

OOPs In Java

□ What is a Class in Java?

A **class** is a blueprint or template for creating objects.
It defines **properties (fields)** and **behaviors (methods)** an object can have.

□ Syntax:

```
class ClassName {  
    // Fields (variables)  
    // Methods (functions)  
}
```

□ Example:

```
class Car {  
    String brand;  
    int speed;  
  
    void drive() {  
        System.out.println(brand + " is driving at " + speed + " km/h.");  
    }  
}
```

□ What is an Object in Java?

An **object** is an instance of a class.
It has its own identity, state (data), and behavior (methods).

□ Syntax:

```
ClassName objName = new ClassName();
```

□ Example:

```
public class Main {  
    public static void main(String[] args) {  
        Car myCar = new Car();      // Creating object  
        myCar.brand = "Toyota";
```

```
myCar.speed = 80;  
myCar.drive();           // Output: Toyota is driving at 80 km/h  
}  
}
```

□ **Real-Life Analogy:**

Concept	Real-Life Analogy
Class	Blueprint of a house
Object	Actual house built from the blueprint

Key Interview Points:

Question	Answer
Is it possible to create multiple objects from one class?	Yes, each object will have its own copy of instance variables.
Where is object memory allocated?	On the heap.
What keyword is used to create an object?	new keyword
Are classes data types?	Yes, user-defined reference data types.

OOP Concept: Constructor in Java

□ **What is a Constructor?**

A **constructor** is a special method used to initialize objects. It is **automatically called** when an object is created.

□ **Characteristics:**

- Same name as the class
- No return type (not even void)
- Can be **default, parameterized, or overloaded**

□ **Syntax of Constructor:**

```
class ClassName {  
    // Constructor  
    ClassName() {  
        // Initialization code  
    }  
}
```

□ **Types of Constructors in Java**

1.Default Constructor

Automatically provided by Java **if no constructor** is defined.
It initializes variables with **default values**.

□ **Example:**

```
class Student {  
    String name;  
    int age;  
  
    Student() {  
        name = "Not Assigned";  
        age = 0;  
    }  
  
    void display() {  
        System.out.println(name + " - " + age);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Student s1 = new Student(); // Default constructor called  
        s1.display();           // Output: Not Assigned - 0
```

```
    }  
}
```

2.Parameterized Constructor

Used to initialize the object with **custom values** at the time of creation.

□ Example:

```
class Student {  
    String name;  
    int age;  
  
    Student(String n, int a) { // Parameterized constructor  
        name = n;  
        age = a;  
    }  
  
    void display() {  
        System.out.println(name + " - " + age);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Student s1 = new Student("Prathamesh", 21);  
        s1.display(); // Output: Prathamesh - 21  
    }  
}
```

3.Overloaded Constructor

Java allows **multiple constructors** in a class with **different parameter lists**.

□ Example:

```
class Student {  
    String name;
```

```
int age;

// Default Constructor
Student() {
    name = "Unknown";
    age = 0;
}

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

void display() {
    System.out.println(name + " - " + age);
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();           // Calls default
        Student s2 = new Student("Ravi", 22); // Calls parameterized
        s1.display(); // Unknown - 0
        s2.display(); // Ravi - 22
    }
}
```

□ Real-Life Analogy:

Concept	Analogy
Constructor	Application form constructor (you fill details)
Default	Empty form with default values
Parameterized	Pre-filled form

Concept	Analogy
Overloaded	Multiple forms with different sections

Key Interview Points:

Question	Answer
Can we overload constructors?	Yes, by changing number/type/order of parameters
Is constructor inherited?	No
Can constructor be private?	Yes, used in Singleton design pattern
Can constructor call another constructor?	Yes, using this() keyword

OOP Concept: this Keyword in Java

What is this Keyword?

this is a **reference variable** in Java that refers to the **current object**.

Whenever there is a **naming conflict** between **instance variables and parameters**, this helps distinguish them.

