Java OOP

Classes and objects

🔶 What is a Class?

A **class** is a *blueprint* or *template*.
Example:
A "Car" class describes:

- wheels
- color
- speed
- behaviours like accelerate(), brake()

But it is NOT a real car.

🔶 What is an Object?

An **object** is a real instance created from the class.
If Car is the blueprint →
Car car1 = new Car(); is an **actual car**.

🔶 The Three Things a Class Usually Contains

I want you to observe these 3 because interviewers love them:

1. **Fields (variables / state)**
2. **Methods (behaviour)**
3. **Constructors (initialization)**

We will cover constructors separately in the next topic.

Here is a basic example:

```
class Car {
    private String color;
    private int speed;

    public void accelerate() {
        speed += 10;
    }

    public void brake() {
        speed -= 10;
    }
}
```

And creating an object:

```
Car myCar = new Car();
myCar.accelerate();
```

Let’s check your understanding before we go deeper.
If Car is a blueprint, and you create **two objects**:

```
Car car1 = new Car();
Car car2 = new Car();
```

Do both objects share **the same variables** (color, speed),
or does each object get its **own copy** of those variables?

Each object gets **its own independent copy** of instance variables.
So:

- car1.speed and
- car2.speed

are separate values in memory.
That understanding is crucial before we move to **constructors**.

Constructors

You already know objects get their own data.
But **who sets the initial values** of that data?
➡️ **Constructors do.**
Let’s build intuition first.

🔶 What is a Constructor?

A constructor is a **special method** that:

- runs automatically when an object is created
- initializes the object
- has **no return type**
- has **same name as class**

Example:

```
class Car {
    private String color;
    private int speed;

    Car(String color) {
        this.color = color;
        this.speed = 0;
    }
}
```

Usage:

```
Car car1 = new Car("Red");
Car car2 = new Car("Black");
```

Each object now starts with different initial state.

✔ Constructors guarantee **valid initial state**
✔ Avoid **repetitive code**
✔ Prevent **bugs caused by forgetting to set required fields**

🔶 Types of Constructors in Java

Java gives you **three main types**:

1. **Default Constructor (no-arg constructor)**
2. **Parameterized Constructor**
3. **Copy Constructor** *(not built-in, but a common pattern)*

Let's go through them quickly, then I'll ask a check-in question.

1️⃣ Default Constructor

If you don't write any constructor, Java adds one automatically:

```
Car car = new Car();   // allowed only if no constructor is written
```

But **if you define ANY constructor**, the default disappears.

2️⃣ Parameterized Constructor

Used when you want initial values:

```
Car(String color, int speed) {
    this.color = color;
    this.speed = speed;
}
```

3️⃣ Copy Constructor

Not built into Java (unlike C++), but commonly used:

```
Car(Car other) {
    this.color = other.color;
    this.speed = other.speed;
}
```

Look at this class:

```
class User {
    private String name;
    private int age;

    User(String n) {
        name = n;
    }
}
```

Now someone writes:

```
User u = new User();
```

Will this compile?

Correct Answer: No, it will NOT compile.

Here's why:

💡 **Java gives you a default constructor ONLY if you do NOT write any**

constructor yourself.
But in your class:

```
User(String n) {
    name = n;
}
```

You **already wrote a constructor**.
➡️ So Java **does NOT** create a default no-arg constructor anymore.
➡️ Therefore:

```
new User();
```

❌ **will cause a compile-time error**
because the compiler looks for a constructor with **zero arguments** and cannot find one.

⭐ **To make it compile, you need to explicitly add a no-arg constructor:**

```
User() {
    // you may initialize default values here
}

User(String n) {
    name = n;
}
```

Now both are valid:

```
new User();         // OK
new User("Sagar"); // OK
```

Why does Java stop generating the default constructor when you add your own constructor?
Java assumes you want full control over how objects are initialized.

Public & Private Modifiers

🔶 Why do we even need access modifiers?

