Object-Oriented Programming (OOP) is the heart of Java.

**1. Encapsulation**

* **Definition**: Wrapping data (variables) and code (methods) together in one unit (a class).
* **Goal**: Hide internal details; expose only what’s necessary using *getters* and *setters*.
* **Encapsulation is about protecting data**. It ensures that an object’s internal representation is hidden from the outside and can only be accessed through public methods (getters and setters).
* Real-world Analogy: **Think of a capsule** — it contains medicine inside, and you can't open it and mess with the ingredients directly. You just take it as instructed. In code, we hide the variables (private) and provide controlled access (public methods).
* **Interview Twist: They may ask: “What if you don’t use getters/setters?”**Answer: Without them, you expose your data directly, which breaks encapsulation and can lead to bugs, because anyone can modify the values freely.
* **Example**:

class Student {

private String name; // hidden from outside

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

**2. Inheritance**

* **Definition**: One class (child) inherits the fields and methods of another (parent).
* **Keyword**: extends
* **Goal**: Code reusability
* **Real-world Analogy:** Imagine your dad is a musician. You inherit his guitar skills, so you don’t need to learn from scratch.
* **Interview Twist: They might ask: “What is the problem with multiple inheritance?”**  
  Answer: Java doesn't support multiple inheritance with classes to avoid the diamond problem. It supports it through interfaces.
* What is the Diamond Problem?
* The diamond problem occurs in languages that support multiple inheritance with classes (like C++). It refers to an ambiguity that arises when a class inherits from two classes that both inherit from a single base class.

A

/ \

B C

\ /

D

* **Class A is the base class**.
* **Classes B and C inherit from A.**
* **Class D inherits from both B and C.**
* If both B and C override a method from A, and D doesn't override it again, then… 😵‍♂️ Which version of the method should D inherit? That's the ambiguity—the "diamond problem."
* 🟨 In Java?
* Java avoids this problem by not allowing multiple inheritance with classes. That means: ⛔ A class can’t extend more than one class.
* ✅ But Java does allow multiple inheritance with interfaces.
* To handle ambiguity in interfaces, Java requires you to resolve it explicitly in your code.

**Interview Insight:**

**If you're asked: 🗣️ "Does Java face the diamond problem?" You can say:**

* "**No, Java prevents it with classes by disallowing multiple inheritance**. With interfaces, Java provides default methods and forces the implementing class to resolve any ambiguity."
* **Example of inheritance**:

class Animal {

void sound() {

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

}

}

class Dog extends Animal {

void bark() {

System.out.println("Dog barks");

}}

**3. Polymorphism**

* **Definition**: One thing behaves in many ways.
* **Polymorphism means “many forms**.” A method can behave differently depending on the object that invokes it.
* Real-world Analogy: The word “**run**” can mean running a race or running a company. Same word, different actions.
* **Types**:
  + **Compile-time** (Method Overloading)
  + **Runtime** (Method Overriding)
* Interview Twist: Q: “**Can static methods be overridden?”** A: No. Static methods belong to the class, not instances. So they can be hidden, but not overridden.
* **Example (Overloading)**:

class MathOps {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

}

* **Example (Overriding)**:

Abstract class Animal {

void sound() ;

}

class Dog extends Animal {

void sound() {

System.out.println("Dog barks");

}}

**4. Abstraction**

* **Definition**: Hiding complex details and showing only essential features.
* Abstraction hides unnecessary details and shows only the essential information.
* **Real-world Analogy:** When you use a phone, you don’t see the wiring inside. You just use the buttons.
* Interview Twist: Q: “**Why use interfaces when we have abstract classes?”** A: Interfaces allow multiple inheritance and define a strict contract. Abstract classes are for shared behavior + abstraction.
* **Achieved by**:
  + **Abstract class** (partial abstraction)
  + **Interface** (full abstraction in older Java)
* **Example**:

abstract class Animal {

abstract void sound(); // no body

void sleep() {

System.out.println("Sleeping...");

}

}

class Cat extends Animal {

void sound() {

System.out.println("Meow");

}

}

**Follow-Up Interview Questions:**

* **What’s the difference between abstraction and encapsulation**?

| **Feature** | **Abstraction** | **Encapsulation** |
| --- | --- | --- |
| Focus | Hiding implementation complexity | Hiding internal data |
| Purpose | Show only essential features | Protect and restrict access |
| How it’s done | Interfaces & abstract classes | private fields + getters/setters |
| Example | abstract void draw(); | private int age; public setAge(); |

* **What happens if you override a method and throw a different exception?**

In Java, when you override a method, you must follow certain rules regarding the exceptions you throw.