Use Cases of this Keyword:

Use Case	Description
1. Resolve instance vs parameter naming conflict	Common in constructors & setters

Use Case	Description
2. Call one constructor from another	Using this()
3. Pass current object as parameter	To methods or constructors
4. Return current object	Useful in method chaining

1.Resolving Naming Conflict

```
class Student {  
    String name;  
  
    Student(String name) {  
        this.name = name; // this.name → instance, name → parameter  
    }  
  
    void display() {  
        System.out.println("Name: " + this.name);  
    }  
}
```

2.Calling Constructor from Another Constructor

```
class Student {  
    String name;  
    int age;  
  
    Student() {  
        this("Unknown", 0); // calls parameterized constructor  
    }  
  
    Student(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    void display() {
```

```
        System.out.println(name + " - " + age);
    }
}
```

- this() must be the **first statement** in the constructor.
-

3.Passing Current Object

```
class Student {
    void show(Student s) {
        System.out.println("Object received: " + s);
    }

    void call() {
        show(this); // passes current object to method
    }
}
```

4.Returning Current Object

```
class Student {
    Student getStudent() {
        return this; // returns current object
    }

    void display() {
        System.out.println("Returned object called");
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.getStudent().display(); // Method chaining
    }
}
```

Real-Life Analogy:

Think of this as saying “**myself**” in real life.

```
this.name = name;  
// Means: My name = Given name
```

Interview Points:

Question	Answer
What is this in Java?	It is a reference to the current object
Can this() be used in methods?	No, only in constructors
Where must this() appear in a constructor?	As the first line
What happens if we remove this in constructors with same variable names?	The assignment won't affect instance variables

Object-Oriented Programming (OOP)

Java is based on **Object-Oriented Programming (OOP)**, and it revolves around four key principles:

Encapsulation
Inheritance
Polymorphism
Abstraction

1. Encapsulation

Definition:

Encapsulation is the process of **wrapping data (variables)** and **code (methods)** together into a **single unit** (class) and restricting direct access to some of the object's components.

□ **Goal: Data Hiding + Data Binding**

□ **How in Java?**

- Declare variables as **private**
- Provide **public getters and setters**

□ **Example:**

```
class Employee {  
    private String name;    // Private data  
    private double salary;  
  
    public void setName(String name) {  
        this.name = name;    // Setter method  
    }  
  
    public String getName() {  
        return name;        // Getter method  
    }  
}
```

Benefits:

- Improves **security**
- Makes the class **modular & maintainable**
- Enables **control over data**

□ **Real-Life Example:**

ATM Machine – You interact using buttons (methods) without knowing internal hardware (data).

2.Inheritance

Definition:

Inheritance is the **mechanism of acquiring** properties and behaviors of one class (**parent/superclass**) into another (**child/subclass**).

Goal: Code Reusability

Java Syntax:

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

Types:

- Single
- Multilevel
- Hierarchical

(Note: Java does not support multiple inheritance with classes directly)

Real-Life Example:

Child inherits traits from parents — eyes, height, etc.

Note:

Java supports **multiple inheritance** via **interfaces**, not classes.

3.Polymorphism

Definition:

Polymorphism means "**many forms**". The same method can behave **differently** based on object context.

Goal: Flexibility & Maintainability

Types in Java:

Type	Description	Example
Compile-time	Method Overloading	sum(int, int) and sum(float, float)
Runtime	Method Overriding	Overriding run() in Thread class

Example: Method Overloading (Compile-time)

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

Example: Method Overriding (Runtime)

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

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

4.Abstraction

Definition:

Abstraction means hiding **implementation details** and only exposing **essential features**.

□ **Goal: Security + Simplicity**

Achieved via:

- **Abstract class** (partial abstraction)
- **Interface** (100% abstraction till Java 7, now supports default methods)

□ **Abstract Class Example:**

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

□ **Interface Example:**

```
interface Shape {  
    void draw(); // abstract by default  
}  
  
class Circle implements Shape {  
    public void draw() {  
        System.out.println("Drawing Circle");  
    }  
}
```

□ **Real-Life Example:**

Using a TV remote — you press a button (interface) without knowing internal circuits (implementation).

Comparison Table

Principle	Purpose	Achieved via	Real-Life Example
Encapsulation	Hide data	Private fields + getters	ATM, Medical capsule
Inheritance	Reuse code	extends keyword	Child inheriting traits
Polymorphism	Flexibility	Overloading/Overriding	One task, many forms
Abstraction	Hide complexity	Abstract class, Interface	Remote, Car pedal

OOPs Concept: Encapsulation in Java

Definition:

Encapsulation is the technique of **binding data (variables)** and **methods (functions)** together into a single unit (class) and **restricting access** to some of the object's components.

Key Goals:

- **Data Hiding**
- **Controlled Access**
- **Improved Security**
- **Code Modularity**

How to Achieve Encapsulation in Java?

