OOP I Summary (Java)

* Classes
* Interfaces
* Encapsulation
* Abstraction
* Inheritance
* Polymorphism

Classes

* Variables and data types
* Create classes
* Executable class
* Fields & Methods
* New instance of objects
* Accessing fields and methods of an object
* Constructor

Abstract Classes

* Abstract method
* Constructor

Interfaces

* Pure abstraction/Is it?

Inheritance

* Classes/objects sharing commonalities

Implementation

* Interface-led

Access modifiers

Polymorphism

SOLID

* Single Responsibility Principle (SRP)
* Open-Closed Principle (OCP)
* Liskov Substitution Principle (LSP)
* Interface Segregation Principle (ISP)
* Dependency Inversion Principle (DIP)

**🤯 OOP I Summary (Java)**

OOP is about designing software using real-world concepts like **objects**, **classes**, **inheritance**, and **polymorphism**. Java is built around these ideas.

**Key Principles**

* **Classes**: Templates for creating objects.
* **Interfaces**: Contracts defining behavior.
* **Encapsulation**: Hide internal data.
* **Abstraction**: Hide complexity.
* **Inheritance**: Reuse common behavior.
* **Polymorphism**: Same interface, different implementation.

**🔹 Classes**

**◾ Variables and Data Types**

int age = 25;

double height = 5.9;

boolean isActive = true;

char grade = 'A';

String name = "Alice";

**◾ Creating Classes**

public class Person {

String name;

int age;

}

**◾ Executable Class with main()**

public class Main {

public static void main(String[] args) {

System.out.println("Hello, Java!");

}

}

**◾ Fields & Methods**

public class Calculator {

int result;

int add(int a, int b) {

return a + b;

}

}

**◾ Instantiating Objects**

Calculator calc = new Calculator();

System.out.println(calc.add(5, 3));

**◾ Accessing Fields and Methods**

calc.result = 10;

System.out.println(calc.result);

**◾ Constructors**

public class Book {

String title;

public Book(String title) {

this.title = title;

}

}

Book book = new Book("Java 101");

**🔹 Abstract Classes**

**◾ Abstract Method**

abstract class Animal {

abstract void makeSound();

}

**◾ Constructor in Abstract Class**

abstract class Shape {

Shape() {

System.out.println("Shape Created");

}

}

**🔹 Interfaces**

**◾ What is Pure Abstraction?**

Interfaces **define behavior** without implementing it (before Java 8).

interface Drawable {

void draw();

}

class Circle implements Drawable {

public void draw() {

System.out.println("Drawing Circle");

}

}

From Java 8+, interfaces can have default and static methods.

interface Printer {

default void status() {

System.out.println("Ready to print.");

}

}

**🔹 Inheritance**

**◾ Classes Sharing Commonalities**

class Vehicle {

void start() {

System.out.println("Starting vehicle");

}

}

class Car extends Vehicle {

void drive() {

System.out.println("Driving car");

}

}

Car inherits start() from Vehicle.

**🔹 Interface-led Implementation**

Encourages decoupling and flexibility.

interface Engine {

void start();

}

class ElectricEngine implements Engine {

public void start() {

System.out.println("Electric engine starts silently");

}

}

**🔹 Access Modifiers**

|  |  |
| --- | --- |
| **Modifier** | **Scope** |
| private | Same class only |
| default | Same package |
| protected | Package + subclasses |
| public | Anywhere |

Example:

public class BankAccount {

private int balance = 1000;

public int getBalance() {

return balance;

}

}

**🔹 Polymorphism**

One interface, many implementations.

class Animal {

void sound() {

System.out.println("Animal sound");

}

}

class Dog extends Animal {

void sound() {

System.out.println("Bark");

}

}

Animal pet = new Dog();

pet.sound(); // Outputs: Bark

**🪛 SOLID Principles**

**1. ✅ Single Responsibility Principle**

One class = one job.

class InvoicePrinter {

void print(Invoice invoice) {

// logic here

}

}

**2. ✅ Open-Closed Principle**

Open for extension, closed for modification.

interface Shape {

double area();

}

class Circle implements Shape {

double radius;

public double area() {

return Math.PI \* radius \* radius;

}

}

**3. ✅ Liskov Substitution Principle**

Subclasses must be substitutable.

class Bird {

void fly() {}

}

class Sparrow extends Bird {} // OK

// class Penguin extends Bird {} // ❌ LSP issue if can't fly

**4. ✅ Interface Segregation Principle**

Clients shouldn’t depend on unused interfaces.

interface Printer {

void print();

}

interface Scanner {

void scan();

}

**5. ✅ Dependency Inversion Principle**

Depend on abstractions.

interface MessageService {

void sendMessage(String message);

}

class EmailService implements MessageService {

public void sendMessage(String message) {

System.out.println("Sending email: " + message);

}

}

class Notification {

private MessageService service;

public Notification(MessageService service) {

this.service = service;

}

void notifyUser() {

service.sendMessage("Welcome!");

}

}