behaves **differently based on context**.

🔹 **Types**

1. **Compile-time (Static Polymorphism)** → Method overloading, operator overloading.
2. **Runtime (Dynamic Polymorphism)** → Method overriding, virtual functions.

🔹 **Real-world Analogy**

- The word **“run”** →
    - Person runs.
    - Program runs.
    - Machine runs.
      Same word → different meanings in different contexts.

🔹 **Example (Java – Method Overloading vs Overriding)**

```
// Compile-time polymorphism (Overloading)
class MathOperation {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
}

// Runtime polymorphism (Overriding)
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
public class Main {
    public static void main(String[] args) {
        MathOperation m = new MathOperation();
        System.out.println(m.add(5, 10));      // 15
        System.out.println(m.add(5.5, 4.5));  // 10.0

        Animal a = new Dog();
        a.sound(); // Dog barks
    }
}
```

🎯 **Interview Tip**

Say:
"Polymorphism means one name, many forms. It allows flexibility: same function behaves differently based on situation."

🔹 **Q9. What is the difference between Method Overloading and Method Overriding?**

📖 **Method Overloading**

- Same method name, different parameter list (number or type).
- Happens in the **same class**.
- **Compile-time polymorphism**.

📖 **Method Overriding**

- Same method name and signature, but **defined in child class**.
- Happens in **inheritance**.
- **Runtime polymorphism**.

🔹 **Example (Java)**

```
class Example {
    // Overloading
    int sum(int a, int b) { return a + b; }
    int sum(int a, int b, int c) { return a + b + c; }
}

class Parent {
    void display() {
        System.out.println("Parent display");
    }
}
class Child extends Parent {
    @Override
    void display() {
        System.out.println("Child display");
    }
}
public class Main {
    public static void main(String[] args) {
        Example e = new Example();
```

```
        System.out.println(e.sum(2, 3));       // 5
        System.out.println(e.sum(2, 3, 4));    // 9

        Parent p = new Child();
        p.display();  // Child display
    }
}
```

🎯 Interview Tip

If asked: *"Can we overload by changing return type only?"* → **No**, because compiler won't differentiate only by return type.

🔹 Q10. What are Constructors and Destructors?

📖 Constructor

- A **special method** in a class, automatically called when an object is created.
- Purpose: **Initialize object's data members**.
- In Java/Python → Constructor is used (`__init__` in Python, same name as class in Java).

📖 Destructor

- A **special method** called automatically when object is destroyed.
- Used to **free resources (like memory, files, DB connections)**.
- In Java → garbage collector handles destruction (no explicit destructor).
- In C++ → destructors (`~ClassName()`) are explicitly defined.

🔹 Example (Python)

```
class Student:
    def __init__(self, name):  # constructor
        self.name = name
        print("Student created:", self.name)

    def __del__(self):         # destructor
        print("Destructor called for", self.name)
```

```
s1 = Student("Alice")
del s1
```

🔹 **Example (Java)**

```
class Student {
    String name;
    // Constructor
    Student(String name) {
        this.name = name;
        System.out.println("Student created: " + name);
    }
}
public class Main {
    public static void main(String[] args) {
        Student s = new Student("Alice"); // Constructor called
        // Destructor is not explicit in Java; Garbage Collector handles it.
    }
}
```

🎯 **Interview Tip**

If asked in **Java context**:

- *"Java doesn't support explicit destructors. Garbage collector handles memory cleanup."*
- In **C++ context**, mention destructor explicitly.

🔹 **Q11. What is the difference between Procedural Programming and OOP?**

📖 **Procedural Programming**

- Program is divided into **functions**.
- Focus: **Step-by-step procedure** to get the output.
- Data and functions are **separate**.
- Example languages → C, Pascal.

📖 **Object-Oriented Programming**

- Program is divided into **objects**.
- Focus: **Data + behavior together**.
- Data is hidden inside classes; access only via methods.
- Example languages → Java, C++, Python.

🔹 **Comparison Table**

Feature	Procedural Programming	OOP
Structure	Divided into functions	Divided into objects/classes
Data Security	Data is exposed (global variables)	Data is encapsulated (hidden)
Reusability	Limited	High (through inheritance & polymorphism)
Example	C	Java, Python

🔹 **Real-world Analogy**

- **Procedural** → Recipe book with step-by-step instructions.
- **OOP** → Factory with machines (objects) each having specific responsibility.

🎯 **Interview Tip**

Say:
*"The key difference is that procedural focuses on **functions and procedures**, while OOP focuses on **objects and encapsulation**, making OOP more secure and reusable."*

🔹 **Q12. What is the difference between Abstraction and Encapsulation?**

📖 **Abstraction**

- Hides **implementation details**, shows only **functionality**.
- Focus: **What the object does**.

📖 **Encapsulation**

- Hides **data (variables)** by restricting direct access.
- Focus: **How the object's data is protected and accessed**.

🔹 **Key Differences**

Feature	Abstraction	Encapsulation
Meaning	Hides implementation details	Hides internal data
Achieved By	Abstract classes, Interfaces	Access modifiers (private, public, protected)
Focus	Design level (what object does)	Implementation level (how data is accessed)
Example	"Drive()" method of Car (we don't know how engine works)	Car's speed variable hidden, only accessible through setter/getter

🔹 **Real-world Analogy**

- **Abstraction** → Driving a car (you just use steering/brakes, internal mechanism hidden).
- **Encapsulation** → Car's speedometer (actual engine speed hidden; you only see controlled values).

🎯 **Interview Tip**

Say:
"Abstraction is about hiding implementation; Encapsulation is about hiding data. Abstraction is achieved via abstract classes/interfaces; Encapsulation is done via access modifiers."

🔹 **Q13. What is the difference between Interface and Abstract Class?**

📖 **Interface**

- A **contract** that defines what a class must do (methods), but not how.
- All methods are **abstract** (in Java 7 and below).
- Supports **multiple inheritance**.
- Cannot have instance variables (except `public static final` constants).

📖 **Abstract Class**

- A class with **abstract methods** (unimplemented) + **normal methods** (implemented).
- Cannot be instantiated.
- Supports **single inheritance** only.
- Can have **instance variables** and constructors.

🔹 **Comparison Table**

Feature	Interface	Abstract Class
Inheritance	Multiple inheritance possible	Only single inheritance
Variables	Only `public static final`	Normal instance variables allowed
Methods	Only abstract (Java 7) / default & static allowed (Java 8+)	Can have abstract + concrete methods
Constructor	Not allowed	Allowed
Use Case	When classes need to follow a contract	When we want partial abstraction

🔹 **Example (Java)**

```
interface Vehicle {
    void start();  // no implementation
}

abstract class Car {
    abstract void fuel();
    void wheels() {
        System.out.println("Car has 4 wheels");
    }
}

class Honda extends Car implements Vehicle {
    public void start() {
        System.out.println("Honda starts with a button");
    }
    void fuel() {
        System.out.println("Petrol used");
    }
}
```

🎯 Interview Tip

Say:
"Use Interface when you want to define a contract for multiple classes. Use Abstract Class when you want partial implementation + abstraction."

🔹 Q14. What are Access Modifiers (public, private, protected)?

📖 Definition

Access modifiers define the **scope/visibility** of classes, variables, and methods.

🔹 Types in Java

1. **Public** → Accessible everywhere.
2. **Private** → Accessible only within the same class.
3. **Protected** → Accessible within same package + subclasses (inheritance).
4. **Default (no modifier)** → Accessible within the same package only.

🔹 Example (Java)

```
class Example {
    public int x = 10;     // accessible anywhere
    private int y = 20;    // only within this class
    protected int z = 30;  // same package + subclasses
    int a = 40;            // default (package-private)
}
```

🔹 Real-world Analogy

- **Public** → Public park (anyone can access).
- **Private** → Your bedroom (only you can access).
- **Protected** → Family house (family + relatives allowed).
- **Default** → Apartment complex (only residents allowed).

🎯 Interview Tip

Say:
"Access modifiers control data visibility and security. `private` is most restrictive, `public` is least."

🔹 Q15. Can a class implement multiple interfaces?

📖 Answer

Yes ✅ (in Java, Python, C++).

- A class **can implement multiple interfaces** → allows multiple inheritance of **type**.
- Resolves **diamond problem** because interfaces only provide method signatures, not implementations.

🔹 Example (Java)

```
interface Flyable {
    void fly();
}
interface Swimmable {
    void swim();
}
class Duck implements Flyable, Swimmable {
    public void fly() { System.out.println("Duck flies"); }
    public void swim() { System.out.println("Duck swims"); }
}
public class Main {
    public static void main(String[] args) {
        Duck d = new Duck();
        d.fly();   // Duck flies
        d.swim();  // Duck swims
    }
}
```

🔹 Real-world Analogy

- A **smartphone** → acts as a **camera** and also as a **music player**. Multiple interfaces, one class.

🎯 Interview Tip

Say:
"Yes, a class can implement multiple interfaces to achieve multiple inheritance of behavior.

Unlike classes, interfaces avoid diamond problem by only declaring methods, not implementing them."

🔹 Q16. Can a class extend multiple classes? Why not?

📖 Answer

- In **Java**, a class **cannot extend multiple classes**.
- Reason: To avoid the **Diamond Problem** (ambiguity when two parent classes have the same method).
- However, Java allows **multiple inheritance through interfaces**.

🔹 Example (Java – Not Allowed)

```
class A {
   void show() { System.out.println("A"); }
}
class B {
   void show() { System.out.println("B"); }
}
// ❌ This is not allowed in Java
// class C extends A, B { }
```

🔹 Real-world Analogy

- Imagine having **two fathers** giving conflicting instructions. You'd be confused whose instructions to follow → This is the diamond problem.

🎯 Interview Tip

Say:
"A class cannot extend multiple classes in Java to avoid ambiguity (diamond problem). Instead, multiple inheritance is achieved using interfaces."

🔹 Q17. What is the Diamond Problem in Multiple Inheritance? How is it resolved?

📖 Diamond Problem

Occurs in **multiple inheritance** when a class inherits from two classes that have the same method.