Because in object-oriented design:

- Some things should be **accessible openly**
- Some things should be **protected**
- Some things should be **totally hidden**

Just like:

- You can **start the car**
- But you cannot directly **modify its engine internals**

🔶 Java's Key Access Modifiers

For now we focus on the two most used in interviews:

1️⃣ private — strict encapsulation

- visible **only inside the same class**
- used to protect fields
- forces controlled access through methods

Example:

```
private int balance;
```

2️⃣ public — open access

- visible **everywhere**
- used for APIs, constructors, getters, setters, services, etc.

Example:

```
public void deposit(int amount) { ... }
```

Look at this class:

```
class BankAccount {
    public int balance;
}
```

Do you see any design problem with making balance **public**?

✔ Why public balance is dangerous?

Because anyone can do this:

```
account.balance = -999999;
```

or

```
account.balance = 0;  // wiping out the money
```

No validation, no security, no business logic.

This breaks:

- data safety
- business rules
- audit control
- encapsulation

So making it private is the only safe design.

🔶 So how should it really look?

```
class BankAccount {
    private int balance;

    public void deposit(int amount) {
        if(amount > 0) {
            balance += amount;
        }
    }
```

```
    public int getBalance() {
        return balance;
    }
}
```

Now:

- Balance is **protected**
- Access happens only through **public methods**
- All changes go through **validation & rules**

Inheritance

Inheritance means:
One class (child) **reuses and extends** the behavior of another class (parent).
Real-life example:
A **Car** *is a* **Vehicle** — it inherits the basic structure and adds more features.
Java example:

```
class Vehicle {
    int speed;
    void start() { }
}

class Car extends Vehicle {
    void playMusic() { }
}
```

Car gets:

- speed
- start()
- plus its own features like playMusic()

Correct! 🎯
A is exactly right.
When Car extends Vehicle:

- It inherits **public** and **protected** fields & methods
- It **does NOT inherit private members**
- It **can modify behavior** (this is *overriding*)
- It is **not forced** to duplicate fields from the parent

Perfect — you're thinking the right way.
Now we can move to the next important concept.

Method Overriding

🔶 What is Method Overriding?

When a **child class** provides its *own version* of a method that exists in the **parent class**, with:

- same name
- same parameters
- same return type (or covariant)

Example:

```
class Vehicle {
    void start() {
        System.out.println("Vehicle starting...");
    }
}

class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car starting with keyless
ignition...");
    }
}
```

Here, *Car* **modifies** the parent's behavior.

🔶 Why do we override methods?

Because sometimes the parent behavior is *too generic*.
Example:

- Every vehicle can "start"
- But a car, bus, or bike may start *differently*

🔶 Important Rules for Overriding

1. Same **method signature** (name + parameters)
2. Same **return type** (or covariant)
3. Cannot reduce visibility
   (e.g., public → private ❌)
4. Must be in **inheritance relationship**
5. Use @Override annotation
   - avoids mistakes
   - improves readability

What do you think happens when you run this code?

```
Vehicle v = new Car();
v.start();
```

Will it call:
A. Vehicle.start()
B. Car.start()

Car.start() is called

Even though the reference type is:
`Vehicle v`
the **actual object** is:
`new Car()`
So Java calls the **Car's overridden method**.
This behavior is called:

Runtime Polymorphism (Dynamic Method Dispatch)

and it is one of the MOST important fundamentals in OOP.

How Java Decides Which Method to Call

When you compile the code, the compiler checks:

- "Does the Vehicle class have a start() method?"
  ✔ Yes → compile success.

But **the compiler does NOT decide which method body will run.**

At runtime, Java checks the **actual object type**, not the reference type:

```
Reference type → Vehicle
Object type → Car
```

So JVM calls:

👉 the method of **Car**

This is called **dynamic dispatch**.

Virtual Method Table (V-Table)

Every Java object has a hidden pointer to a **v-table**, which stores:

- mapping of method names → actual method implementation for that object type

When you write:

```
Vehicle v = new Car();
v.start();
```

JVM does:

1. Look at the object (Car)
2. Look at Car's v-table
3. Find the overridden start() entry
4. Call Car.start()

This is why runtime polymorphism is powerful.

