**OOPs Interview Questions**

# **Basics**

## Q1. What are the four main principles of Object-Oriented Programming (OOP)?

The four main principles of Object-Oriented Programming (OOP) are:

1. **Encapsulation** – Bundling data and methods that operate on that data within a class, and restricting direct access to some of the object's components.
2. **Abstraction** – Hiding complex implementation details and showing only essential features to the user.
3. **Inheritance** – Enabling a new class to acquire the properties and behaviors of an existing class.
4. **Polymorphism** – Allowing objects to be treated as instances of their parent class, with behavior determined at runtime (dynamic) or compile-time (static).

## Q2. What is difference between method overloading and overriding.

| **Feature** | **Method Overloading** | **Method Overriding** |
| --- | --- | --- |
| **Definition** | Same method name with different parameters | Subclass redefines a method of the parent class |
| **Class** | Happens within the same class | Involves parent and child classes |
| **Parameters** | Must differ in type, number, or order | Must be exactly same as in parent class |
| **Return Type** | Can be different | Must be same or covariant |
| **Inheritance Required** | No | Yes |
| **Binding Type** | Compile-time (static binding) | Runtime (dynamic binding) |
| **Purpose** | To increase method flexibility | To provide specific implementation in subclass |

## Q3. What are access modifiers in Java?

Access modifiers in Java define the visibility/scope of classes, methods, and variables. There are **four** main access modifiers:

1. **private** – Accessible **only within the same class**.
2. **default** (no modifier) – Accessible **within the same package**.
3. **protected** – Accessible **within the same package** and by **subclasses outside the package**.
4. **public** – Accessible **from anywhere** in the program.

These control how other classes interact with your code, supporting encapsulation.

## Q4. Why is Java not considered a purely object-oriented language?

Java is **not considered a purely object-oriented language** for the following reasons:

1. It supports **primitive data types** (int, char, boolean, etc.) which are **not objects**.
2. It allows the use of **static members** (methods and variables) which belong to the class rather than objects.

These features break the "everything is an object" principle of pure object-oriented languages.

## Q5. Can you override a static method in Java?

**No**, you **cannot override** a static method in Java.

Static methods belong to the **class**, not instances. If you define a static method with the same signature in a subclass, it is called **method hiding**, **not overriding**.

So, polymorphism doesn’t apply to static methods.

## Q6. What is the difference between == and .equals() in Java?

In Java:

* == checks **reference equality** – whether two references point to the **same object** in memory.
* .equals() checks **value/content equality** – whether two objects have **equivalent values** (can be overridden in custom classes).

String a = new String("Hello");  
String b = new String("Hello");  
  
a == b // false (different objects)  
a.equals(b) // true (same content)

## Q7. What is the difference between an abstract class and an interface in Java?

An **abstract class** in Java can have both abstract (unimplemented) and concrete (implemented) methods. It can also have instance variables, constructors, and any access modifiers. It’s used when classes share common code or state, and supports only single inheritance.

An **interface**, on the other hand, defines a contract with method declarations that classes **must implement**. From Java 8 onward, it can also have default and static methods. Interfaces are used to achieve multiple inheritance and provide flexibility in defining capabilities across unrelated classes.

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Methods** | Can have abstract and concrete methods | From Java 8, can have default, static, and abstract methods |
| **Fields** | Can have instance variables | Only public static final (constants) |
| **Inheritance** | Supports **single inheritance** only | Supports **multiple inheritance** |
| **Constructor** | Can have constructors | Cannot have constructors |
| **Access Modifiers** | Methods can have any access modifier | All methods are implicitly public |
| **Use Case** | When there’s a base class with shared code | When defining a contract or capability |

## Q8. Can abstract classes have constructor?

Yes, **abstract classes can have constructors** in Java.

Although you cannot create an instance of an abstract class directly, its constructor is called when a subclass object is created. This helps initialize common fields or perform setup tasks for the abstract class part of the subclass.

## Q9. Explain the concept of multiple inheritance and how Java handles it.

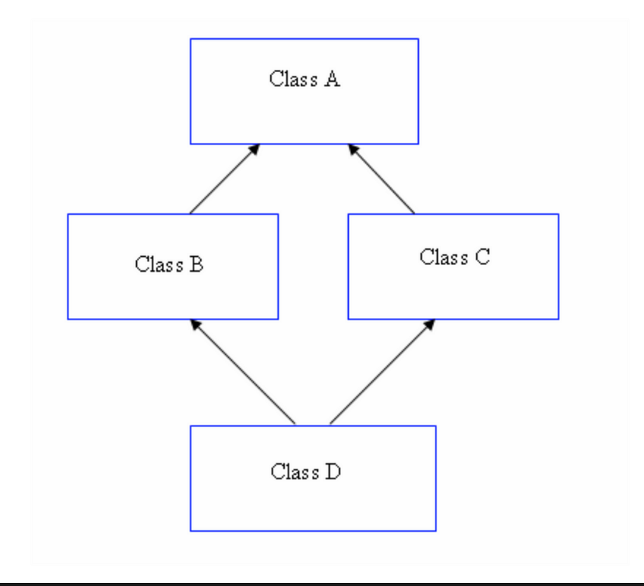
**Multiple inheritance** means a class inherits from more than one parent class.

Java **does not support multiple inheritance with classes** to avoid complexity and ambiguity (like the diamond problem). Instead, Java achieves multiple inheritance through **interfaces**, allowing a class to implement multiple interfaces and inherit their abstract methods without conflicts.

## Q10. What is diamond problem, and how java handles it.

The **Diamond Problem** occurs in multiple inheritance when a class inherits from two classes that both inherit from a common superclass, leading to ambiguity about which superclass method to use.

Java avoids this problem with classes by disallowing multiple class inheritance but allows multiple interface inheritance with default methods, which can cause a diamond problem-like situation. In such cases, the subclass must **override the conflicting method** to resolve ambiguity.



## Q11. Can a class inherit the constructor of it's base class?

**No**, a class in Java **does not inherit constructors** from its base class.

However, a subclass can **call the base class constructor** using super(), either implicitly or explicitly, to initialize the superclass part of the object. Constructors are not inherited but can be reused via super.