- Ambiguity: Which method should the child class use?

🔹 **Example (C++ – Diamond Problem)**

```
class A {
public:
   void display() { cout << "A"; }
};
class B : public A {};
class C : public A {};
class D : public B, public C {}; // Diamond shape
```

If `D` calls `display()`, should it come from `B→A` or `C→A`? → Ambiguity.

🔹 **Solution**

- **C++** → Use **virtual inheritance**.
- **Java** → Doesn't allow multiple inheritance with classes → only via interfaces.

🔹 **Example (Java – Interfaces, No Problem)**

```
interface A {
   void display();
}
interface B {
   void display();
}
class C implements A, B {
   public void display() {
      System.out.println("Resolved: C display");
   }
}
```

🎯 **Interview Tip**

Say:
"Diamond problem occurs in multiple inheritance when ambiguity arises. Java solves it by allowing multiple inheritance only through interfaces."

🔹 **Q18. What is the difference between "is-a" and "has-a" relationship?**

📖 **IS-A (Inheritance)**

- Represents **inheritance relationship**.
- Child class is a specialized version of parent.
- Example: Dog **is-a** Animal.

📖 **HAS-A (Composition / Aggregation)**

- Represents **ownership or usage relationship**.
- One class contains another class as a member.
- Example: Car **has-a** Engine.

🔹 **Example (Java)**

```
// IS-A (Inheritance)
class Animal {}
class Dog extends Animal {}  // Dog is an Animal

// HAS-A (Composition)
class Engine {}
class Car {
    Engine engine;   // Car has-an Engine
}
```

🔹 **Real-world Analogy**

- **IS-A** → Teacher is a Person.
- **HAS-A** → Teacher has an Address.

🎯 **Interview Tip**

Say:
"IS-A means inheritance; HAS-A means composition. IS-A models specialization, HAS-A models ownership."

🔹 **Q19. What is Composition vs Aggregation vs Association?**

📖 **Association**

- A general relationship between two classes.
- Example: Student associated with Teacher.

📖 **Aggregation (Has-a, but weak)**

- A **weak "has-a" relationship** → One class can exist independently of the other.
- Example: Department has Students, but Students can exist without Department.

📖 **Composition (Has-a, but strong)**

- A **strong "has-a" relationship** → Child cannot exist without parent.
- Example: House has Rooms → If House is destroyed, Rooms don't exist.

🔹 **Example (Java)**

```
// Association
class Teacher {
   String name;
}
class Student {
   Teacher teacher; // student is associated with teacher
}

// Aggregation
class Department {
   List<Student> students; // students can exist without department
}

// Composition
class House {
   Room room; // Room cannot exist without House
}
class Room {}
```

🔹 **Real-world Analogy**

- **Association** → Doctor works in Hospital (doctor can work in many places).
- **Aggregation** → Team has Players (players can exist without team).
- **Composition** → Human has Heart (heart cannot exist without human).

🎯 Interview Tip

Say:
"Association is a general relationship, Aggregation is a weak has-a, and Composition is a strong has-a relationship."

🔹 Q20. Difference between Compile-time and Runtime Polymorphism.

📖 Compile-time Polymorphism (Static Binding)

- Decided at **compile time**.
- Achieved by **method overloading** or **operator overloading**.
- Faster but less flexible.

📖 Runtime Polymorphism (Dynamic Binding)

- Decided at **runtime**.
- Achieved by **method overriding**.
- More flexible but slightly slower.

🔹 Example (Java)

```
// Compile-time Polymorphism (Overloading)
class Calculator {
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
}

// Runtime Polymorphism (Overriding)
class Animal {
    void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
    void sound() { System.out.println("Dog barks"); }
}

public class Main {
    public static void main(String[] args) {
        Calculator c = new Calculator();
        System.out.println(c.add(5, 10));     // 15
```

```
        System.out.println(c.add(5.5, 4.5));  // 10.0

        Animal a = new Dog();
        a.sound();  // Dog barks (decided at runtime)
    }
}
```

🔹 **Real-world Analogy**

- **Compile-time** → Knowing which train ticket you'll buy before leaving home.
- **Runtime** → Choosing which train to board after reaching station.

🎯 **Interview Tip**

Say:
"Compile-time polymorphism is achieved via method overloading, Runtime polymorphism via overriding. Overloading = early binding, Overriding = late binding."

🔹 **Q21. Difference between Static Methods and Instance Methods**

📖 **Static Methods**

- Belong to the **class** rather than an object.
- Can be called **without creating an object**.
- Can access only **static variables/methods** directly.
- Useful for utility/helper methods.

📖 **Instance Methods**

- Belong to an **object** of a class.
- Require object creation to call.
- Can access both **instance variables** and **static variables**.

🔹 **Example (Java)**

```
class MathUtils {
    static int square(int x) {  // static method
```

```
        return x * x;
    }
    int cube(int x) {          // instance method
        return x * x * x;
    }
}
public class Main {
    public static void main(String[] args) {
        // static method → no object needed
        System.out.println(MathUtils.square(5));  // 25

        // instance method → needs object
        MathUtils m = new MathUtils();
        System.out.println(m.cube(3));  // 27
    }
}
```

🔹 **Real-world Analogy**

- **Static method** → A general formula (like Pythagoras theorem), can be used by anyone without ownership.
- **Instance method** → Specific action of an individual object (Car object starts its engine).

🎯 **Interview Tip**

Say:
"Static methods belong to class, instance methods belong to object. Static = utility, Instance = behavior of object."

🔹 **Q22. What is Method Hiding in OOP?**

📖 **Definition**

- When a **subclass defines a static method with the same name and signature as in the parent class**, it is **method hiding**.
- Static methods are **not overridden**, they are hidden.
- Method resolution depends on **reference type (compile-time)**, not object type.

🔹 **Example (Java)**

```
class Parent {
    static void show() {
        System.out.println("Parent static show");
    }
}
class Child extends Parent {
    static void show() {
        System.out.println("Child static show");
    }
}
public class Main {
    public static void main(String[] args) {
        Parent p = new Child();
        p.show();  // Output: Parent static show (method hiding)
    }
}
```

🔹 **Real-world Analogy**

- Like a **notice board**: if parent puts one notice and child puts another, which one you see depends on **which notice board you look at**, not who pinned it.

🎯 **Interview Tip**

Say:
"Overriding works with instance methods at runtime. Static methods cannot be overridden, they are hidden, and resolution happens at compile time."

🔹 **Q23. Can a Constructor be Private? Why/When do we use it?**

📖 **Answer**

Yes ✅, a constructor can be declared `private`.

📖 **Why/When?**

1. **Singleton Pattern** → Ensure only one instance of a class is created.
2. **Factory Methods** → Restrict object creation to controlled methods.
3. **Utility Classes** → Prevent instantiation of classes containing only static methods.

🔹 **Example (Java – Singleton)**

```
class Singleton {
    private static Singleton instance;  // single instance
    private Singleton() {}              // private constructor

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
public class Main {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println(s1 == s2);  // true
    }
}
```

🔹 **Real-world Analogy**

- **Private constructor** → Like a **passport office** → you can't print your own passport; only the office can create it under controlled conditions.

🎯 **Interview Tip**

Say:
"Yes, constructors can be private. We use it in Singleton design pattern, factory methods, or utility classes to control object creation."

🔹 **Q24. What is the difference between Shallow Copy and Deep Copy of an object?**

📖 **Shallow Copy**

- Copies only the **references** of objects, not the actual nested objects.
- Changes in nested objects reflect in both copies.

📖 **Deep Copy**

- Copies the object along with all its **nested objects**.

- Both objects are fully independent.

🔹 **Example (Java)**

```
class Address {
    String city;
    Address(String city) { this.city = city; }
}
class Student implements Cloneable {
    String name;
    Address addr;
    Student(String name, Address addr) {
        this.name = name;
        this.addr = addr;
    }
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();  // shallow copy
    }
}
public class Main {
    public static void main(String[] args) throws Exception {
        Address addr = new Address("Delhi");
        Student s1 = new Student("Alice", addr);
        Student s2 = (Student) s1.clone();  // shallow copy

        s2.addr.city = "Mumbai";
        System.out.println(s1.addr.city);  // Output: Mumbai (changed in both!)
    }
}
```

For **deep copy**, we'd clone `addr` separately so changes don't affect the original.

🔹 **Real-world Analogy**

- **Shallow copy** → Photocopy of a form where attached documents (nested objects) are **not duplicated**, just referenced.

- **Deep copy** → Photocopy including all attached documents.

🎯 **Interview Tip**

Say:
"Shallow copy copies references, deep copy duplicates everything including nested objects. Deep copy ensures independence, shallow doesn't."

🔹 Q25. What is Object Cloning?

📖 Definition

- **Cloning** = Creating an **exact copy of an object** with the same state (data).
- In Java, achieved using the `clone()` method of the `Object` class.

📖 Requirements

- Class must implement `Cloneable` interface.
- Override `clone()` method.

🔹 Example (Java)

```
class Student implements Cloneable {
    String name;
    Student(String name) { this.name = name; }

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
public class Main {
    public static void main(String[] args) throws Exception {
        Student s1 = new Student("Alice");
        Student s2 = (Student) s1.clone();
        System.out.println(s1.name); // Alice
        System.out.println(s2.name); // Alice
        System.out.println(s1 == s2); // false (different objects)
    }
}
```

🔹 Real-world Analogy

- Like creating a **duplicate key** → looks the same, works the same, but is a separate physical entity.

🎯 Interview Tip

Say:
"Cloning creates a new object with the same values as the existing one. In Java, we use `clone()` after implementing `Cloneable`."

🔹 Q26. Difference between `equals()` and `==` operator

📖 `==` Operator

- Compares **references (memory addresses)** in case of objects.
- For primitives, compares **actual values**.

📖 `equals()` Method

- Defined in `Object` class, but often overridden (e.g., in `String`).
- Compares **contents/values** of objects (logical equality).

