**OOPs Interview Questions**

# **Standard**

## 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.

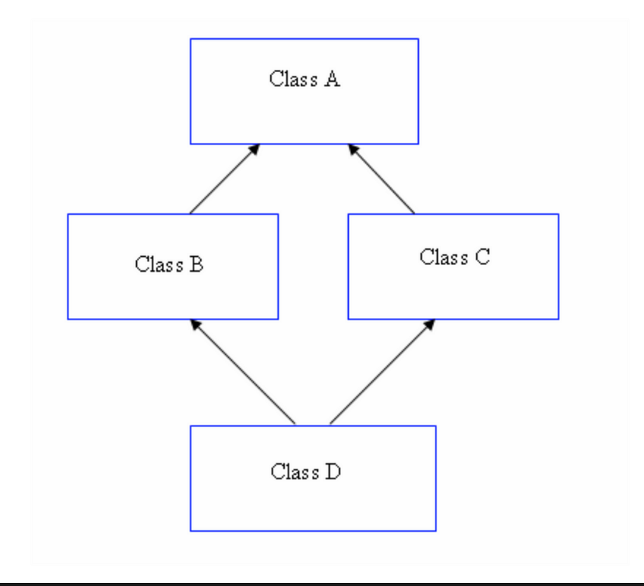
**Multiple inheritance** means a class inherits from more than one parent class.

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.

## Q12. What is the difference between final, finally, and finalize in Java?

- **final** is a keyword used to declare constants, prevent method overriding, or inheritance.  
- **finally** is a block used in exception handling to execute code after a try-catch block.  
- **finalize** is a method used for cleanup operations before an object is garbage collected.

## Q13. What is the difference between throw and throws in Java?

- **throw** is used within a method to throw an exception explicitly.  
- **throws** is used in a method signature to declare that the method might throw certain exceptions.

# **Advanced**

## Q1. What are inner classes?

**Inner classes** in Java are classes defined **within another class**. They are used to logically group classes and can access members (including private) of the outer class.

Types of Inner Classes:

* **Non-static inner class**
* **Static nested class**
* **Local inner class** (defined inside a method)
* **Anonymous inner class**

class Outer {  
 int x = 10;  
  
 class Inner {  
 void display() {  
 System.*out*.println("x = " + x); // Can access outer class variable  
 }  
 }  
  
 public static void main(String[] args) {  
 Outer outer = new Outer();  
 Outer.Inner inner = outer.new Inner();  
 inner.display(); // Output: x = 10  
 }  
}

## Q2. How inner class is created inside a method

An **inner class inside a method** is called a **local inner class**. It is defined **within a method body**, and it can only be accessed **within that method**.

public class Outer {  
 void outerMethod() {  
 // Local inner class  
 class Inner {  
 void display() {  
 System.*out*.println("Hello from local inner class!");  
 }  
 }  
  
 // Creating and using the inner class  
 Inner inner = new Inner();  
 inner.display();  
 }  
  
 public static void main(String[] args) {  
 Outer outer = new Outer();  
 outer.outerMethod();  
 }  
}

* Local inner class **cannot be accessed outside** the method.
* It **can access final or effectively final variables** from the enclosing method.

## Q3. What is the significance of the final keyword in Java?

The final keyword in Java is used to **restrict modification**. Its significance depends on where it's used:

1. **final variable** – Value cannot be changed once assigned (acts like a constant).
2. **final method** – Cannot be overridden by subclasses.
3. **final class** – Cannot be inherited (extended) by any class.

It helps in creating **immutable entities**, ensuring **security, consistency, and clarity** in code.

## Q4. How to create immutable class in java?

Immutability means **once an object is created, its state (data) cannot be changed**.

**How to Create an Immutable Class**:

* Mark the class as final (so it can’t be subclassed).
* Make all fields private and final.
* Initialize all fields via constructor.
* Don’t provide setters.
* If a field is mutable (like a List), return a **defensive copy** in the getter.

import java.util.List;  
import java.util.ArrayList;  
import java.util.Collections;  
  
public final class Person {  
 private final String name;  
 private final int age;  
 private final List<String> hobbies; // Mutable field  
  
 public Person(String name, int age, List<String> hobbies) {  
 this.name = name;  
 this.age = age;  
 // Defensive copy in constructor  
 this.hobbies = new ArrayList<>(hobbies);  
 }  
  
 public String getName() { return name; }  
  
 public int getAge() { return age; }  
  
 public List<String> getHobbies() {  
 // Defensive copy in getter  
 return new ArrayList<>(hobbies);  
 }  
}