Let's look at two scenarios:

🟩 1. ✅ If the overridden method throws a **checked exception**:

You can:

* Throw the same exception
* Throw a subclass of that exception
* Not throw anything at all

But you cannot:

* Throw a new or broader checked exception

🟦 2. ✅ If the overridden method does NOT throw a checked exception:

* You cannot throw any new checked exceptions in the subclass override.
* **Can we achieve polymorphism without inheritance?**

✅ Yes — but it depends on what kind of polymorphism you're talking about.

There are two main types of polymorphism in Java:

| **Type of Polymorphism** | **Inheritance Needed?** |
| --- | --- |
| Method Overloading | ❌ No |
| Method Overriding | ✅ Yes |

* Can you give an example of interface vs abstract class in a real project?
* Explain method hiding vs overriding.

**Let’s do a round of OOPS tricky Java interview questions**

1. Can a class be both abstract and final in Java?

💡 Answer:

**No. It will result in a compile-time error.**

✅ Explanation:

* abstract → means the class is incomplete and needs to be subclassed.
* final → means the class cannot be subclassed.

So they directly conflict.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2. **What will be the output?**

class A {

int i = 10;

}

class B extends A {

int i = 20;

}

public class Test {

public static void main(String[] args) {

A obj = new B();

System.out.println(obj.i);

}

}

💥 Trick: Variable hiding, not overriding

🧩 Output: 10

✅ Explanation: Even though obj is a B object, it’s referenced as type A. Variables are resolved at compile time using reference type, not object type — unlike overridden methods.

🧠 Interview Insight: Shows if you understand field hiding vs method overriding.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

3. **Which one will compile successfully?**

A.

abstract class Animal {

abstract void sound();

}

B.

abstract class Animal {

abstract void sound() {

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

}

}

💥 Trick: Can abstract method have body?

🧩 Answer: Only A will compile.

✅ Explanation: An abstract method cannot have a body. If you want a default implementation, remove abstract and make it a normal method.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

4. **Can an abstract class have a constructor?**

💥 Trick question

🧩 Answer: Yes

✅ Explanation: Abstract classes can have constructors. They’re used to initialize common fields when subclass constructors are called.

Example:

abstract class Shape {

Shape() {

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

}

}

5. What will this print?

class A {

A() {

print();

}

void print() {

System.out.println("A");

}

}

class B extends A {

int x = 1;

void print() {

System.out.println(x);

}

}

public class Test {

public static void main(String[] args) {

new B();

}

}

💥 Trick: Constructor calling overridden method

🧩 Output: 0

✅ Explanation: During A's constructor, print() is called. But the version in B is executed. At that point, x is not yet initialized, so default value 0 is printed.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

6. What are the 4 pillars of OOP?

✅ Answer:

The four main pillars are:

1. **Encapsulation** – Wrapping data and methods into a single unit (class) and restricting direct access to internal data using access modifiers.
2. **Abstraction** – Hiding internal implementation details and showing only essential features.
3. **Inheritance** – Acquiring properties and behaviors from a parent class.
4. **Polymorphism** – Ability to take many forms. Achieved via method overloading and overriding.

💬 Tip: Use real-world analogies — like a car for abstraction (you use it without knowing how it works inside).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

7. What is the difference between **abstraction** and **encapsulation**?

✅ Answer:

* Abstraction is about hiding unnecessary implementation details from the user.
* Encapsulation is about binding data and methods together and restricting access.