🔹 Example (Java)

```
public class Main {
   public static void main(String[] args) {
      String s1 = new String("Hello");
      String s2 = new String("Hello");

      System.out.println(s1 == s2);       // false (different objects)
      System.out.println(s1.equals(s2));  // true  (same content)
   }
}
```

🔹 Real-world Analogy

- **==** → Two books look identical but you check their barcodes (reference).
- **equals()** → You read the content inside and check if they match.

🎯 Interview Tip

Say:
"`==` compares reference (for objects), `equals()` compares content. For primitives, `==` compares value."

🔹 Q27. What is an Immutable Class? Example?

📖 Definition

- An **immutable class** is a class whose objects **cannot be modified** after creation.
- Any change creates a **new object**.

📖 Example in Java

- `String` is **immutable**.

🔹 How to Create an Immutable Class?

1. Declare class as `final` (optional, but prevents inheritance).
2. Make all fields `private` and `final`.
3. No setters.
4. Provide getters but return copies of mutable fields.

🔹 Example (Custom Immutable Class)

```
final class Student {
   private final String name;
   private final int age;

   public Student(String name, int age) {
      this.name = name;
      this.age = age;
   }
   public String getName() { return name; }
   public int getAge() { return age; }
}
```

🔹 Real-world Analogy

- Like a **printed book** → Once published, its content cannot be changed. If you want a new version, you print another copy.

🎯 Interview Tip

Say:
"Immutable objects cannot change once created. Example: String in Java. They help in thread safety and caching."

🔹 Q28. What are Inner Classes? Types?

📖 Definition

- An **inner class** is a class defined **inside another class**.
- Useful for logically grouping classes and increasing encapsulation.

📖 Types of Inner Classes in Java

1. **Non-static nested class (inner class)**
2. **Static nested class**
3. **Method-local inner class**
4. **Anonymous inner class**

🔹 Example (Java)

```
class Outer {
    private String msg = "Hello";

    class Inner {   // Non-static inner class
        void printMsg() {
            System.out.println(msg); // can access outer private members
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Outer o = new Outer();
        Outer.Inner i = o.new Inner();  // object creation
        i.printMsg();  // Hello
    }
}
```

🔹 Real-world Analogy

- Like a **department inside a company** → Inner classes exist only inside their outer class.

🎯 Interview Tip

Say:
"Inner classes are nested inside outer classes. They improve encapsulation and are useful for event handling (anonymous classes)."

🔹 **Q29. What is Early Binding and Late Binding?**

📖 **Early Binding (Static Binding)**

- Method call resolved at **compile time**.
- Example: Method overloading, static methods, private methods, final methods.

📖 **Late Binding (Dynamic Binding)**

- Method call resolved at **runtime**.
- Example: Method overriding.

🔹 **Example (Java)**

```
class Animal {
   void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
   void sound() { System.out.println("Dog barks"); }
}
public class Main {
   public static void main(String[] args) {
      Animal a = new Dog();
      a.sound();  // Late binding → resolved at runtime (Dog barks)
   }
}
```

🔹 **Real-world Analogy**

- **Early binding** → Deciding your exam center before exam day (fixed in advance).
- **Late binding** → Deciding which cab to take after reaching the station (runtime decision).

🎯 **Interview Tip**

Say:
"Early binding = compile-time (overloading, static methods), Late binding = runtime (overriding)."

🔹 **Q30. Can we override static methods? Why/Why not?**

📖 **Answer**

- ❌ No, static methods **cannot be overridden** in Java.
- They are **class-level methods**, resolved at **compile-time**.
- If a subclass defines a static method with the same signature → It's **method hiding**, not overriding.

🔹 **Example (Java)**

```
class Parent {
    static void greet() {
        System.out.println("Hello from Parent");
    }
}
class Child extends Parent {
    static void greet() {
        System.out.println("Hello from Child");
    }
}
public class Main {
    public static void main(String[] args) {
        Parent p = new Child();
        p.greet(); // Output: Hello from Parent (not overridden, hidden)
    }
}
```

🔹 **Real-world Analogy**

- Like a **family surname**: If the parent has it, the child has it too, but you don't "override" it — you just **hide it under your own identity**.

🎯 **Interview Tip**

Say:
"Static methods cannot be overridden, only hidden. Overriding applies only to instance methods (runtime polymorphism)."

🔹 **Q31. Difference between `final`, `finally`, and `finalize()`**

📖 **`final` (keyword)**

- Used with **variables, methods, classes**.
- Variable → constant (value can't change).
- Method → cannot be overridden.
- Class → cannot be inherited.

📖 **`finally` (block)**

- Used in **exception handling**.
- Always executes (whether exception occurs or not).
- Commonly used for cleanup (closing files, releasing resources).

📖 **`finalize()` (method)**

- A method of `Object` class.
- Called by **Garbage Collector (GC)** before destroying an object.
- Rarely used in modern Java (not reliable).

🔹 **Example**

```
class Demo {
    final int x = 10;  // final variable

    final void display() {  // final method
        System.out.println("Final method");
    }

    @Override
    protected void finalize() {
        System.out.println("Finalize called");
    }
}
public class Main {
    public static void main(String[] args) {
        try {
```

```
            int a = 5 / 0;
        } catch (Exception e) {
            System.out.println("Exception");
        } finally {
            System.out.println("Finally block always executes");
        }
    }
}
```

🔹 **Real-world Analogy**

- **final** → A “Do Not Change” label.
- **finally** → A “Cleanup after use” instruction.
- **finalize** → Garbage collector’s “last rites” before disposing.

🎯 **Interview Tip**

Say:
“`final` is a keyword, `finally` is a block, and `finalize()` is a method of `Object` class. Each serves a completely different purpose.”

🔹 **Q32. How does Garbage Collection work in OOP?**

📖 **Garbage Collection (GC)**

- Automatic memory management in Java.
- Frees memory occupied by **unreachable objects** (no active references).
- Prevents memory leaks.

📖 **How GC Works**

1. Finds objects not referenced by any variable.
2. Marks them as “eligible for collection.”
3. Deletes them and reclaims memory.

🔹 **Example**

```
class Demo {
    @Override
    protected void finalize() {
        System.out.println("Object destroyed by GC");
    }
}
public class Main {
    public static void main(String[] args) {
        Demo d = new Demo();
        d = null;              // object becomes unreachable
        System.gc();           // request garbage collection
    }
}
```

🔹 **Real-world Analogy**

- Like a **housemaid cleaning unused items**. If no one is using an object, GC sweeps it away.

🎯 **Interview Tip**

Say:
"GC in Java automatically removes unused objects to free memory. We can suggest GC using `System.gc()`, but it's not guaranteed."

🔹 **Q33. Difference between Abstract Class and Interface**

📖 **Abstract Class**

- Can have **abstract** and **concrete** methods.
- Can have **constructors, fields (with any access modifiers)**.
- Supports **single inheritance**.

📖 **Interface**

- Only method declarations (till Java 7).
- From Java 8 → can have `default` and `static` methods.
- Fields → always `public static final`.

- Supports **multiple inheritance**.

🔹 **Example**

```
abstract class Animal {
   abstract void sound();      // abstract method
   void sleep() { System.out.println("Sleeping..."); } // concrete method
}

interface Pet {
   void play();               // by default public abstract
}

class Dog extends Animal implements Pet {
   void sound() { System.out.println("Bark"); }
   public void play() { System.out.println("Plays fetch"); }
}
```

🔹 **Real-world Analogy**

- **Abstract class** → Blueprint with some ready-made parts.
- **Interface** → 100% contract that must be followed.

🎯 **Interview Tip**

Say:
"Abstract classes are partial blueprints, interfaces are full contracts. Interfaces allow multiple inheritance, abstract classes don't."

🔹 **Q34. What is a Marker Interface? Example?**

📖 **Definition**

- An **interface with no methods or fields**.
- Used to "mark" a class to give it special meaning to JVM or frameworks.

📖 **Examples in Java**

- `Serializable`

- `Cloneable`
- `Remote`

🔹 **Example**

```
import java.io.Serializable;

class Student implements Serializable {
   String name;
   Student(String name) { this.name = name; }
}
```

🔹 **Why Marker Interface?**

- Tells JVM/compiler: “This class has special capability.”
- e.g., `Serializable` tells JVM that object can be converted to a byte stream.

🔹 **Real-world Analogy**

- Like **stickers on a parcel** → “Fragile”, “Handle with Care”. Stickers don’t do anything but give **special meaning**.

🎯 **Interview Tip**

Say:
“Marker interface has no methods. It just marks a class to signal JVM or frameworks (e.g., Serializable, Cloneable).”

🔹 **Q35. What is the difference between `transient` and `volatile`?**

📖 **`transient`**

- Used in **serialization**.
- Prevents a field from being serialized (ignored during object saving).

📖 **`volatile`**

- Used in **multithreading**.
- Ensures variable is always read from **main memory**, not from thread's local cache.
- Provides **visibility guarantee** (not atomicity).

🔹 **Example**

```
class Student implements java.io.Serializable {
   String name;
   transient int age; // will not be serialized
}

class SharedResource {
   volatile boolean flag = false;
}
```

🔹 **Real-world Analogy**

- **transient** → Like a personal note you don't include when photocopying a document.
- **volatile** → Like a **shared notice board** where all updates are immediately visible to everyone.

🎯 **Interview Tip**

Say:
"`transient` is for serialization (skip field), `volatile` is for concurrency (visibility across threads)."

🔹 **Q36. What is Multiple Inheritance? Why not supported in Java?**

📖 **Definition**

- **Multiple inheritance** → A class inherits from **more than one parent class**.
- Supported in C++, but **not directly in Java**.

📖 **Why not in Java? (Problem)**

- Causes **Diamond Problem**: If two parent classes have the same method, child doesn't know which one to inherit.

🔹 **Example (C++)**

```
class A {
public:
    void show() { cout << "A"; }
};
class B {
public:
    void show() { cout << "B"; }
};
class C : public A, public B { };  // multiple inheritance
int main() {
    C obj;
    obj.show(); // ERROR: Ambiguity (A::show or B::show?)
}
```

🔹 **How Java Solves It**

- Java allows **multiple inheritance using interfaces**, since interfaces don't carry method implementation (till Java 7).
- From Java 8, default methods in interfaces are resolved using explicit overriding.