## Q5. What is difference between deep copy and shallow copy?

| **Copy Type** | **Description** |
| --- | --- |
| **Shallow Copy** | Copies the **reference** to objects – both objects share the same nested objects. |
| **Deep Copy** | Copies the **actual objects recursively** – changes in one do not affect the other. |

## Q6. Explain the use of the instanceof operator.

object instanceof ClassName

It returns true if the object is an instance of the class (or its subclass), otherwise false

* Commonly used for **type checking** before casting.
* Helps avoid ClassCastException.

class Animal {}  
class Dog extends Animal {}  
  
public class Test {  
 public static void main(String[] args) {  
 Animal a = new Dog();  
  
 System.*out*.println(a instanceof Dog); // true  
 System.*out*.println(a instanceof Animal); // true  
 System.*out*.println(a instanceof String); // Compilation error  
 }  
}

## Q7. What will get executed first, static block or main method

The **static block executes first**, **before** the main() method, because static blocks are run **once when the class is loaded**.

public class Demo {  
 static {  
 System.*out*.println("Static block executed");  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println("Main method executed");  
 }  
}  
//Static block executed   
//Main method executed

## Q8. Why default methods were introduced in interfaces?

Default methods were introduced to:

1. **Add new methods to interfaces without breaking existing implementations.**  
   ➤ Before Java 8, adding a method to an interface would break all implementing classes.
2. **Provide method implementations directly in interfaces.**  
   ➤ Useful for **backward compatibility** and sharing common code.

## Q9. What will happen if class C implements A,B interfaces, both having same default method and C wants to use that method?

If class C implements two interfaces A and B, **both having the same default method**, then **C must override that method**, otherwise the compiler will throw an error due to **ambiguity**.

interface A {  
 default void show() {  
 System.*out*.println("A's default show");  
 }  
}  
  
interface B {  
 default void show() {  
 System.*out*.println("B's default show");  
 }  
}  
  
class C implements A, B {  
 // Must override show() to resolve ambiguity  
 public void show() {  
 A.super.show(); // or B.super.show()  
 }  
  
 public static void main(String[] args) {  
 new C().show(); // Output: A's default show  
 }  
}

## Q10. How does Java handle memory management for objects? How Garbage collection works here?

Java handles memory management automatically using the **Java Memory Model (JMM)** and **Garbage Collector (GC)**.

**🔹 Key Memory Areas:**

1. **Heap**
   * Stores all **objects and their instance variables**.
   * Shared across all threads.
   * Objects created using new are stored here.
2. **Stack**
   * Stores **method calls, local variables, and references** to objects in the heap.
   * Each thread has its own stack.

🔹 **Garbage Collection (GC):**

Java automatically removes **unreachable objects** from the heap using **Garbage Collector**.

* **Mark-and-Sweep**: Marks reachable objects and sweeps out unreferenced ones.
* **Generational GC**: Divides heap into Young, Old, and Permanent generations for efficient cleanup.

When obj goes out of scope and is not referenced anymore, **GC collects it automatically**.

# **Pro**

## Q1. What are marker interfaces? Provide examples.

**Marker interfaces** are interfaces **with no methods or fields**. They are used to **mark or tag** a class with metadata so that the **JVM or framework** can apply special behavior.

**🔹 Purpose:**

* They provide **runtime information** to the JVM or libraries.
* Help in applying special processing to the marked classes.

**🔹 Common Examples:**

| **Marker Interface** | **Purpose** |
| --- | --- |
| Serializable | Marks a class whose objects can be serialized. |
| Cloneable | Marks a class that allows cloning via .clone(). |
| Remote | Marks a class for remote method invocation (RMI). |

import java.io.Serializable;  
  
public class Person implements Serializable {  
 private String name;  
 private int age;  
}

Here, Person is marked as Serializable, so its objects can be converted into a byte stream.

## Q2. Why Cloneable is needed even though every class extends Object and Object has .clone()?

Yes, **every class in Java implicitly extends Object**, and Object has a clone() method.  
But here's the key point: If you call clone() on an object that doesn't implement Cloneable, the JVM throws **CloneNotSupportedException**

 The Object.clone() method checks **at runtime** whether the object implements the **Cloneable marker interface**.