Best Practice for Overriding

Always use:

```
@Override
```

Example:

```
@Override
void start() {
    System.out.println("Car starts with push-button
ignition");
}
```

If you miss the method signature, the compiler will warn you.

Can you overload static methods?

→ YES

Because overloading is compile-time.

Can you override static methods?

→ NO

Static methods belong to the class, not the object.

ABSTRACT CLASS

An **abstract class** is:

✔ A class that:

- **cannot be instantiated**
- **can contain incomplete methods (abstract methods)**
- **can contain complete methods**
- **can have constructors**
- **can have state (fields)**
- **supports inheritance & partial implementation**

This makes it the perfect base for frameworks and template patterns.

🔥 Why can an abstract class have a constructor?

Because constructors help initialize **common shared state**, even if the class cannot be instantiated directly.

Example:

```
abstract class Vehicle {
    int speed;

    Vehicle(int speed) {
        this.speed = speed;
    }

    abstract void start();
}
```

Even though you cannot do:

```
new Vehicle();
```

You *can* and *must* call it through:

```
class Car extends Vehicle {
    Car() {
        super(100); // constructor of abstract class
    }
    void start() { }
}
```

🔶 Now let's understand abstract *methods*

ABSTRACT METHOD

A method with:

- no body
- no implementation
- must be overridden by child class

Example:

```
abstract void start();
```

🔥 Why do we use abstract methods?

Because **parent defines WHAT must be done**,
child defines **HOW it will be done**.
Example:

```
abstract class Payment {
    abstract void pay();
}

class UPI extends Payment {
    void pay() { System.out.println("Paid via UPI"); }
}
```

🧠 Memory & Runtime Behavior

An abstract class:

- **HAS** memory when part of a child object
- **Does NOT** get instantiated alone
- **Participates in v-table**
- Calls to overridden methods still go through **dynamic dispatch**

This is why:

```
Payment p = new UPI();
p.pay(); // calls UPI implementation
```

INTERFACES

Most developers only know the old-school definition:
“Interface is 100% abstract.”
But since **Java 8, 9, 14**, this has changed dramatically.

🔶 WHAT AN INTERFACE REALLY IS

An interface:

- defines **a contract** (what must be done)
- allows **multiple inheritance**
- can have **abstract methods**
- can have **default methods (Java 8+)**
- can have **static methods (Java 8+)**
- can have **private methods (Java 9+)**
- has **no constructors** (cannot store object state)

🟢 Example (Modern Interface)

```
interface Engine {
    void start(); // abstract method

    default void info() {
        System.out.println("This is an engine");
    }

    static void version() {
        System.out.println("Engine v1.0");
```

```
    }
}
```

🔥 INTERVIEW-LEVEL DIFFERENCES (ABSTRACT CLASS vs INTERFACE)

Feature	Abstract Class	Interface
State/fields	✔ Yes	❌ No (only public static final constants)
Constructors	✔ Yes	❌ No
Multiple inheritance	❌ No	✔ Yes
Default methods	✔ Can	✔ Can (Java 8+)
Private methods	✔ Yes	✔ Yes (Java 9+)
Purpose	Partial implementation	Pure contract

🔥 Deep Reality

Modern Java interfaces are **not 100% abstract** anymore.
This is why interviewers ask:
"If interfaces can have default methods, how are they different from abstract classes?"
You now know the deep answer:

- abstract classes **can have fields & constructors**
- interfaces **cannot maintain object state**
- interfaces allow **multiple inheritance**

Exactly — **B (Interface)** is correct.
Java allows **multiple inheritance of type** through interfaces:
`class Car implements Engine, MusicSystem { }`
But Java does **not** allow:
`class Car extends Vehicle, Machine  // ❌ not allowed`
This is foundational for how polymorphism works in Java.
You're now ready for the next major chapter.

POLYMORPHISM

Polymorphism = *many forms*
But most developers only know the surface definition.
Let's go **deep** the way a senior backend engineer understands it.

🔶 TWO TYPES OF POLYMORPHISM

1️⃣ Compile-Time Polymorphism (Static Binding)

→ Achieved by: **Method Overloading**

- Decided at compile time
- Based on reference type
- No v-table involved

Example:

```
void log(int x)
void log(String s)
```
Compiler selects method signature.