🔹 **Real-world Analogy**

- Like a child having **two fathers giving conflicting instructions** → Confusion.

🎯 **Interview Tip**

Say:
"Java avoids multiple inheritance with classes to prevent ambiguity (Diamond Problem). Instead, Java uses interfaces to achieve it safely."

🔹 **Q37. Difference between Overloading Operators and Method Overloading**

📖 **Method Overloading**

- Same method name, different parameter list (compile-time polymorphism).
- Supported in Java.

📖 **Operator Overloading**

- Redefining how operators (`+`, `-`, `==`) work for user-defined types.
- Supported in C++ (not in Java).

🔹 **Example (C++) → Operator Overloading**

```
class Complex {
   int real, imag;
public:
   Complex(int r, int i) : real(r), imag(i) {}
   Complex operator+(Complex c) {  // operator overloading
      return Complex(real + c.real, imag + c.imag);
   }
};
```

🔹 **Example (Java) → Method Overloading**

```
class MathUtil {
   int add(int a, int b) { return a + b; }
   double add(double a, double b) { return a + b; }
}
```

🔹 **Real-world Analogy**

- **Method Overloading** → Same button but different modes (short press = flashlight, long press = camera).
- **Operator Overloading** → Giving new meaning to an existing symbol (`+` for strings in Java = concatenation).

🎯 **Interview Tip**

Say:
"Java supports method overloading, but not operator overloading (except `+` for strings). C++ supports both."

🔹 **Q38. What is the role of the `super` keyword in OOP?**

📖 **`super` Keyword**

- Refers to **immediate parent class**.

- Used for:
    1. Calling parent class **constructor**.
    2. Accessing parent class **methods/variables** when overridden.

🔹 **Example**

```
class Parent {
    int value = 100;
    Parent() { System.out.println("Parent constructor"); }
    void display() { System.out.println("Parent method"); }
}
class Child extends Parent {
    int value = 200;
    Child() {
        super(); // calls Parent constructor
        System.out.println("Child constructor");
    }
    void display() {
        super.display(); // calls Parent method
        System.out.println("Child method");
    }
}
public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.display();
        System.out.println(c.value);       // 200
        System.out.println(((Parent)c).value); // 100
    }
}
```

🔹 **Real-world Analogy**

- Like a child saying **“Ask my parent first”**.

🎯 **Interview Tip**

Say:
“`super` is used to call parent’s constructor, method, or field when child overrides or hides them.”

🔹 **Q39. Difference between Abstract Class vs Interface vs Concrete Class**

📖 **Concrete Class**

- A normal class → has full implementation.
- Objects can be created.

📖 **Abstract Class**

- Partial implementation.
- Can have abstract + non-abstract methods.
- Cannot be instantiated.

📖 **Interface**

- Pure contract (till Java 7).
- Only abstract methods (Java 8+ → default & static methods allowed).
- Supports multiple inheritance.

🔹 **Quick Comparison**

Feature	Concrete Class	Abstract Class	Interface
Object Creation	✅ Yes	❌ No	❌ No
Method Type	Concrete only	Abstract + Concrete	Abstract (default/static since Java 8)
Constructors	✅ Yes	✅ Yes	❌ No
Multiple Inheritance	❌ No	❌ No	✅ Yes

🔹 **Real-world Analogy**

- **Concrete class** → Fully built house (ready to use).
- **Abstract class** → House blueprint + some pre-built rooms.

- **Interface** → Just a blueprint (no implementation).

🎯 Interview Tip

Say:
"Concrete = full implementation, Abstract = partial implementation, Interface = 100% contract. Interfaces allow multiple inheritance."

🔹 Q40. Difference between Association and Dependency

📖 Association

- A **relationship** between two classes.
- Can be one-to-one, one-to-many, many-to-many.
- Both can exist **independently**.

📖 Dependency

- A **"uses-a" relationship**.
- One class depends on another for a method or functionality.
- If dependency class changes, dependent class may break.

🔹 Example (Java)

```
class Engine { }
class Car {     // Association (Car has Engine)
   Engine engine;
}

// Dependency
class Driver {
   void drive(Car c) {    // Driver depends on Car
      System.out.println("Driving car");
   }
}
```

🔹 Real-world Analogy

- **Association** → A teacher works in a school, but both can exist independently.
- **Dependency** → A person depends on electricity to use a computer.

🎯 **Interview Tip**

Say:
"Association = structural relationship ('has-a'). Dependency = behavioral ('uses-a'). Dependency is weaker and more temporary."

Perfect 👍 Let's move on to **Q41–Q45** with the same **structured breakdown (definition → why → example → analogy → interview tip)**.

🔹 **Q41. What is Object Slicing?**

📖 **Definition**

- **Object slicing** happens when a derived class object is assigned to a base class variable by value.
- The **extra attributes of the derived class get "sliced off."**

👉 Happens in **C++** (not in Java, because Java always uses references).

🔹 **Example (C++)**

```
#include <iostream>

using namespace std;


class Base {

public:

    int x;

};
```

```
class Derived : public Base {
public:
    int y;
};

int main() {
    Derived d;
    d.x = 10;
    d.y = 20;

    Base b = d;  // object slicing
    cout << b.x << endl;  // ✅ prints 10
    // cout << b.y << endl; ❌ ERROR (y sliced off)
}
```

🔹 **Real-world Analogy**

- Imagine copying only the **parent's photo** from a family photo → child's extra details are lost.

🎯 **Interview Tip**

Say:
"Object slicing occurs when a derived object is assigned to a base object by value, causing derived-specific data to be lost."

🔹 **Q42. What is Runtime Type Identification (RTTI)?**

📖 **Definition**

- **RTTI** allows determination of an object's type at runtime.

- Used with **polymorphism** when we have a base-class reference pointing to a derived object.

📖 **In C++**

- Uses `dynamic_cast` and `typeid`.

📖 **In Java**

- Uses `instanceof` and reflection.

🔹 **Example (Java)**

```
class Animal { }

class Dog extends Animal { }


public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();


        if (a instanceof Dog) {
            System.out.println("a is a Dog");
        }
    }
}
```

🔹 **Real-world Analogy**

- Like asking **“What breed of animal is this?”** after you see it.

🎯 **Interview Tip**

Say:
"RTTI checks the actual object type at runtime, not the reference type. In Java, `instanceof` is used; in C++, `dynamic_cast` or `typeid`."

🔹 **Q43. What is Covariance and Contravariance in OOP?**

📖 **Covariance**

- A method can return a **subclass type** instead of the parent type when overriding.
- In Java, **return types are covariant** since Java 5.

📖 **Contravariance**

- Parameter types can be more general (superclass type) than in the overridden version.
- Java **does not allow contravariant parameters in overriding**.

🔹 **Example (Java - Covariant return)**

```
class Animal {}

class Dog extends Animal {}


class AnimalShelter {

    Animal getAnimal() { return new Animal(); }

}

class DogShelter extends AnimalShelter {

    @Override

    Dog getAnimal() {   // Covariant return type

        return new Dog();

    }

}
```

🔹 **Real-world Analogy**

- **Covariance** → Child returning a more specific gift than the parent promised.
- **Contravariance** → Accepting broader inputs (not allowed in Java for methods).

🎯 **Interview Tip**

Say:
"Covariance allows overriding methods to return a more specific type. Contravariance is the opposite but not supported in Java overriding."

🔹 **Q44. Explain the Liskov Substitution Principle (LSP) with an example.**

📖 **Definition**

- One of the **SOLID principles**.
- **If S is a subclass of T, then objects of type T should be replaceable with objects of type S without breaking the program.**

📖 **Problem Example**

```
class Bird {
    void fly() { System.out.println("Flying"); }
}
class Penguin extends Bird {
    @Override
    void fly() { throw new UnsupportedOperationException("Penguins can't fly"); }
}
```

👉 Violates LSP, because substituting `Penguin` for `Bird` breaks functionality.

📖 **Correct Design**

```
interface Bird { }
interface FlyingBird extends Bird {
```

```
    void fly();
}
class Sparrow implements FlyingBird {
    public void fly() { System.out.println("Flying high"); }
}
class Penguin implements Bird {
    void swim() { System.out.println("Swimming"); }
}
```

🔹 **Real-world Analogy**

- If all cars inherit from `Car`, then an `ElectricCar` must still behave like a car (drive), not break the assumption.

🎯 **Interview Tip**

Say:
"LSP means subclasses should be usable anywhere the parent class is expected, without altering correctness. If not, it violates substitution."

🔹 **Q45. What is Object Persistence?**

📖 **Definition**

- **Object persistence** = ability of an object to **outlive the program execution**.
- Object state is stored permanently (in database/file) and later restored.

📖 **Techniques**

- Serialization (Java).
- Storing in relational database (ORM tools like Hibernate).
- File storage.

🔹 **Example (Java - Serialization)**

```
import java.io.*;


class Student implements Serializable {
    String name;
    int age;
    Student(String n, int a) { name = n; age = a; }
}


public class Main {
    public static void main(String[] args) throws Exception {
        Student s1 = new Student("John", 22);


        // Save object to file
        ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("student.ser"));
        out.writeObject(s1);
        out.close();


        // Restore object
        ObjectInputStream in = new ObjectInputStream(new FileInputStream("student.ser"));
        Student s2 = (Student) in.readObject();
        System.out.println(s2.name + " " + s2.age);
    }
}
```

🔹 **Real-world Analogy**

- Like **saving a game** → you can quit and resume later without losing progress.

🎯 Interview Tip

Say:
"Object persistence means object state survives beyond program execution, usually via serialization or databases."

🔹 Q46. What is Immutability in OOP?

📖 Definition

- An **immutable object** is one whose **state cannot be changed** after it is created.
- Common in functional programming and safe for multi-threading.

📖 In Java

- `String`, `Integer`, `LocalDate` are immutable.
- Once created, modification creates a **new object**.