 If it **doesn't**, it considers the class **not eligible for cloning**, and throws an exception.

class Person {  
 int age = 25;  
}  
  
public class Test {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 Person p1 = new Person();  
 Person p2 = (Person) p1.clone(); // ❌ Compile-time OK, runtime error  
 }  
}

Output: java.lang.CloneNotSupportedException

class Person implements Cloneable{  
 int age = 25;  
 public Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
}  
  
public class Test {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 Person p1 = new Person();  
 Person p2 = (Person) p1.clone(); // ❌ Compile-time OK, runtime error  
 System.*out*.println(p2.age);  
 }  
}

Even though clone() exists in Object, only classes that **implement Cloneable** are allowed to be cloned.  
The Cloneable interface acts as a **marker to signal cloning is safe** for that class.

## Q3. What is Serialization in Java?

**Serialization** is the process of **converting an object into a byte stream** so it can be:

* Saved to a file
* Transferred over a network
* Stored in a database

It allows the object's **state** to be persisted or shared.

**Deserialization**

**Deserialization** is the reverse process — **converting the byte stream back into a copy of the original object**.

**Why Use Serialization?**

* To **save objects** (e.g., to disk)
* To **send objects over a network** (e.g., in distributed systems, RMI)
* To **cache data**

To make class serializable: just implement the marker interface

class MyClass implements Serializable {  
 // Now this class can be serialized  
}

import java.io.\*;  
  
// Marker interface  
interface Serializable {}  
  
// Class implementing marker interface  
class Person implements Serializable {  
 private String name;  
 private int age;  
  
 // Constructor  
 public Person(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 // To display object data  
 public String toString() {  
 return name + " (" + age + ")";  
 }  
}  
  
public class SerializationDemo {  
 public static void main(String[] args) {  
 Person person = new Person("Alice", 30);  
  
 // Serialize  
 try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("person.ser"))) {  
 out.writeObject(person);  
 System.*out*.println("Object serialized: " + person);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
  
 // Deserialize  
 try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("person.ser"))) {  
 Person deserialized = (Person) in.readObject();  
 System.*out*.println("Object deserialized: " + deserialized);  
 } catch (IOException | ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
 }  
}

OUTPUT:

Object serialized: Alice (30)

Object deserialized: Alice (30)

## Q4. What is a Method Reference in Java?

A **method reference** is a **shorthand syntax** for calling a method via a reference to it, without executing it.  
It is a **concise way** to use lambda expressions when you're **only calling an existing method**.

ClassName::methodName

**Types:**

| **Type** | **Example** | **Equivalent Lambda** |
| --- | --- | --- |
| Static method | ClassName::staticMethod | x -> ClassName.staticMethod(x) |
| Instance method (on object) | obj::instanceMethod | x -> obj.instanceMethod(x) |
| Instance method (on type) | ClassName::instanceMethod | (obj, arg) -> obj.instanceMethod(arg) |
| Constructor reference | ClassName::new | () -> new ClassName() or args -> new |

List<String> names = Arrays.asList("Zara", "Bob", "Alice");  
  
// Using lambda  
names.forEach(name -> System.out.println(name));  
  
// Using method reference  
names.forEach(System.*out*::println); // ✅ Shorter, cleaner

Method references are a **cleaner alternative to lambdas** when you're just **calling an existing method**, and improve readability.  
They were introduced in **Java 8** as part of functional programming features.

## Q5. How method reference really works, how to call person.getName for single person in above case?

Method references and lambda expressions are **shortcuts for implementing a functional interface**.  
They are **not standalone code** — Java needs a **target type** to bind them to.

() -> "Hello"  
//This lambda doesn't do anything by itself. Java needs to know where to plug it in— i.e., what method it is replacing. That’s where a functional interface comes in.

Supplier<String> s = () -> "Hello";  
  
//This means: "I'm supplying an object with a get() method that returns Hello."

Because person::getName is not a method call — it’s a **reference to a method**.  
It needs a **functional interface** to **invoke it**.