2️⃣ Runtime Polymorphism (Dynamic Binding)
→ Achieved by: **Method Overriding**
- Decided at runtime
- Based on object type
- Uses **v-table + dynamic dispatch**
- Enables framework-like behavior (Spring, Hibernate, etc.)

Example:
```
Animal a = new Dog();
a.speak(); // Dog.speak()
```

🔥 THE V-TABLE (Most developers don't know this!)
Every class with overridden methods has a **virtual method table**:
- Parent → Base entries
- Child → Overwrites entries
- JVM uses the v-table to call methods based on the actual object

This is why:
```
A obj = new C();
obj.show();
```
calls **C.show()**.

🔥 POLYMORPHISM + INTERFACE + ABSTRACT CLASS (Real Power)
Example:
```
Payment p = new UPI();
p.pay();
```
Here:
- Payment is an abstract type
- UPI is the implementation
- You get runtime polymorphism

This pattern is used everywhere in backend:
➡ Service layer
➡ DAO layer
➡ Strategy pattern
➡ Dependency Injection (Spring)

ENCAPSULATION
You already know the basic idea:
Hide data (private) + expose controlled access (public methods).
But here's the **full, enterprise-level meaning** used in real backend systems.

🔶 1️⃣ Encapsulation is NOT just hiding fields
It is about **maintaining invariants** inside an object.

Example: A BankAccount should never have:

- negative balance
- invalid account number
- unauthorized access

If fields were public:

```
account.balance = -99999;  // disaster
```

Encapsulation protects the object from entering illegal states.

🔶 2️⃣ Encapsulation is foundational for Security

It allows you to:

- validate data
- apply business rules
- protect internal logic
- comply with regulations (PCI, GDPR, RBI guidelines)

Backend systems rely heavily on encapsulation for data integrity.

🔶 3️⃣ Encapsulation enables loose coupling

Suppose you change how balance is stored:

```
private BigDecimal balance;
```

All consumers still use:

```
getBalance();
deposit();
withdraw();
```

→ No breaking changes
→ Safe evolution of codebase

🔶 4️⃣ Encapsulation makes frameworks possible

Spring does:

- proxying
- access control
- AOP
- dependency injection

Only because fields are private and behaviors go through methods.

🔶 5️⃣ Encapsulation + Abstraction work together

- **Abstraction**: *WHAT* can be done
- **Encapsulation**: *HOW* it is done safely

Example:

```
transactionService.transfer();
```

The caller knows WHAT will happen.
The service encapsulates HOW it happens securely.

Concept	Purpose	Key Insight

Abstract Class	Partial implementation	Use when you need fields + constructors + shared logic
Interface	Contract	Allows multiple inheritance; modern interfaces can have default + static methods
Polymorphism	Flexible behavior	Overriding = runtime (dynamic dispatch), Overloading = compile-time
Encapsulation	Protection + safety	Keep state private; expose behavior through methods

To summarize the decision rule:
✔ Choose INTERFACE when:

- You want a behaviour contract
- Child types don't share internal fields
- Multiple inheritance is useful
- You want flexibility + loose coupling

✔ Choose ABSTRACT CLASS when:

- Child types share internal fields
- You want shared constructors
- You want some default implementations + mandatory ones
- You want a partially built base class

SCANNER CLASS

Most beginners think Scanner is "just for input," but let's understand it like a backend engineer.

✔ What Scanner actually does

Scanner is a **tokenizing parser** that reads input and breaks it into meaningful pieces (tokens).
It can read from:

- System.in (keyboard)
- A String
- A File
- A Path
- A URLConnection input stream
  ...and more.