🔹 Example

```
public class Main {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = s1.concat(" World");

        System.out.println(s1); // "Hello" (unchanged)
        System.out.println(s2); // "Hello World"
    }
}
```

🔹 How to Create Custom Immutable Class

1. Make class `final`.
2. Declare all fields `private final`.
3. No setters, only getters.
4. Return deep copies for mutable fields.

```
final class Student {
    private final String name;
    private final int age;

    Student(String n, int a) {
        name = n;
        age = a;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
}
```

🔹 **Real-world Analogy**

- Like a **birth certificate** → once issued, cannot be changed.

🎯 **Interview Tip**

Say:
"Immutable objects cannot be changed once created. They improve thread-safety and caching. Example: `String` in Java."

🔹 **Q47. What is Reflection in OOP?**

📖 **Definition**

- **Reflection** = ability of a program to **inspect and modify structure/behavior of objects at runtime**.

- Java provides `java.lang.reflect` package.

📖 **Uses**

- Debugging, testing frameworks (JUnit), dependency injection (Spring), serialization, ORMs (Hibernate).

🔹 **Example (Java)**

```
import java.lang.reflect.*;


class Student {

    private String name = "John";

}


public class Main {

    public static void main(String[] args) throws Exception {

        Student s = new Student();

        Field field = Student.class.getDeclaredField("name");

        field.setAccessible(true);  // bypass private

        System.out.println("Value: " + field.get(s)); // prints John

    }

}
```

🔹 **Real-world Analogy**

- Like an **X-ray machine** → lets you see internal details of a person without cutting them open.

🎯 **Interview Tip**

Say:
"Reflection lets a program analyze or modify itself at runtime. It powers frameworks like Spring, Hibernate, and JUnit."

🔹 **Q48. What is Duck Typing? Is it supported in Java?**

📖 **Definition**

- **Duck typing**: An object's suitability is determined by the presence of certain methods, **not by inheritance hierarchy or type**.
- Term: *"If it walks like a duck and quacks like a duck, it's a duck."*

📖 **Example (Python)**

```
class Duck:
    def quack(self):
        print("Quack")


class Person:
    def quack(self):
        print("I can quack too!")


def make_it_quack(obj):
    obj.quack()


make_it_quack(Duck())     # Quack
make_it_quack(Person())   # I can quack too!
```

📖 **In Java**

- Java is **statically typed**, so duck typing is **not directly supported**.

- But **interfaces** and **reflection** can mimic it.

🔹 **Real-world Analogy**

- If someone behaves like a doctor, talks like a doctor, and treats like a doctor → you assume they are a doctor (even without degree proof).

🎯 **Interview Tip**

Say:
"Duck typing is common in dynamically typed languages like Python. Java does not directly support it, but interfaces provide similar flexibility."

🔹 **Q49. Difference between Factory Method and Abstract Factory Pattern**

📖 **Factory Method Pattern**

- Defines an **interface** for creating an object, but lets subclasses decide which class to instantiate.
- Provides **single product family**.

📖 **Abstract Factory Pattern**

- Provides an **interface for creating families of related objects** without specifying their concrete classes.
- Used when we need multiple product families.

🔹 **Example**

Factory Method (Java)

```
abstract class ShapeFactory {
    abstract Shape createShape();
}
class CircleFactory extends ShapeFactory {
    Shape createShape() { return new Circle(); }
```

```
}
```

Abstract Factory (Java)

```
interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}
class WindowsFactory implements GUIFactory {
    public Button createButton() { return new WindowsButton(); }
    public Checkbox createCheckbox() { return new WindowsCheckbox(); }
}
```

🔹 **Real-world Analogy**

- **Factory Method** → Coffee shop with a menu item (decides which coffee to prepare).
- **Abstract Factory** → Starbucks franchise that supplies **entire sets** (coffee + snack + cup design).

🎯 **Interview Tip**

Say:
"Factory Method = one product, Abstract Factory = families of related products. Abstract Factory is a factory of factories."

🔹 **Q50. What is the Builder Pattern?**

📖 **Definition**

- **Builder Pattern** = used to construct **complex objects step-by-step**.
- Allows creation of immutable objects with many optional parameters.

📖 **Why?**

- Avoids **telescoping constructors** (constructors with many parameters).

🔹 **Example (Java)**

```
class Student {
    private final String name;
    private final int age;
    private final String address;

    private Student(Builder b) {
        this.name = b.name;
        this.age = b.age;
        this.address = b.address;
    }

    static class Builder {
        private String name;
        private int age;
        private String address;

        Builder setName(String name) { this.name = name; return this; }
        Builder setAge(int age) { this.age = age; return this; }
        Builder setAddress(String addr) { this.address = addr; return this; }

        Student build() { return new Student(this); }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Student s = new Student.Builder()
                        .setName("John")
                        .setAge(22)
                        .setAddress("NY")
                        .build();
    }
}
```

🔹 **Real-world Analogy**

- Like **ordering a pizza** → you add toppings step by step, and finally build the pizza.

🎯 **Interview Tip**

Say:
"Builder Pattern helps in constructing complex objects step by step, especially when many optional parameters are involved."

🔹 **Q51. What is the Adapter Pattern?**

📖 **Definition**

- **Adapter Pattern** allows two incompatible interfaces to work together.
- It **wraps an existing class** with a new interface so clients can use it.

📖 **Why?**

- Useful when integrating **legacy code** with new systems.

🔹 **Example (Java)**

```
// Old interface
class OldPrinter {
```

```
    void printOld(String text) {
        System.out.println("Old Print: " + text);
    }
}

// New interface expected
interface Printer {
    void print(String text);
}

// Adapter
class PrinterAdapter implements Printer {
    private OldPrinter oldPrinter = new OldPrinter();
    public void print(String text) {
        oldPrinter.printOld(text); // adapts old to new
    }
}

public class Main {
    public static void main(String[] args) {
        Printer printer = new PrinterAdapter();
        printer.print("Hello Adapter");
    }
}
```

- **Analogy**

- A **travel plug adapter** → lets your US plug work in an EU socket.

🎯 Interview Tip

Say:
"Adapter Pattern acts as a bridge between incompatible interfaces. Example: converting old API calls to new ones."

🔹 Q52. What is the Prototype Pattern?

📖 Definition

- **Prototype Pattern** allows creating new objects by **cloning an existing object** instead of creating from scratch.

📖 Why?

- Useful when object creation is **costly** (e.g., database connection, network-heavy).

🔹 Example (Java)

```
class Student implements Cloneable {
    String name;
    int age;

    Student(String n, int a) { name = n; age = a; }

    public Student clone() throws CloneNotSupportedException {
        return (Student) super.clone();
    }
}

public class Main {
```

```
    public static void main(String[] args) throws Exception {
        Student s1 = new Student("John", 22);
        Student s2 = s1.clone();
        System.out.println(s2.name + " " + s2.age);
    }
}
```

🔹 **Analogy**

- **Photocopy machine** → instead of rewriting, you clone an existing paper.

🎯 **Interview Tip**

Say:
"Prototype pattern avoids expensive object creation by cloning existing ones."

🔹 **Q53. Difference between Strategy and State Pattern**

📖 **Strategy Pattern**

- Defines a family of algorithms and makes them **interchangeable**.
- Used when **behavior** changes at runtime, chosen by client.

📖 **State Pattern**

- Allows an object to **alter behavior when internal state changes**.
- The object **appears to change class**.

🔹 **Example**

Strategy (Java):

```
interface Payment {
    void pay(int amount);
```

```
}
class CreditCardPayment implements Payment {
    public void pay(int amount) { System.out.println("Paid by card: " + amount); }
}
class PayPalPayment implements Payment {
    public void pay(int amount) { System.out.println("Paid by PayPal: " + amount); }
}
```

👉 Client chooses payment strategy.

State (Java):

```
interface State {
    void handle();
}
class HappyState implements State {
    public void handle() { System.out.println("I am happy"); }
}
class SadState implements State {
    public void handle() { System.out.println("I am sad"); }
}
```

👉 Behavior depends on object's current state.

🔹 **Analogy**

- **Strategy** = choosing different travel routes.
- **State** = mood changes depending on situation.

🎯 **Interview Tip**

Say:
"Strategy = external choice of behavior. State = behavior changes internally based on state."

🔹 **Q54. What is Dependency Injection (DI) in OOP?**

📖 **Definition**

- **Dependency Injection** is a technique where an object's dependencies are **provided from outside** instead of being created inside.

📖 **Why?**

- Improves **testability, maintainability, flexibility**.
- Key principle in **Spring framework**.

🔹 **Example (Java)**

❌ Without DI:

```
class Engine {}

class Car {
    private Engine engine = new Engine(); // tightly coupled
}
```

✅ With DI:

```
class Engine {}

class Car {
    private Engine engine;
    public Car(Engine e) { this.engine = e; } // injected
}
```

🔹 **Analogy**

- Like hiring a **driver** instead of learning to drive yourself. You inject dependency instead of creating it.

🎯 Interview Tip

Say:
"Dependency Injection means providing an object's dependencies externally. It promotes loose coupling and is heavily used in frameworks like Spring."

🔹 Q55. How does Garbage Collection work in OOP?

📖 Definition

- **Garbage Collection (GC)** = automatic process of reclaiming memory occupied by unused objects.
- Used in **Java, .NET, Python**.

📖 How?

- GC identifies objects with **no live references** and removes them.
- In Java → works with **Generational GC** (Young Gen, Old Gen).

🔹 Example (Java)

```
public class Main {
    public static void main(String[] args) {
        String s = new String("Hello");
        s = null;  // object eligible for GC
        System.gc();  // suggest GC
    }
}
```

🔹 Analogy

- Like a **cleaner in your house** → removes unused items automatically.

🎯 Interview Tip

Say:
"Garbage Collection automatically frees memory by destroying unreachable objects. In Java, `System.gc()` requests GC, but JVM decides actual timing."

🔹 Q56. What is Object Pool Pattern?

📖 Definition

- **Object Pool Pattern** reuses a fixed set of objects instead of creating and destroying them frequently.