* **person.getName() →** ✅ method call → returns the name immediately
* **person::getName** → ❌ not a call → it's a **pointer to the method**, like a function handle

person::getName; // ❌ Invalid by itself — no context, Java doesn't know what to do  
  
// Create a reference to getName, assigned to a functional interface  
Supplier<String> s = person::getName;  
System.out.println(s.get()); // Now it gets called

## Q6. Use method references to sort a list of Person objects by name.

import java.util.\*;  
  
class Person {  
 private String name;  
 private int age;  
  
 // Constructor  
 public Person(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 // Getter  
 public String getName() {  
 return name;  
 }  
  
 // ToString for easy output  
 public String toString() {  
 return name + " (" + age + ")";  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 List<Person> people = new ArrayList<>();  
 people.add(new Person("Zara", 30));  
 people.add(new Person("Alice", 25));  
 people.add(new Person("Bob", 28));  
  
 // ✅ Sort using method reference  
 people.sort(Comparator.comparing(Person::getName));  
  
 // Print sorted list  
 people.forEach(System.*out*::println);  
 }  
}

* Person::getName is a **method reference**.
* Comparator.comparing() uses it to sort the list by name.
* Clean, readable, and uses modern Java (Java 8+).

## Q7. What are the best practices for designing extensible and maintainable OOP systems?

* Follow **SOLID** principles for clean and scalable code.
* Prefer **composition over inheritance** for flexibility.
* **Program to interfaces**, not implementations.
* **Encapsulate what varies** to isolate change.
* Use **design patterns** like Strategy, Factory, and Observer wisely.
* Keep classes **small, focused, and cohesive**.
* Favour **immutability** where possible.
* Avoid **tight coupling**; use **dependency injection**.
* Use **clear, meaningful names** and keep documentation updated.
* Write **unit tests** to ensure reliability and maintainability.

## Q8. What are the best practices for designing extensible and maintainable OOP systems?

**Coupling** is the degree of **interdependence** between software modules or classes.  
It shows **how much one class knows about another**.

 **Tight Coupling** – Classes are highly dependent on each other.

 **Loose Coupling** – Classes are independent and interact through interfaces or abstraction.

//Tight coupling  
class Engine {  
 public void start() {  
 System.*out*.println("Engine started");  
 }  
}  
  
class Car {  
 Engine engine = new Engine(); // Tight coupling  
  
 public void startCar() {  
 engine.start();  
 }  
}

 Car **directly depends** on Engine.

 If Engine changes, Car must also be updated.

//Loose Coupling  
interface Engine {  
 void start();  
}  
  
class PetrolEngine implements Engine {  
 public void start() {  
 System.*out*.println("Petrol engine started");  
 }  
}  
  
class Car {  
 private Engine engine;  
  
 public Car(Engine engine) { // Engine injected  
 this.engine = engine;  
 }  
  
 public void startCar() {  
 engine.start();  
 }  
}

* Car depends on **abstraction**, not concrete class.
* Can easily replace PetrolEngine with ElectricEngine later.

## Q9. What Are Annotations in Java?

**Annotations** are a form of metadata in Java that provide data about a program but are not part of the program itself. They have no direct effect on the operation of the code they annotate.

Annotations can be used for:

* Compiler instructions (e.g., @Override, @SuppressWarnings)
* Build-time processing (e.g., using tools like annotation processors)
* Runtime behavior changes (e.g., frameworks like Spring or JPA)

**Common Built-in Annotations**

| **Annotation** | **Purpose** |
| --- | --- |
| @Override | Ensures a method is overridden correctly |
| @Deprecated | Marks a method/class as deprecated |
| @SuppressWarnings | Suppresses compiler warnings |
| @FunctionalInterface | Declares a functional interface |

## Q10. What is Reflection in Java?

**Reflection** is a feature in Java that allows your code to **inspect and manipulate classes, methods, fields, and constructors** at runtime — even if you don’t know them at compile time.

It’s part of the java.lang.reflect package.

import java.lang.reflect.\*;  
  
class Person {  
 private String name = "Alice";  
 public void sayHello() {  
 System.*out*.println("Hello, I am " + name);  
 }  
}  
  
public class ReflectionExample {  
 public static void main(String[] args) throws Exception {  
 Class<?> clazz = Class.*forName*("Person");  
 Object obj = clazz.getDeclaredConstructor().newInstance();  
  
 Field field = clazz.getDeclaredField("name");  
 field.setAccessible(true);  
 field.set(obj, "Bob"); // Modify private field  
  
 Method method = clazz.getMethod("sayHello");  
 method.invoke(obj); // Output: Hello, I am Bob  
 }  
}

**🔹 Why is it useful?**

* Access private fields/methods
* Load classes dynamically
* Create objects at runtime
* Useful for frameworks (like Spring, Hibernate)

**🔹 When to Use Reflection:**

* In frameworks
* When working with annotations
* For debugging, testing, serialization, object inspection