

Object-Oriented Programming (OOPS) in Java

1 Object-Oriented Programming (OOP)

Definition (WRITE THIS)

Object-Oriented Programming is a paradigm that organizes software around **objects**, where each object represents a **real-world entity** having **state, behavior, and identity**.

Key Characteristics

- Program is divided into **objects**
 - Objects interact using **methods**
 - Focus is on **data + behavior**, not just logic
-

Advantages

- Code reusability
 - Modularity
 - Maintainability
 - Scalability
 - Real-world modeling
-

Four Pillars of OOP

1. **Encapsulation**
2. **Inheritance**

3. Polymorphism

4. Abstraction

2 Class and Object

Class

- A **logical blueprint**
 - Defines:
 - Variables (state)
 - Methods (behavior)
 - Constructors
 - ```
class Student {
```
  - ```
    int id;
```
 - ```
 String name;
```
  - ```
}
```
-

Object

- A **runtime instance** of a class
 - Occupies memory
 - ```
Student s = new Student();
```
- 

### Exam Point

One class can create **multiple objects**, but each object has its own state.

---

## 3 Encapsulation

### Definition (VERY IMPORTANT)

Encapsulation is the process of **binding data and methods together** and **restricting direct access** to data.

---

### How Encapsulation is Achieved

1. Declare variables as **private**
2. Provide access through **public** methods

- ```
class Account {
```
- ```
 private double balance;
```
- 
- ```
    public double getBalance() {
```
- ```
 return balance;
```
- ```
    }
```
- ```
}
```

---

### Benefits

- Data hiding
  - Controlled access
  - Loose coupling
  - Increased security
- 

### Exam Tip

- ✗ Public variables = **poor encapsulation**
  - ✓ Private variables + getters/setters = **proper encapsulation**
-

## 4 Constructors

### Definition

A constructor is a **special member of a class** used to **initialize objects**.

---

### Characteristics

- Same name as class
  - No return type
  - Invoked automatically during object creation
  - Can be overloaded
- 

### Types of Constructors

#### Default Constructor

- Provided by compiler
- Only if **no constructor is defined**

#### No-Argument Constructor

- `public Student() { }`

#### Parameterized Constructor

- `public Student(int id, String name) { }`
- 

### Constructor Overloading

- Multiple constructors with different parameter lists

---

## Important Rules (EXAM FAVORITE)

- Constructors are **not inherited**
  - First statement is `super()` (implicit or explicit)
  - Used only for **initialization**, not business logic
- 

## 5 Inheritance (IS-A Relationship)

### Definition

Inheritance allows a class to **acquire properties and methods** of another class.

- `class SavingsAccount extends Account { }`
- 

### Purpose

- Code reuse
  - Logical hierarchy
  - Runtime polymorphism
- 

### Types of Inheritance in Java

1. Single
2. Multilevel
3. Hierarchical

✗ Multiple inheritance using classes is **not supported**

---

## Important Rules

- Uses `extends` keyword
  - Private members are not inherited
  - Constructors are not inherited
  - `super` refers to parent class object
- 

## 6 Object Class

### Key Facts (HIGH SCORING)

- Root class of Java
  - Every class implicitly extends `Object`
- 

### Common Methods

- `toString()`
  - `equals()`
  - `hashCode()`
  - `getClass()`
  - `wait(), notify()`
- 

### Exam Point

Polymorphism is possible because all objects are treated as `Object` references.

---

## 7 Polymorphism

### Definition

Polymorphism means **one method behaving differently in different situations**.

---

### Types of Polymorphism

1. **Compile-time Polymorphism**
  2. **Runtime Polymorphism**
- 

## 8 Method Overloading (Compile-Time Polymorphism)

### Definition

Method overloading allows multiple methods with the **same name but different parameter lists**.

- `void add(int a, int b)`
  - `void add(double a, double b)`
- 

### Rules

- Parameter list must differ
  - Return type alone is not sufficient
  - Binding occurs at compile time
- 

### Exam Note

Overloading is resolved using **reference type**.

---

## 9 Method Overriding (Runtime Polymorphism)

### Definition

Method overriding occurs when a subclass provides its **own implementation** of a parent class method.

- `@Override`
  - `public void display() { }`
- 

### Rules

- Same method signature
  - Same or wider access modifier
  - Cannot override `final` or `private` methods
  - Static methods are hidden, not overridden
- 

### Exam Note

Overriding is resolved using **object type** at runtime.

---

## 10 Abstraction

### Definition

Abstraction is the process of **hiding implementation details** and exposing only **essential features**.

---

### Achieved Using

1. Abstract classes

## 2. Interfaces

---

### 11 Abstract Class

#### Definition

An abstract class is a class that:

- Cannot be instantiated
- Can contain abstract and concrete methods
- Can have fields and constructors
- `abstract class Shape {`
- `abstract double area();`
- `}`

---

#### Key Points

- Uses `abstract` keyword
- Supports inheritance
- Provides partial implementation

---

### 12 Abstract Methods

#### Definition

An abstract method is a method declared **without implementation**.

- `abstract void calculateArea();`
-

## Rules

- Must be implemented by subclass
  - Cannot be `private`, `static`, or `final`
  - Forces correctness
- 

## 13 Interfaces

### Definition (WRITE THIS)

An interface is a **pure contract** that specifies **what a class can do**, not how it does it.

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

### Key Characteristics

- No instance variables
  - Methods are `public abstract` by default
  - Supports multiple inheritance
  - Cannot be instantiated
- 

### Java 8+ Features

- Default methods
- Static methods
- Private methods (Java 9)

---

## 14 Abstract Class vs Interface (VERY IMPORTANT)

| Feature              | Abstract Class | Interface |
|----------------------|----------------|-----------|
| Instantiation        | ✗              | ✗         |
| State                | Yes            | No        |
| Constructors         | Yes            | No        |
| Multiple inheritance | No             | Yes       |
| Purpose              | IS-A           | CAN-DO    |

---

## 15 Why Java Does NOT Support Multiple Inheritance

### Reasons

- Avoids **Diamond Problem**
  - Prevents ambiguity
  - Simplifies JVM design
  - Achieved safely using interfaces
-

## FINAL QUICK SUMMARY

- Encapsulation → data hiding
- Inheritance → code reuse
- Polymorphism → flexibility
- Abstraction → complexity reduction
- Overloading → compile time
- Overriding → runtime
- Abstract class → partial blueprint
- Interface → pure contract