📖 Why?

- Useful for costly objects (e.g., database connections).

🔹 Example (Conceptual)

- Database **connection pool** in JDBC.

```
class ConnectionPool {
   private List<Connection> availableConnections;
   // return connections from pool instead of new()
}
```

🔹 Analogy

- Like **car rentals** → instead of buying new cars for each trip, reuse existing ones.

🎯 Interview Tip

Say:
"Object Pool manages a set of reusable objects to reduce creation cost. Common in JDBC connection pools."

🔹 **Q57. What are Weak References in Java?**

📖 **Definition**

- A **weak reference** does not prevent an object from being collected by GC.
- In Java → `WeakReference<T>` class.

📖 **Why?**

- Useful for **caches** where objects can be garbage collected when memory is low.

🔹 **Example**

```
import java.lang.ref.WeakReference;


class Student {
    String name;
    Student(String n) { name = n; }
}


public class Main {
    public static void main(String[] args) {
        Student s = new Student("John");
        WeakReference<Student> weakRef = new WeakReference<>(s);


        s = null; // now only weak reference exists
        System.gc();
    }
}
```

🔹 **Analogy**

- Like a **sticky note** attached lightly → can fall off anytime.

🎯 **Interview Tip**

Say:
"Weak references allow GC to collect objects even if referenced, commonly used in caches."

🔹 **Q58. Explain Strong, Weak, Soft, and Phantom References in Java.**

📖 **Strong Reference**

- Normal reference → prevents GC.

📖 **Weak Reference**

- Object is eligible for GC if only weak references exist.

📖 **Soft Reference**

- Collected only when **memory is low**. Used in caches.

📖 **Phantom Reference**

- Object already finalized → phantom used for cleanup tracking.

🔹 **Analogy**

- **Strong = tightly holding hand.**
- **Soft = loosely holding, but still present.**
- **Weak = barely touching, let go easily.**
- **Phantom = ghost reference after death.**

🎯 **Interview Tip**

Say:
"Java has 4 reference types. Strong prevents GC, soft helps with caching, weak allows easy GC, phantom used for cleanup."

🔹 **Q59. What is Double Dispatch in OOP?**

📖 **Definition**

- **Double dispatch** allows method execution to depend on **two objects' runtime types** instead of one.

📖 **Why?**

- Normal polymorphism → single dispatch (depends only on receiver).
- Double dispatch → depends on **both objects**.

🔹 **Example (Java)**

```
interface Shape {
    void collideWith(Shape s);
    void collideWithCircle(Circle c);
    void collideWithRectangle(Rectangle r);
}
class Circle implements Shape {
    public void collideWith(Shape s) { s.collideWithCircle(this); }
    public void collideWithCircle(Circle c) { System.out.println("Circle hits Circle"); }
    public void collideWithRectangle(Rectangle r) { System.out.println("Circle hits Rectangle");
}
}
class Rectangle implements Shape {
    public void collideWith(Shape s) { s.collideWithRectangle(this); }
    public void collideWithCircle(Circle c) { System.out.println("Rectangle hits Circle"); }
```

```
    public void collideWithRectangle(Rectangle r) { System.out.println("Rectangle hits Rectangle"); }

}
```

🔹 **Analogy**

- Like **two people greeting** → action depends on both (if teacher meets student = bow, if friends meet = handshake).

🎯 **Interview Tip**

Say:
"Double dispatch ensures method depends on two object types at runtime. Common in Visitor Pattern."

🔹 **Q60. What is the Visitor Pattern?**

📖 **Definition**

- **Visitor Pattern** allows adding new operations to existing object structures without modifying them.
- Achieved by separating **algorithm from object structure**.

📖 **Why?**

- Useful when object structure is stable, but operations change frequently.

🔹 **Example (Java)**

```
interface Visitor {
    void visit(Book b);
    void visit(Computer c);
}

interface Item {
```

```
    void accept(Visitor v);
}

class Book implements Item {
    public void accept(Visitor v) { v.visit(this); }
}
class Computer implements Item {
    public void accept(Visitor v) { v.visit(this); }
}

class PriceVisitor implements Visitor {
    public void visit(Book b) { System.out.println("Book price = $20"); }
    public void visit(Computer c) { System.out.println("Computer price = $500"); }
}
```

🔹 **Analogy**

- Like a **tax officer visiting businesses** → applies new rules without changing business itself.

🎯 **Interview Tip**

Say:
"Visitor Pattern separates algorithms from objects, allowing new operations without modifying object classes. It uses double dispatch."

🔹 **Q61. What is the Composite Pattern?**

📖 **Definition**

- **Composite Pattern** lets you treat **individual objects and groups of objects** uniformly.

- Used to represent **hierarchical structures (tree-like)**.

📖 **Why?**

- Simplifies client code → you don't need to distinguish between single objects and collections.

🔹 **Example (Java)**

```
import java.util.*;

interface Employee {
    void showDetails();
}

class Developer implements Employee {
    String name;
    Developer(String n) { name = n; }
    public void showDetails() { System.out.println("Developer: " + name); }
}

class Manager implements Employee {
    String name;
    List<Employee> team = new ArrayList<>();
    Manager(String n) { name = n; }
    public void add(Employee e) { team.add(e); }
    public void showDetails() {
        System.out.println("Manager: " + name);
        for(Employee e : team) e.showDetails();
    }
}

public class Main {
    public static void main(String[] args) {
        Employee dev1 = new Developer("Alice");
        Employee dev2 = new Developer("Bob");
        Manager mgr = new Manager("John");
        mgr.add(dev1); mgr.add(dev2);
        mgr.showDetails();
    }
}
```

🔹 **Analogy**

- **Organization hierarchy** → Manager can manage employees (individuals) or other managers (groups).

🎯 Interview Tip

Say:
"Composite Pattern is used when you want to represent part-whole hierarchies, like trees. Example: file system, org chart."

🔹 Q62. What is the Command Pattern?

📖 Definition

- **Command Pattern** turns a request into a **standalone object** that contains all details about the request.
- Supports **undo/redo, logging, queuing**.

📖 Why?

- Decouples the sender of a request from its receiver.

🔹 Example (Java)

```
interface Command {
    void execute();
}

class Light {
    void on() { System.out.println("Light ON"); }
    void off() { System.out.println("Light OFF"); }
}

class LightOnCommand implements Command {
    Light light;
    LightOnCommand(Light l) { light = l; }
    public void execute() { light.on(); }
}

public class Main {
    public static void main(String[] args) {
        Light light = new Light();
        Command cmd = new LightOnCommand(light);
        cmd.execute(); // Light ON
```

```
    }
}
```

🔹 **Analogy**

- **TV remote** → button press is a command object executed on the TV.

🎯 **Interview Tip**

Say:
"Command Pattern encapsulates a request as an object. Very useful for undo/redo operations and remote controls."

🔹 **Q63. What is the Mediator Pattern?**

📖 **Definition**

- **Mediator Pattern** defines an object (mediator) that **coordinates interaction** between multiple objects.
- Prevents objects from referring to each other directly → reduces coupling.

📖 **Why?**

- Useful when you have **many-to-many communications**.

🔹 **Example (Chat Room)**

```
class ChatRoom {
    public void showMessage(String user, String msg) {
        System.out.println(user + ": " + msg);
    }
}

class User {
    String name;
    ChatRoom chat;
    User(String n, ChatRoom c) { name = n; chat = c; }
    public void send(String msg) { chat.showMessage(name, msg); }
}

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

```
        ChatRoom chat = new ChatRoom();
        User u1 = new User("Alice", chat);
        User u2 = new User("Bob", chat);
        u1.send("Hello Bob!");
        u2.send("Hi Alice!");
    }
}
```

🔹 **Analogy**

- **Air Traffic Controller** → planes don't talk directly; ATC coordinates communication.

🎯 **Interview Tip**

Say:
"Mediator Pattern centralizes complex communication logic, reducing dependencies between objects."

🔹 **Q64. What is the Memento Pattern?**

📖 **Definition**

- **Memento Pattern** captures an object's **internal state** so it can be restored later (undo functionality).

📖 **Why?**

- Useful in text editors, games, transactions.

🔹 **Example (Java)**

```
class Memento {
    String state;
    Memento(String s) { state = s; }
    String getState() { return state; }
}

class Editor {
    String text;
    void setText(String t) { text = t; }
    String getText() { return text; }
    Memento save() { return new Memento(text); }
    void restore(Memento m) { text = m.getState(); }
```

```
}

public class Main {
    public static void main(String[] args) {
        Editor editor = new Editor();
        editor.setText("Version1");
        Memento saved = editor.save();
        editor.setText("Version2");
        editor.restore(saved);
        System.out.println(editor.getText()); // Version1
    }
}
```

🔹 **Analogy**

- **Game save/load system** → save state, restore later.

🎯 **Interview Tip**

Say:
"Memento Pattern is used to capture and restore an object's state. Example: undo in text editors."

🔹 **Q65. What is the Interpreter Pattern?**

📖 **Definition**

- **Interpreter Pattern** defines a grammar for a language and uses an interpreter to interpret sentences in the language.

📖 **Why?**

- Used in **parsers, SQL engines, rule engines**.

🔹 **Example (Simple Math Interpreter)**

```
interface Expression {
    int interpret();
}

class Number implements Expression {
    int num;
    Number(int n) { num = n; }
```

```
    public int interpret() { return num; }
}

class Add implements Expression {
    Expression left, right;
    Add(Expression l, Expression r) { left = l; right = r; }
    public int interpret() { return left.interpret() + right.interpret(); }
}

public class Main {
    public static void main(String[] args) {
        Expression exp = new Add(new Number(5), new Number(10));
        System.out.println(exp.interpret()); // 15
    }
}
```

🔹 **Analogy**

- **Calculator interpreting equations** step by step.

🎯 **Interview Tip**

Say:
"Interpreter Pattern is used for parsing languages, compilers, or SQL engines."

🔹 **Q66. What is the Flyweight Pattern?**

📖 **Definition**

- **Flyweight Pattern** reduces memory usage by **sharing common objects** instead of creating duplicates.

📖 **Why?**

- Best when you have a **large number of similar objects**.

🔹 **Example (Characters in Text Editor)**