Step	Description
1.	Declare class variables as private
2.	Provide public getter and setter methods
3.	Access or modify data only through these methods

□ **Java Example:**

```
class BankAccount {  
    private String accountHolder;  
    private double balance;  
  
    // Getter for accountHolder  
    public String getAccountHolder() {  
        return accountHolder;  
    }  
  
    // Setter for accountHolder  
    public void setAccountHolder(String name) {  
        accountHolder = name;  
    }  
  
    // Getter for balance  
    public double getBalance() {  
        return balance;  
    }  
  
    // Setter for balance with validation  
    public void setBalance(double amount) {  
        if (amount > 0)  
            balance = amount;  
    }  
}
```

□ **Real-Life Analogy:**

Medical Capsule:

Medicine (data) is hidden inside the capsule (class), and it can only be accessed in a controlled way.

Advantages of Encapsulation

Feature	Benefit
Data hiding	Protects internal state of object
Security	Sensitive info like password, balance
Reusability	Classes become modular & reusable
Control	Set rules via validation in setters
Maintainability	Easy to change internals without affecting other code

□ Interview-Level Points:

Question	Answer
Is encapsulation same as abstraction?	No. Encapsulation hides data; abstraction hides implementation.
Can we achieve encapsulation without using private variables?	Not fully. Private scope is essential for full encapsulation.
Which design principle supports encapsulation?	Data Hiding is the core idea behind it.
Is JavaBeans an example of encapsulation?	Yes. It follows the encapsulation pattern.

OOPs Concept: Abstraction in Java

□ Definition:

Abstraction is the process of **hiding the internal implementation details and showing only the essential features** to the user.

It helps in **reducing complexity** and **increasing efficiency** by focusing only on what an object does rather than how it does it.

□ Key Objectives of Abstraction:

- Hide complexity from the user
 - Provide a simple interface
 - Achieve loose coupling between classes
-

□ How to Achieve Abstraction in Java?

Java provides two ways to achieve abstraction:

Way	Keyword Used	Level of Abstraction
Abstract Class	abstract class	Partial Abstraction
Interface	interface	Full Abstraction (Java 7)

□ Example 1: Using Abstract Class

```
abstract class Animal {  
    abstract void sound(); // abstract method (no body)
```

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

```
class Dog extends Animal {  
    void sound() {  
        System.out.println("Dog barks.");  
    }  
}
```

□ Example 2: Using Interface

```
interface Vehicle {  
    void start(); // public + abstract by default  
}  
  
class Car implements Vehicle {  
    public void start() {  
        System.out.println("Car starts with key.");  
    }  
}
```

□ Real-Life Analogy:

Driving a Car

You don't need to know how the engine works — you just use the **steering, accelerator, and brakes** (interface).

□ Abstract Class vs Interface

Feature	Abstract Class	Interface
Methods	Can have abstract + concrete	Only abstract (Java 7), can have default/static (Java 8+)
Constructor	Can have	Can't have
Access Modifiers	Can be any	Only public for methods/fields
Multiple Inheritance	Not supported	Supported
Usage	When classes are closely related	When unrelated classes implement common behavior

Interview Points:

Question	Answer
Can abstract class have constructor?	Yes, but cannot instantiate directly
Can interface have constructor?	No
Can we declare an object of abstract class?	No, but we can have a reference
What is the difference between abstraction and encapsulation?	Abstraction hides implementation; encapsulation hides data
Can abstract class have final methods?	Yes, but not abstract and final together

OOPs Concept: Method Overriding in Java

Definition:

Method Overriding occurs when a **subclass** provides a **specific implementation** of a method that is **already defined** in its **parent class**.

It enables **runtime polymorphism** and allows Java to decide at **runtime** which method to execute.

Key Rules of Overriding

Rule #	Description
1.	Method name, return type, and parameters must match
2.	Only inherited methods can be overridden
3.	Access modifier must be same or more accessible

Rule #	Description
4.	Cannot override final, static, or private methods
5.	Use <code>@Override</code> annotation (recommended for readability & compile-time check)

Java Example:

```
class Animal {  
    void sound() {  
        System.out.println("Animal makes a sound");  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    void sound() {  
        System.out.println("Dog barks");  
    }  
}
```

Real-Life Analogy:

Think of a **general machine** that performs operations. A **printer**, a **scanner**, or a **fax machine** might override the "operate" function differently even though the base definition exists.

Difference: Overriding vs Overloading

Feature	Overriding	Overloading
Scope	Between base and derived class	Within the same class
Parameters	Must be same	Must be different

Feature	Overriding	Overloading
Return Type	Must be same or covariant	Can be same or different
Runtime/Compile	Runtime Polymorphism	Compile-time Polymorphism
Inheritance	Required	Not required

Interview-Level Questions:

Question	Answer
Can constructors be overridden?	No, constructors are not inherited
Can we override static methods?	No, static methods are class-level (hiding occurs)
Can we override private methods?	No, private methods are not inherited
Can return type differ?	Yes, if it's a covariant return type
Is overriding possible across packages?	Yes, if the method is protected or public

OOPs Concept: Inheritance in Java

Definition:

Inheritance is an OOP mechanism where **one class (child/subclass)** inherits the **properties and behaviors** (fields and methods) of another class **(parent/superclass)**.