🧠 Analogy:

* Abstraction: You see only the steering wheel, not how the engine works.
* Encapsulation: You can’t directly access the engine parts; you go through controls (getters/setters).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

8. What is method **overloading** vs method **overriding**?

✅ Answer:

* Overloading: Same method name, different parameter list, within the same class. (Compile-time polymorphism)
* Overriding: Same method name and signature in subclass — provides new behavior. (Runtime polymorphism)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

9. What are access modifiers and how do they support encapsulation?

✅ Answer:

Access modifiers define the visibility of variables, methods, and classes. Java provides:

* private – accessible only within the same class
* default – within the same package
* protected – same package and subclasses
* public – accessible everywhere

Encapsulation uses private fields + public getter/setter methods to control access.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

10. What is constructor overloading?

✅ Answer:

**Constructor overloading** means having multiple constructors in a class with different parameter lists. It allows different ways to initialize an object.

Example:

class Book {

Book() {}

Book(String title) {}

Book(String title, int pages) {}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

11. Why do we use interfaces in Java?

✅ Answer:

* Interfaces support 100% abstraction (before Java 8).
* They help achieve multiple inheritance (which classes can't do).
* They define contracts for what a class must do, not how.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

12. What is **dynamic method dispatch**?

✅ Answer:

It is the process in which a call to an overridden method is resolved at runtime, not compile time.

Used in: Polymorphism (method overriding)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

13. What is the use of the **super** keyword?

✅ Answer:

* super() is used to call the parent class constructor.
* super.methodName() is used to call a parent method when overridden.
* Used to resolve method/field hiding issues in inheritance.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

14. Can you instantiate an abstract class?

✅ Answer: No, you cannot instantiate an abstract class directly. But you can:

* Create a subclass and instantiate it.
* Use an anonymous class to provide implementation on the spot.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

15. Why is Java not considered 100% Object-Oriented?

✅ Answer:  
Because it supports **primitive** data types like int, float, char etc., which are not objects.

**Scenario-Based OOP Interview Questions**

1. **Design a system to model an online library. Which OOP concepts will you use?**

✅ Sample Answer:

* Class: Book, User, Librarian
* Inheritance: User → Librarian (special kind of user)
* Encapsulation: Book details (title, author) made private with getters/setters
* Abstraction: Interface Searchable for searching books
* Polymorphism: Overload search methods (by title, by author)

1. **How would you model a banking system using OOP?**

✅ Sample Approach:

* Class: Account (base), SavingsAccount, CurrentAccount
* Inheritance: Account → SavingsAccount, CurrentAccount
* Polymorphism: Method calculateInterest() overridden in each type
* Encapsulation: balance is private with getBalance(), deposit(), withdraw()

1. **You’re designing a Hospital Management System. What classes and OOP concepts would you apply?**

✅ Answer:

* Class: Patient, Doctor, Appointment, Department
* Inheritance: Person → Patient, Doctor
* Encapsulation: Patient records kept private
* Polymorphism: scheduleAppointment() might behave differently for new and follow-up cases

**unexpected but common” OOP questions that often surprise candidates:**

**Q1. Can a Java class implement multiple interfaces? Why is this allowed, but multiple inheritance isn’t?**

✅ Answer: Yes, a class can implement multiple interfaces.  
Java doesn’t allow multiple inheritance with classes to avoid the diamond problem, but interfaces don't hold state (only method signatures), so there's no ambiguity in inheritance.

**Q2. Can we override private or static methods?**

✅ Answer: No.

* private methods are not visible in the child class → not overridden.
* static methods are class-level → they can be hidden, not overridden.

**Q3. What is method hiding?**

✅ Answer: If a subclass defines a static method with the same signature as a static method in the parent class, it’s called method hiding, not overriding.

**Q4. Can a constructor be final, static, or abstract?**

✅ Answer: No.

* final → makes no sense, as constructors aren’t inherited
* static → constructors are tied to object creation, not class
* abstract → constructors must be implemented, so abstract is invalid

**Q5. Why are constructors not inherited?**

✅ Answer: Constructors are specific to the class they are defined in.  
Each class defines how its own objects are initialized — and allowing inheritance could lead to incorrect initialization.

**Q6. What is the difference between composition and inheritance? Which one is better?**

✅ Answer:

* Inheritance: “is-a” relationship → tightly coupled
* Composition: “has-a” relationship → loosely coupled

Prefer composition over inheritance when possible (for flexibility and testability).