Example:

```
Scanner sc = new Scanner(System.in);
```

```
int age = sc.nextInt();
String name = sc.next();
```

🔥 Deep Insight: Scanner uses regex internally

Scanner splits input using a **delimiter pattern** (default: whitespace).
You can even change the delimiter:

```
sc.useDelimiter(",");
```

This makes Scanner useful for:

- parsing CSV
- reading structured input
- reading logs
- token-based parsing

🔥 Important Scanner Pitfall (Interview & Real Life)

Using nextInt + nextLine back-to-back:

```
int age = sc.nextInt();
String name = sc.nextLine();  // ← often skipped!
```

Why does this happen?
Because nextInt() does *not* consume the newline \n.
Solution:

```
sc.nextLine(); // consume leftover newline
String name = sc.nextLine();
```

instanceof Operator

Most people think instanceof simply checks type.
But let's understand it at JVM-level.

🔶 What instanceof actually does

It checks:

1. **Actual object type** (not reference type)
2. **Class inheritance hierarchy**
3. **Interface implementation**

Example:

```
Car c = new Car();
System.out.println(c instanceof Vehicle);   // true
System.out.println(c instanceof Car);       // true
System.out.println(c instanceof Object);    // true
```

Even interfaces:

```
System.out.println(c instanceof Engine); // true if Car
implements Engine
```

🔥 instanceof + polymorphism (crucial)

```
Vehicle v = new Car();
System.out.println(v instanceof Car);  // true
System.out.println(v instanceof Vehicle); // true
```

This confirms:

- Compile-time reference does NOT matter
- Runtime object type decides the result

🔥 instanceof and Null

Very important interview question:

```
Car c = null;
c instanceof Car     // false
```

Why?

Because null is considered **not an instance of anything**.

🔥 instanceof After Java 14 — Pattern Matching

Modern Java adds pattern matching:

```
if (obj instanceof String s) {
    System.out.println(s.toLowerCase());
}
```

This combines **type check + cast** in one step.

The super Keyword

Most beginners only know:

- super refers to parent class

But as a senior Java engineer, you must know **all 3 roles** of super and the deeper behavior behind it.

🔶 1️⃣ super() → Calls Parent Constructor

When creating a subclass object:

```
class A { A(){ System.out.println("A"); } }
class B extends A { B(){ System.out.println("B"); } }
```

Creating:

```
new B();
```

prints:

```
A
B
```

✔ Why?

Because Java must initialize **parent part of the object first**.

Memory layout:

```
[A fields]
[B fields]
```

Parent exists *inside* the child, so it must be initialized first.

If you don't write super(), Java inserts it automatically.

🔶 2️⃣ super.methodName() → Calls Parent Class Method

Used when:

- child overrides a method
- but still wants parent's behavior

Example:

```
class Car extends Vehicle {
```

```
    @Override
    void start() {
        super.start();  // call parent's start
        System.out.println("Car start logic");
    }
}
```

🔶 3️⃣ Accessing Parent Fields

If child hides parent fields:

```
class A { int x = 10; }
class B extends A { int x = 20; }
```

Then:

```
System.out.println(super.x); // 10
```

🔥 Deep Insight: super is NOT a reference

It is resolved at **compile time**, not runtime.

Meaning:

- this participates in polymorphism
- super does NOT

✅ Why does Java call the parent constructor first even when we don't write super()?

Let's use your exact example:

```
class A {
    A() {
        System.out.println("A");
    }
}

class B extends A {
    B() {
        System.out.println("B");
    }
}
```

And then:

```
new B();
```

Output:

A

B

But you **never** wrote:

```
super();
```

So why does Java still call the parent constructor?

🔥 Deep Explanation: Java *automatically inserts* super() at compile time

Inside every constructor of a child class, Java secretly adds the FIRST line:

`super();`
if you don't write any super/this constructor call.
So the compiler **rewrites** your code:

Your code:

```
B() {
    System.out.println("B");
}
```

Compiler's version:

```
B() {
    super();               // ← injected by compiler
    System.out.println("B");
}
```

🔶 Why does Java force this behavior?

✔ Reason 1:

A child object **contains** a parent object inside it.
Memory is structured like this:

```
[A fields]
[B fields]
```

So the **parent part must be created first**.

✔ Reason 2:

Parent constructor is responsible for:

- initializing parent fields
- setting up behavior the child depends on
- preparing the object structure

If parent does not initialize properly, the child cannot exist safely.