```
import java.util.*;

class CharFactory {
    private static Map<Character, Character> pool = new HashMap<>();
    public static Character getChar(char c) {
```

```
        pool.putIfAbsent(c, c);
        return pool.get(c);
    }
}

public class Main {
    public static void main(String[] args) {
        Character a1 = CharFactory.getChar('a');
        Character a2 = CharFactory.getChar('a');
        System.out.println(a1 == a2); // true (same object)
    }
}
```

🔹 **Analogy**

- **Fonts in a text editor** → instead of storing font for every character, share the same object.

🎯 **Interview Tip**

Say:
"Flyweight Pattern minimizes memory by sharing intrinsic data. Example: Java String pool, text editors."

🔹 **Q67. What is the Bridge Pattern?**

📖 **Definition**

- **Bridge Pattern** decouples abstraction from implementation → both can vary independently.

📖 **Why?**

- Helps avoid class explosion when combining multiple abstractions with implementations.

🔹 **Example (Java)**

```
interface Device {
    void turnOn();
}

class TV implements Device {
```

```
    public void turnOn() { System.out.println("TV ON"); }
}

abstract class Remote {
    protected Device device;
    Remote(Device d) { device = d; }
    abstract void pressButton();
}

class TVRemote extends Remote {
    TVRemote(Device d) { super(d); }
    void pressButton() { device.turnOn(); }
}

public class Main {
    public static void main(String[] args) {
        Device tv = new TV();
        Remote remote = new TVRemote(tv);
        remote.pressButton();
    }
}
```

🔹 **Analogy**

- **Remote control and device** → remote works with different devices without changing code.

🎯 **Interview Tip**

Say:
"Bridge Pattern separates abstraction from implementation so both can evolve independently."

🔹 **Q68. What is the Proxy Pattern?**

📖 **Definition**

- **Proxy Pattern** provides a surrogate or placeholder for another object to control access.

📖 **Types**

- **Virtual Proxy** (lazy loading).
- **Remote Proxy** (access over network).
- **Protection Proxy** (access control).

🔹 **Example (Java)**

```
interface Image {
   void display();
}

class RealImage implements Image {
   String file;
   RealImage(String f) { file = f; loadFromDisk(); }
   void loadFromDisk() { System.out.println("Loading " + file); }
   public void display() { System.out.println("Displaying " + file); }
}

class ProxyImage implements Image {
   RealImage realImage;
   String file;
   ProxyImage(String f) { file = f; }
   public void display() {
      if(realImage == null) realImage = new RealImage(file);
      realImage.display();
   }
}
```

🔹 **Analogy**

- **Movie ticket booking site** → acts as proxy between you and the cinema.

🎯 **Interview Tip**

Say:
"Proxy Pattern provides a substitute to control access. Example: lazy loading images, RMI in Java."

🔹 **Q69. What is the Chain of Responsibility Pattern?**

📖 **Definition**

- **Chain of Responsibility** passes a request along a chain of handlers until one handles it.

📖 **Why?**

- Decouples sender from receiver.

🔹 **Example (Java)**

```
abstract class Handler {
   Handler next;
   void setNext(Handler h) { next = h; }
   abstract void handle(int level);
}

class Manager extends Handler {
   void handle(int level) {
      if(level <= 1) System.out.println("Manager approved");
      else if(next != null) next.handle(level);
   }
}
class Director extends Handler {
   void handle(int level) {
      if(level <= 2) System.out.println("Director approved");
      else if(next != null) next.handle(level);
   }
}
```

🔹 **Analogy**

- **Customer support escalation** → rep → supervisor → manager.

🎯 **Interview Tip**

Say:
"Chain of Responsibility lets multiple handlers process a request without knowing which one will handle it."

🔹 **Q70. What is the Template Method Pattern?**

📖 **Definition**

- **Template Method** defines a **skeleton of an algorithm** in a base class but lets subclasses provide specific implementations.

📖 **Why?**

- Promotes **code reuse** and enforces a fixed sequence of steps.

🔹 **Example (Java)**

```
abstract class Game {
    abstract void start();
    abstract void end();
    public final void play() {
        start();
        end();
    }
}

class Chess extends Game {
    void start() { System.out.println("Chess start"); }
    void end() { System.out.println("Chess end"); }
}
```

🔹 **Analogy**

- **Recipe** → steps fixed, but ingredients vary.

🎯 **Interview Tip**

Say:
"Template Method fixes the algorithm structure while allowing subclasses to override specific steps."

🔹 **Q71. Difference between Factory Method and Abstract Factory Pattern?**

📖 **Factory Method**

- Defines an **interface for creating an object**, but subclasses decide which object to create.

- One **family of product**.

📖 **Abstract Factory**

- Provides an **interface for creating families of related objects** without specifying their classes.
- Multiple product families.

🔹 **Example**

Factory Method → `ShapeFactory` creates shapes (circle/square).
Abstract Factory → `GUIFactory` creates both buttons and checkboxes, but in different styles (Windows, Mac).

🎯 **Interview Tip**

Say:
"Factory Method = one product. Abstract Factory = product families."

🔹 **Q72. What is the State Pattern?**

📖 **Definition**

- **State Pattern** allows an object to change its behavior when its **internal state** changes.
- Looks like the object's class changed.

🔹 **Example (Java)**

```
interface State {
   void handle();
}
class OnState implements State {
   public void handle() { System.out.println("Light is ON"); }
}
class OffState implements State {
   public void handle() { System.out.println("Light is OFF"); }
}
class Light {
   State state;
   void setState(State s) { state = s; }
   void press() { state.handle(); }
}
```

🔹 **Analogy**

- **Fan speed** → same fan object behaves differently based on state (low/medium/high).

🎯 **Interview Tip**

Say:
"State Pattern lets an object alter behavior dynamically by switching its internal state."

🔹 **Q73. What is the Visitor Pattern?**

📖 **Definition**

- **Visitor Pattern** separates an algorithm from the object structure it operates on.
- Useful when you need to perform operations on objects without changing their classes.

🔹 **Example (Java)**

```
interface Visitor { void visit(Book b); }
class Book {
    void accept(Visitor v) { v.visit(this); }
}
class PriceVisitor implements Visitor {
    public void visit(Book b) { System.out.println("Price check for Book"); }
}
```

🔹 **Analogy**

- **Tax auditor** visits different entities → applies rules without changing entity structure.

🎯 **Interview Tip**

Say:
"Visitor Pattern is used when you need to add operations to classes without modifying them."

🔹 **Q74. What is Double Dispatch?**

📖 **Definition**

- **Single Dispatch** = method executed depends on **object's runtime type**.
- **Double Dispatch** = method executed depends on **runtime type of two objects**.

🔹 **Example**

In **Visitor Pattern**, both element and visitor decide behavior.

🎯 **Interview Tip**

Say:
"Double Dispatch ensures correct method call based on runtime type of two interacting objects. Example: Visitor Pattern."

🔹 **Q75. What is a Mix-in in OOP?**

📖 **Definition**

- A **Mix-in** is a class containing methods to add extra functionality but is not meant for standalone use.
- Popular in **Python, Ruby**.

🔹 **Example (Python)**

```
class LoggerMixin:
    def log(self, msg):
        print("LOG:", msg)

class App(LoggerMixin):
    def run(self):
        self.log("Running App")

a = App()
a.run()
```

🎯 **Interview Tip**

Say:
"Mix-in provides reusable functionality across classes without forming deep inheritance hierarchies."

🔹 Q76. What is Monkey Patching in OOP?

📖 Definition

- **Monkey Patching** = dynamically modifying a class or module at runtime.
- Often used in scripting languages (Python, Ruby).

🔹 Example (Python)

```
class Dog:
    def bark(self): print("Woof")

def new_bark(self): print("Bow Bow")

Dog.bark = new_bark
d = Dog()
d.bark()  # Bow Bow
```

🎯 Interview Tip

Say:
"Monkey patching is powerful but dangerous since it changes behavior at runtime and can break code."

🔹 Q77. What is Reflection in OOP?

📖 Definition

- **Reflection** = ability of a program to inspect and modify itself at runtime.

🔹 Example (Java)

```
class Person {
    private String name = "Alice";
}
public class Test {
    public static void main(String[] args) throws Exception {
        Class<?> c = Class.forName("Person");
        java.lang.reflect.Field f = c.getDeclaredField("name");
        f.setAccessible(true);
        System.out.println(f.get(new Person())); // Alice
```

```
    }
}
```

🎯 Interview Tip

Say:
"Reflection allows runtime inspection and modification of classes, fields, and methods. Used in frameworks like Hibernate, Spring."

🔹 Q78. What is an Anonymous Class?

📖 Definition

- **Anonymous Class** = a class without a name, declared and instantiated at the same time.
- Commonly used in Java for quick implementations.

🔹 Example (Java)

```
Runnable r = new Runnable() {
    public void run() { System.out.println("Running"); }
};
new Thread(r).start();
```

🎯 Interview Tip

Say:
"Anonymous classes provide concise, one-time use implementations, often used in event handling."

🔹 Q79. Difference between Cohesion and Coupling in OOP?

📖 Cohesion

- Degree to which elements of a class belong together.
- **High cohesion = good**.

📖 Coupling

- Degree of dependency between classes/modules.
- **Low coupling = good**.

🔹 **Analogy**

- **Restaurant kitchen** → chef, waiter, cashier (high cohesion, low coupling).

🎯 **Interview Tip**

Say:
"Good OOP design has high cohesion and low coupling for maintainability and flexibility."

🔹 **Q80. What is Tight Coupling vs Loose Coupling?**

📖 **Tight Coupling**

- One class is highly dependent on another.
- Difficult to test/maintain.

📖 **Loose Coupling**

- Classes communicate through abstractions (interfaces).
- Promotes flexibility.

🔹 **Example**

- **Tight coupling**: `Car` directly creates `Engine`.
- **Loose coupling**: `Car` depends on `Engine` interface, actual engine injected at runtime.

🎯 **Interview Tip**

Say:
"Loose coupling improves testability and maintainability. Achieved using interfaces, DI, and design patterns."

🔹 Q81. What is the Dependency Inversion Principle (DIP)?

📖 Definition

- Part of **SOLID principles**.
- High-level modules should **not depend on low-level modules**; both should depend on **abstractions**.
- Abstractions should not depend on details, but details should depend on abstractions.

🔹 Example (Java)

❌ Without DIP