Purpose:

- Achieve **code reusability**
- Support **hierarchical class structure**

- Enable **polymorphism**
-

□ **Syntax:**

```
class Parent {  
    // fields and methods  
}  
  
class Child extends Parent {  
    // additional fields and methods  
}
```

□ **Types of Inheritance in Java**

Type	Supported in Java?	Description
Single Inheritance	Yes	One class inherits from another
Multilevel	Yes	Chain of inheritance (A → B → C)
Hierarchical	Yes	Multiple classes inherit from the same parent
Multiple	Not via classes	A class inherits from multiple classes (Not supported due to ambiguity)

1. Single Inheritance

□ **Structure:**

Animal
↑
Dog

□ **Example:**

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

2. Multilevel Inheritance

Structure:

```
Animal  
↑  
Dog  
↑  
Puppy
```

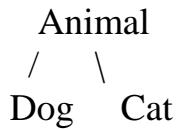
Example:

```
class Animal {  
    void eat() {  
        System.out.println("Eating");  
    }  
}  
  
class Dog extends Animal {  
    void bark() {  
        System.out.println("Barking");  
    }  
}  
  
class Puppy extends Dog {  
    void weep() {  
        System.out.println("Weeping");  
    }  
}
```

```
    }  
}
```

3.Hierarchical Inheritance

□ Structure:



□ Example:

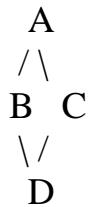
```
class Animal {  
    void eat() {  
        System.out.println("Eating");  
    }  
}  
  
class Dog extends Animal {  
    void bark() {  
        System.out.println("Barking");  
    }  
}  
  
class Cat extends Animal {  
    void meow() {  
        System.out.println("Meowing");  
    }  
}
```

4.Multiple Inheritance – Not Supported via Classes

□ Why Not?

Java doesn't support **multiple inheritance using classes** to avoid **ambiguity (Diamond Problem)**.

□ Diamond Problem:



If both B and C have a method show(), and D inherits both, which method should D use? ➤ **Ambiguity!**

Java's Solution:

Java supports multiple inheritance via **interfaces** (not classes).

Interface-Based Multiple Inheritance

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

- No ambiguity because C must implement the method.
-

□ Summary Table:

Inheritance Type	Supported in Java	Example Classes
Single	Yes	Dog extends Animal

Inheritance Type	Supported in Java	Example Classes
Multilevel	Yes	Puppy extends Dog extends Animal
Hierarchical	Yes	Dog, Cat extend Animal
Multiple (Classes)	No	Ambiguity in method resolution
Multiple (Interfaces)	Yes	C implements A, B

□ Interview Insights:

Question	Answer
Why doesn't Java support multiple inheritance using classes?	To avoid ambiguity (diamond problem)
How to achieve multiple inheritance in Java?	Using interfaces
Can a subclass override parent class methods?	Yes, using method overriding
What is the default superclass of every Java class?	java.lang.Object

OOPs Concept: super Keyword in Java

□ Definition:

The super keyword in Java is used to **refer to the immediate parent class** object.

It is commonly used when a subclass wants to:

- Call **parent class constructor**
- Access **parent class method**
- Access **parent class field**

Why use super?

Use Case	Purpose
Call parent constructor	Avoid duplication, ensure proper initialization
Access parent method	When child overrides parent method but still wants to use it
Access parent variable	When both child and parent have same variable name

Use 1: Access Parent Class Variables

```
class Parent {  
    int x = 100;  
}  
  
class Child extends Parent {  
    int x = 200;  
  
    void show() {  
        System.out.println(super.x); // Output: 100  
        System.out.println(this.x); // Output: 200  
    }  
}
```

Use 2: Call Parent Class Methods

```
class Parent {  
    void display() {  
        System.out.println("Parent class display()");  
    }  
}  
  
class Child extends Parent {  
    void display() {  
        System.out.println("Child class display()");  
    }  
}
```

```
void show() {  
    super.display(); // Calls parent class method  
    this.display(); // Calls child class method  
}  
}
```

□ **Use 3: Call Parent Class Constructor**

```
class Parent {  
    Parent() {  
        System.out.println("Parent constructor");  
    }  
}  
  
class Child extends Parent {  
    Child() {  
        super(); // Must be first statement in constructor  
        System.out.println("Child constructor");  
    }  
}
```

□ **Real-Life Analogy:**

Think of a child who calls their parent for help or refers to what the parent taught — that's how super works in code.