✔ Reason 3:

This rule guarantees that **object creation order is ALWAYS predictable**, even if programmer forgets to write constructor chaining.

🔶 Important rule (Interview-Level)

In Java:
A constructor MUST either call super() or this() as the first statement.
If you write neither, the compiler inserts super() for you.
This is why:

- A() runs first
- then B() runs

even though you didn't explicitly write super().

🔶 What if parent has ONLY a parameterized constructor?

Important corner case:

```
class A {
    A(int x) { }
}

class B extends A {
```

```
    B() {
        System.out.println("B");
    }
}
```
This will NOT compile.
Why?
Because the compiler inserts:
```
super();   // but A() does NOT exist!
```
So you must explicitly write:
```
B() {
    super(10);
    System.out.println("B");
}
```
This proves Java ALWAYS inserts super() unless told otherwise.

static Keyword
Most developers only know:
- static = belongs to class, not object

But we will go far deeper:

🔶 1️⃣ What static really means
A static member belongs to the **class**, loaded into memory once, stored in the **Method Area / MetaSpace** of the JVM.
This includes:
- static variables
- static methods
- static blocks
- static nested classes

🔶 2️⃣ static variables → Shared across ALL objects
Example:
```
class Counter {
    static int count = 0;
    Counter() { count++; }
}
```
If you write:
```
new Counter();
new Counter();
new Counter();
```
count becomes 3.
Why?
Because there is **only ONE copy** of count.

🔶 3️⃣ static methods cannot use non-static variables
Why?
Because static methods run **without any object**, so Java has no this reference.

This will cause error:

```
class A {
    int x = 10;
    static void test() {
        System.out.println(x); // ❌ error
    }
}
```

🔶 4️⃣ static blocks — executed once when class loads

```
static {
    System.out.println("Class loading...");
}
```

Used for:

- loading native libraries
- initializing JDBC drivers
- setting global config

🔶 5️⃣ static + memory model (Deep)

Java loads static members when:

- class is loaded by ClassLoader
- stored in **Metaspace**
- shared across JVM threads

This is why static members are used for:

- caching
- constants
- utility methods
- singletons

final Keyword (Deep, Complete, Interview-Level Understanding)

Most developers only know *"final means cannot change"*.
But there are **3 very different uses** of final.
Let's go deep one by one.

🔷 1️⃣ final Variables — cannot be reassigned

```
final int x = 10;
x = 20;   // ❌ compile error
```

✔ BUT the object itself can still change if it is mutable:

```
final ArrayList<Integer> list = new ArrayList<>();
list.add(10);   // ✔ allowed
list = new ArrayList<>();  // ❌ not allowed
```

Meaning:

- the **reference** cannot change
- the **object** may still change

🔷 2️⃣ final Methods — cannot be overridden

Used to prevent subclasses from changing behavior.

```
class A {
    final void show() { }
}

class B extends A {
    void show() { }   // ❌ error
}
```

Why is this useful?

- to prevent breaking important logic
- enforce a stable API
- used in security-sensitive code

Example:
String class uses **final methods** heavily.

🔷 3️⃣ final Classes — cannot be extended

```
final class Vehicle { }

class Car extends Vehicle { }  // ❌ error
```

Why?
Because some classes must be **immutable** or **security-critical**.
Examples of final classes:

- String
- Integer, Double, Float
- LocalDate, LocalTime
- Many classes in java.lang

🔷 4️⃣ final + constructor → Immutability

If you make fields final **and initialize them in constructor**, object becomes immutable:

```
class User {
    final String name;

    User(String name) {
        this.name = name;
    }
}
```

Now name cannot be changed.
This idea is the foundation of:

- String immutability
- Record classes
- Thread-safe designs
- Functional programming patterns

🔷 5️⃣ final vs effectively final (Advanced Concept)

Java lambda expressions require variables to be **effectively final**, not necessarily declared final.

```
int x = 10;   // effectively final
Runnable r = () -> System.out.println(x);
```

If you try to modify it:

```
x = 20;   // ❌ now not effectively final
```

This is a common interview question.