```
class LightBulb {
    void turnOn() { System.out.println("Bulb on"); }
}
class Switch {
    private LightBulb bulb = new LightBulb();
    void press() { bulb.turnOn(); }
}
```

✔ With DIP

```
interface Switchable { void turnOn(); }
class LightBulb implements Switchable {
    public void turnOn() { System.out.println("Bulb on"); }
}
class Switch {
    private Switchable device;
    Switch(Switchable d) { device = d; }
    void press() { device.turnOn(); }
}
```

🎯 Interview Tip

Say:
"DIP decouples high-level and low-level modules by using abstractions. This improves flexibility and testability."

🔹 Q82. What is the Law of Demeter (LoD)?

📖 **Definition**

- Also called **Principle of Least Knowledge**.
- A class should **only talk to its immediate friends**, not "friends of friends".

🔹 **Example ❌ (Violation)**

```
customer.getOrder().getItem().getPrice(); // too deep chaining
```

✔ Correct approach: delegate inside `Order` or `Customer`.

🎯 **Analogy**

- Don't gossip with your friend's friend → talk only to people you directly know.

🎯 **Interview Tip**

Say:
"LoD reduces coupling by restricting method chaining and promoting proper delegation."

🔹 **Q83. What is an Aggregate Root in OOP/DDD?**

📖 **Definition**

- In **Domain-Driven Design (DDD)**, an **aggregate root** is the entry point to a cluster of related objects.
- Enforces invariants across the whole group.

🔹 **Example**

- `Order` (aggregate root) → contains `OrderItems`.
- External code should interact with `Order`, not directly with `OrderItem`.

🎯 **Interview Tip**

Say:
"Aggregate Root is the main entity that controls access to related entities, ensuring data consistency in domain-driven design."

🔹 Q84. What is the Repository Pattern?

📖 Definition

- A **Repository** mediates between domain objects and data access logic.
- Provides an in-memory collection-like interface for persistence.

🔹 Example (Java)

```
interface StudentRepository {
   void save(Student s);
   Student findById(int id);
}
```

Used in frameworks like **Spring Data JPA**.

🎯 Interview Tip

Say:
"Repository Pattern abstracts data storage, providing cleaner separation between domain and database logic."

🔹 Q85. What is the Service Layer Pattern?

📖 Definition

- A **Service Layer** contains business logic and coordinates operations between domain objects and repositories.
- Helps maintain separation of concerns.

🔹 Example

```
class OrderService {
   private OrderRepository repo;
   void placeOrder(Order o) { repo.save(o); }
}
```

🎯 Interview Tip

Say:
"Service Layer groups business logic in one place, separating it from persistence and UI."

🔹 Q86. Difference between Composition and Inheritance in OOP?

📖 Inheritance

- "**IS-A**" relationship.
- Reuses parent class code.
- Can cause tight coupling.

📖 Composition

- "**HAS-A**" relationship.
- Reuses functionality by including objects inside classes.

🔹 Example

- Inheritance: `Car extends Vehicle`.
- Composition: `Car has Engine`.

🎯 Interview Tip

Say:
"Prefer composition over inheritance because it provides flexibility and avoids fragile hierarchies."

🔹 Q87. What is Method Overriding vs Method Hiding?

📖 Overriding

- Subclass provides its own implementation of a superclass method.
- Happens at **runtime**.

- Supports polymorphism.

📖 **Hiding**

- If a subclass defines a **static method** with same signature, it hides the parent's method (not overrides).
- Happens at **compile time**.

🔹 **Example (Java)**

```
class Parent {
   static void show() { System.out.println("Parent"); }
   void display() { System.out.println("Parent display"); }
}
class Child extends Parent {
   static void show() { System.out.println("Child"); }  // hiding
   void display() { System.out.println("Child display"); } // overriding
}
```

🎯 **Interview Tip**

Say:
"Static methods are hidden, instance methods are overridden."

🔹 **Q88. What are Value Objects in OOP/DDD?**

📖 **Definition**

- A **Value Object** is an object with **no identity**, defined only by its values.
- Immutable by design.

🔹 **Example**

- `Money(100, "USD")`, `Address("123 Street", "NY")`.
- Two `Money(100, "USD")` are equal if values match.

🎯 **Interview Tip**

Say:
"Value Objects are immutable and compared by value, not by identity. Example: Money, Date."

🔹 Q89. Difference between Fail-fast and Fail-safe Iterators in OOP?

📖 Fail-fast

- Immediately throw **ConcurrentModificationException** if the collection is structurally modified during iteration.
- Example: `ArrayList`, `HashMap`.

📖 Fail-safe

- Work on a **copy** of the collection.
- Do not throw exceptions.
- Example: `CopyOnWriteArrayList`, `ConcurrentHashMap`.

🎯 Interview Tip

Say:
"Fail-fast iterators detect modification and fail immediately, while fail-safe iterators work on a copy to avoid errors."

🔹 Q90. What is Serialization in OOP?

📖 Definition

- **Serialization** = process of converting an object into a byte stream for storage or transfer.
- **Deserialization** = reconstructing object from bytes.

🔹 Example (Java)

```
class Student implements Serializable {
   int id; String name;
}
```

🔹 **Uses**

- Saving state to file.
- Sending objects across network (RMI, Web Services).

🎯 **Interview Tip**

Say:
"Serialization lets objects be persisted or transmitted. In Java, it's done using `Serializable` *interface."*

🔹 **Q91. What is the difference between Shallow Copy and Deep Copy in OOP?**

📖 **Shallow Copy**

- Copies object **but not referenced objects** inside it.
- Inner objects still point to the same references.

📖 **Deep Copy**

- Copies object **and all nested objects** recursively.

🔹 **Example (Python)**

```
import copy
a = [[1,2],[3,4]]
shallow = copy.copy(a)
deep = copy.deepcopy(a)
```

- Modifying `a[0][0]` affects `shallow` but not `deep`.

🎯 **Interview Tip**

"Shallow copy = one level copy. Deep copy = full independent clone."

🔹 Q92. What are Immutable Objects? Why are they useful?

📖 Immutable Object

- Once created, cannot be modified.
- Instead of changing state, a new object is created.

🔹 Examples

- `String` in Java.
- `tuple` in Python.

📖 Benefits

- Thread-safe.
- Easier to cache and use as map keys.

🎯 Interview Tip

"Immutability prevents accidental state change, making objects safer in concurrent systems."

🔹 Q93. What is Defensive Copying?

📖 Definition

- Creating a copy of a mutable object instead of exposing internal references.

📖 Example (Java)

```
class Student {
   private Date dob;
   public Date getDob() { return new Date(dob.getTime()); } // defensive copy
}
```

🎯 Interview Tip

"Defensive copying prevents external code from modifying internal state."

🔹 Q94. What is the Role of equals() and hashCode() in OOP?

📖 **equals()**

- Defines logical equality between objects.

📖 **hashCode()**

- Returns integer hash used in hash-based collections.

🔹 **Rule**

- If `a.equals(b)` is true → `a.hashCode() == b.hashCode()` must also hold.

🎯 **Interview Tip**

"equals() defines equality, hashCode() enables fast lookups. Always override both together."

🔹 Q95. What is the Liskov Substitution Principle (LSP)?

📖 **Definition**

- Subclasses must be **substitutable** for their parent classes without breaking correctness.

❌ **Violation Example**

- `Square extends Rectangle` but changing width/height independently breaks behavior.

🎯 **Interview Tip**

"LSP ensures derived classes don't break parent behavior. Example: Avoid forcing `Square` to be `Rectangle`."

🔹 Q96. How does OOP help in handling concurrency?

📖 **Points**

- Encapsulation → prevents unintended shared state.
- Immutable objects → safe for concurrent access.
- Thread-safe classes → synchronize access.
- Actor model (objects process messages independently).

🎯 **Interview Tip**

"OOP concurrency is managed by encapsulating state, using immutability, and thread-safe design."

🔹 **Q97. What is the Active Object Pattern?**

📖 **Definition**

- Concurrency design pattern that decouples method invocation from execution.
- Each object has its **own thread** of control and processes requests asynchronously.

📖 **Use Case**

- GUI frameworks, distributed systems.

🎯 **Interview Tip**

"Active Object Pattern lets objects run in their own thread and process requests asynchronously."

🔹 **Q98. What is the Role of Interfaces in OOP Design?**

📖 **Interfaces provide:**

- Abstraction (contract, not implementation).
- Multiple inheritance of type.
- Decoupling (client code depends on interface, not concrete class).

🎯 Interview Tip

“Interfaces define contracts, promote loose coupling, and enable polymorphism.”

🔹 Q99. How does OOP help in designing Distributed Systems?

📖 Key Points

- Objects map naturally to distributed components (e.g., services).
- Encapsulation hides implementation behind interfaces.
- Serialization enables object transfer.
- Patterns like **Proxy, Remote Facade, DTO** apply.

🎯 Interview Tip

“OOP’s encapsulation, abstraction, and polymorphism make it natural for building modular, distributed services.”

🔹 Q100. What are some OOP Design Trade-offs?

📖 Trade-offs

1. **Flexibility vs. Complexity** – too many abstractions → hard to maintain.
2. **Inheritance vs. Composition** – inheritance is simple but rigid; composition is flexible but adds indirection.
3. **Performance vs. Encapsulation** – abstraction layers may reduce speed.
4. **Immutability vs. Efficiency** – immutable objects are safe but can create more garbage.

🎯 Interview Tip

“OOP brings maintainability and scalability, but sometimes adds performance cost and complexity. Balance is key.”