□ **Interview Insights:**

Question	Answer
Can super() be used after other statements in constructor?	No, it must be the first line
Can we call parent method using super if it is overridden?	Yes
Can super call private members of the parent class?	No, private members are not accessible

Question	Answer
Is super required if no constructor is defined in parent?	No, Java adds super() implicitly

OOPs Concept: Polymorphism in Java

Definition:

Polymorphism means "**many forms**". In Java, it allows **one interface** to be used for **different underlying forms (data types or objects)**.

Why Polymorphism?

- Increases **flexibility** and **reusability**
 - Supports **method behavior adaptation**
 - Enables **code generalization and decoupling**
-

Types of Polymorphism in Java:

Type	Also Known As	Achieved By
1. Compile-time	Static / Early Binding	Method Overloading
2. Runtime	Dynamic / Late Binding	Method Overriding

1. Compile-Time Polymorphism (Method Overloading)

Definition:

Multiple methods in the same class with **same name but different signatures** (parameters).

□ **Example:**

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

Key Rules:

- Same method name
 - Must differ in number/type/order of parameters
 - Cannot overload by return type only
-

2.Runtime Polymorphism (Method Overriding)

□ **Definition:**

When a **subclass provides specific implementation** of a method defined in the **parent class**.

□ **Example:**

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

```
void sound() {  
    System.out.println("Dog barks");  
}  
}
```

Real-Life Analogy:

Type	Analogy
Compile-time	Pressing different buttons on a calculator (same name: "add", different uses)
Runtime	A general remote controlling different brands of TVs (same interface, different behavior)

Polymorphism Interview Questions:

Question	Answer
Can we overload static methods?	Yes
Can we override static methods?	No (they're class-level, not object-level)
Can return types be different in overloading?	Yes, but only return type cannot be used to distinguish methods
Can constructors be overloaded?	Yes
What is dynamic method dispatch?	Runtime decision of which overridden method to call

OOPs Concept: Dynamic Method Dispatch in Java

Definition:

Dynamic Method Dispatch is the process by which a call to an **overridden method** is **resolved at runtime** rather than compile-time.

This is the **core mechanism of runtime polymorphism** in Java.

Key Objective:

To allow **method calls to behave differently** depending on the actual object type that the superclass reference is pointing to — **not the reference type itself**.

Java Example:

```
class Animal {  
    void sound() {  
        System.out.println("Animal makes sound");  
    }  
}  
  
class Dog extends Animal {  
    void sound() {  
        System.out.println("Dog barks");  
    }  
}  
  
class Cat extends Animal {  
    void sound() {  
        System.out.println("Cat meows");  
    }  
}
```

Usage of Dynamic Dispatch:

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

```
Animal a;      // Superclass reference  
  
a = new Dog(); // Reference to Dog object  
a.sound();    // Output: Dog barks  
  
a = new Cat(); // Reference to Cat object  
a.sound();    // Output: Cat meows  
}  
}
```

Even though the reference is of type Animal, the **method that gets executed depends on the object type** (Dog, Cat, etc.)

□ Real-Life Analogy:

Think of an **electric plug point (reference)** and different **devices (objects)** connected to it:

You use the same plug, but what happens (light turns on, fan runs, speaker plays) **depends on the device**.

□ Key Rules of Dynamic Dispatch:

Rule	Description
1	Works only with method overriding , not overloading
2	Reference type determines accessible members (variables)
3	Object type determines overridden method execution
4	Must use inheritance to enable overriding
5	Recommended to use @Override to prevent mistakes

Important Note:

```
class Parent {  
    int x = 10;  
    void show() {  
        System.out.println("Parent");  
    }  
}  
  
class Child extends Parent {  
    int x = 20;  
    void show() {  
        System.out.println("Child");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Parent obj = new Child();  
        obj.show();      // Output: Child (method resolved at runtime)  
        System.out.println(obj.x); // Output: 10 (variable resolved by reference type)  
    }  
}
```

Methods → Resolved by object type

Variables → Resolved by reference type

Interview-Level Questions:

Question	Answer
Is dynamic method dispatch possible without inheritance?	No
What's resolved at runtime in dynamic dispatch?	The overridden method
What happens to variables in dynamic	Resolved using reference type (compile

Question	Answer
dispatch?	time)
Is constructor dispatch dynamic?	No, constructor execution is determined at compile time
What's another name for dynamic dispatch?	Runtime polymorphism