**1. Can a Java class be both abstract and final? Why or why not?**

**Answer:**

❌ **No**, a Java class **cannot** be both abstract and final.

* abstract means the class **must be extended** to provide implementations for abstract methods.
* final means the class **cannot be extended**.

🧠 So putting both together is a **logical contradiction** — a class cannot **require** subclassing (abstract) and **forbid** it at the same time (final).

**3. What happens if you override a static method in Java?**

**Answer:**

In Java, **static methods are not actually overridden**, they are **hidden**.

* Static methods belong to the **class**, not the instance.
* If a subclass declares a static method with the **same signature**, it hides the superclass method — this is called **method hiding**, not overriding.

class Parent {

static void greet() { System.out.println("Parent"); }

}

class Child extends Parent {

static void greet() { System.out.println("Child"); }

}

public class Test {

public static void main(String[] args) {

Parent p = new Child();

p.greet(); // Output: Parent (because method is static)

}

What is the difference between method overloading and method overriding in Java? Can both be applied to the same method?

|  |  |  |
| --- | --- | --- |
| **Feature** | **Overloading** | **Overriding** |
| Definition | Same method name, **different parameters** | Same method name, **same parameters** |
| Binding | **Compile-time** (static) | **Runtime** (dynamic) |
| Class Level | Same class (or subclass) | Must involve a **parent–child** class relationship |
| Return Type | Can be different (with type-safe signature) | Must be the same or covariant |
| Access Modifier | No restriction | Cannot reduce visibility |
| Exception Rules | Independent | Must follow base class exception rules |

**Yes**, you can have both **overloaded and overridden** versions of the same method across class hierarchies.

class Parent {

void greet() { System.out.println("Parent"); }

}

class Child extends Parent {

@Override

void greet() { System.out.println("Child"); } // Overriding

void greet(String name) { System.out.println("Hi " + name); } // Overloading

}

**Rule 1: Overriding method cannot reduce visibility**

You can increase visibility (e.g., from protected to public), but **not reduce** it (e.g., from public to private).

class Parent {

public void show() {

System.out.println("Parent");

}

}

class Child extends Parent {

// ❌ Compilation error: can't reduce visibility from public to private

private void show() {

System.out.println("Child");

}

}

Valid Override — Same or Broader Visibility

class Parent {

protected void display() {

System.out.println("Parent");

}

}

class Child extends Parent {

// ✅ OK: public is broader than protected

public void display() {

System.out.println("Child");

}

**Does Java 8's default method in interfaces violate the “diamond problem”? Why or why not?**

**Answer:**

No, Java 8's default methods in interfaces do **not violate the diamond problem** because **Java requires explicit conflict resolution** when multiple default methods are inherited.

🧠 The compiler forces the class to **override the conflicting method**, eliminating ambiguity.

interface A {

default void greet() { System.out.println("Hello from A"); }

}

interface B {

default void greet() { System.out.println("Hello from B"); }

}

class C implements A, B {

@Override

public void greet() {

A.super.greet(); // or B.super.greet()

}

}

What is the difference between Optional.of(), Optional.ofNullable(), and Optional.empty()?

|  |  |
| --- | --- |
| **Method** | **Behavior** |
| Optional.of(value) | ❌ Throws NullPointerException if value is null |
| Optional.ofNullable(value) | ✅ Returns Optional.empty() if value is null; otherwise wraps value |
| Optional.empty() | ✅ Creates an explicitly empty Optional |
|  |  |

String str = null;

Optional<String> o1 = Optional.of(str); // ❌ Throws NullPointerException

Optional<String> o2 = Optional.ofNullable(str); // ✅ Returns Optional.empty

Optional<String> o3 = Optional.empty(); // ✅ Safe, empty Optional

**. Can a functional interface have multiple default methods?**

**Answer:**

✅ **Yes**, a functional interface in Java 8 **can have multiple default and static methods** — as long as it has **exactly one abstract method**.

@FunctionalInterface

interface MyFunc {

void doWork(); // single abstract method

default void log() {

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

}

default void audit() {

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

}

static void help() {

System.out.println("Help method");

}

}

Sprin boot\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**2. What is the difference between @ComponentScan and @SpringBootApplication? Don’t they both scan components?**

**Answer:**

✅ Great question — yes, @SpringBootApplication **includes** @ComponentScan by default, but there's a subtle distinction:

* @ComponentScan tells Spring **where to look for components**, controllers, services, etc.
* @SpringBootApplication is a **meta-annotation** that combines:
  + @Configuration
  + @EnableAutoConfiguration
  + @ComponentScan

**2. What is the difference between @ComponentScan and @SpringBootApplication? Don’t they both scan components?**

**Answer:**

✅ Great question — yes, @SpringBootApplication **includes** @ComponentScan by default, but there's a subtle distinction:

* @ComponentScan tells Spring **where to look for components**, controllers, services, etc.
* @SpringBootApplication is a **meta-annotation** that combines:
  + @Configuration
  + @EnableAutoConfiguration
  + @ComponentScan

@SpringBootApplication

public class MyApp {

public static void main(String[] args) {

SpringApplication app = new SpringApplication(MyApp.class);

app.setWebApplicationType(WebApplicationType.NONE);

app.run(args);

}

}