

Java Interview - ChatGPT

Java Core Interview Q&A

1 JVM, JDK, JRE

Q: 🤖 What is the difference between JVM, JDK, and JRE?

A:

- **JVM (Java Virtual Machine)** 🖥️ → Runs Java bytecode. Platform-dependent implementation, responsible for memory management, GC, etc.
- **JRE (Java Runtime Environment)** ⚙️ → Provides JVM + core libraries (but no compiler). Used to run applications.
- **JDK (Java Development Kit)** 📦 → JRE + development tools (compiler `javac`, debugger, etc.). Used for development.

💡 Example (Industry):

- A **Java developer** installs **JDK** to build software.
 - A **client machine** with only **JRE** can run the application.
 - The application runs on **JVM**, which is the execution engine.
-

2 Default Values in Class Fields

Q: 🤖 What is the default value of uninitialized properties (class fields) like `boolean b` and `Integer i`?

A:

- `boolean b` → `false`
- `Integer i` → `null`

💡 Example:

```
class Test {
```

```
boolean b;        // false
Integer i;        // null
}
```

In a **Spring Boot app**, if you forget to initialize a Boolean field in a config class, it defaults to `false`.

3 Default Values in Local Variables

Q: 😊 What is the default value of uninitialized variables inside a local method (`boolean b`, `Integer i`)?

A: ❌ They have **no default value**. You must **initialize them before use** → otherwise compiler error.

💡 Example:

```
void test() {
    boolean b; // ❌ error if accessed before assignment
}
```

4 Final Variables Initialization

Q: 🗝️ How to initialize `final` variables?

A:

- At declaration
- Inside constructor
- Inside instance initializer block

💡 Example:

```
final int x;
Test() {
    x = 10; // ✅ constructor initialization
}
```

Industry case: Used in **thread-safe immutable objects** like `String`.

5 Passing Strings vs Primitives

Q: 📦 How are Strings passed to a method – by reference or value? What about primitive booleans?

A:

- Java is always pass-by-value.
- For **primitives** → actual value is copied.
- For **objects (like String)** → reference is copied (so both point to same object, but cannot reassign original reference).

💡 Example:

```
void modify(String s) { s = "changed"; } void modifyBool(boolean b) { b = false; }
```

Neither changes the original.

Industry case: Avoids accidental modifications in **microservices DTOs**.

6 Where Stored in JVM?

Q: 🧠 Where are booleans and Strings stored in JVM?

A:

- **Primitives (boolean, int, etc.)** → stored in stack or heap depending on scope.
- **Strings** → stored in **heap** (with special String Pool for literals).

💡 Example: String pooling improves performance in **high-traffic web apps** by reusing `"OK"` strings for HTTP status.

7 Overloading vs Overriding

Q: ✂ What is the difference between overriding and overloading?

A:

- **Overriding:** Child class changes method behavior of parent (runtime polymorphism).
- **Overloading:** Same method name with different parameter list in the same class (compile-time polymorphism).

💡 Example:

- **Overloading:** `log(String msg)` & `log(String msg, int level)`
 - **Overriding:** `List.add()` overridden in `ArrayList`.
-

8 Can I Override Overloaded Method?

Q: ? Can I override the overloaded method?

A: ✅ Yes.

- Overloading happens in the **same class**.
- Those methods can still be **overridden** in subclass.

💡 **Example:**

```
class A {  
    void print(int x) {}  
    void print(String y) {}  
}  
class B extends A {  
    @Override void print(int x) {} // ✅ overriding overloaded method  
}
```

9 Can I Override Private Method?

Q: ❌ *Can I override a private method?*

A: No.

- Private methods are **not visible to child classes**, so overriding is impossible.
- If you define a method with the same signature in child, it's a **new method (not override)**.

💡 **Example:**

Used in security → **private credentials check** cannot be overridden.

10 Can I Override Static Method?

Q: ⚡ *Can I override a static method?*

A: No.

- Static methods belong to **class**, not object.
- If same signature exists, it's **method hiding** not overriding.

11 Can I Overload Overridden Method?

Q: 🔄 *Can I overload an overridden method?*

A: ✅ Yes.

- Overriding is about **same signature**.
- You can **add new overloaded variants**.

💡 **Example:**

```
class A { void show(int x) {} }  
class B extends A {  
    @Override void show(int x) {} // override  
    void show(String y) {} // overload  
}
```

```
}
```

1² Can I Overload Final Method?

Q: 🛑 Can I overload a final method?

A: ✅ Yes.

- `final` only prevents **overriding**, not overloading.

1³ Can I Declare Static Method in Interface?

Q: 📖 Can I declare a static method in interface?

A: ✅ Yes (since Java 8).

- Example: `Comparator.naturalOrder()`

1⁴ Can I Change Return Type in Overridden Method?

Q: 🔄 Can I change return type in overridden method?

A:

- Only **covariant return types** allowed (child class return type).

💡 Example:

```
class Parent { Object get() { return null; } }
class Child extends Parent {
    @Override String get() { return "hi"; } // ✅ allowed
}
```

1⁵ Calling Non-Static from Static

Q: ⚡ Can I call a non-static method from a static method?

A: ✅ Yes, but you must create an object.

💡 Example:

```
class Test {
    void hello() {}
    static void call() {
```

```
new Test().hello(); // ✔  
}  
}
```

⚡ Java Core (Part 2)

16 Composition vs Aggregation

Q: 🧩 What is the difference between composition and aggregation?

A:

- **Composition (strong ownership)** 🏠 → Child object's lifecycle depends on parent. If parent is destroyed, child also destroyed.
- **Aggregation (weak ownership)** 🤝 → Child object can exist independently of parent.

💡 Example (Industry):

- **Composition:** A **Car** 🚗 has an **Engine**. Without engine, car cannot exist.
 - **Aggregation:** A **Team** 🏆 has **Players**. Players can exist without team.
-

17 Abstraction vs Encapsulation

Q: 🤖 What is the difference between abstraction and encapsulation?

A:

- **Abstraction** → Hides *implementation details* and shows only functionality. (e.g., **List** interface)
- **Encapsulation** → Hides *data* by using private fields and getters/setters.

💡 Example:

- **Abstraction:** JDBC API (you call **executeQuery()**, don't know DB internals).
 - **Encapsulation:** In **BankAccount**, balance is private; accessed via methods.
-

18 Polymorphism

Q: 🔄 What is polymorphism?

A:

Ability of one object to take many forms.

- Compile-time polymorphism → Overloading.
- Runtime polymorphism → Overriding.

💡 Example:

Spring framework uses polymorphism in `ApplicationContext.getBean()` → same method returns different beans.

19 String, StringBuilder, StringBuffer

Q: 📖 Why do we need 3 classes for strings: *String*, *StringBuilder*, *StringBuffer*?

A:

- **String** → Immutable. Good for constants, keys in HashMap.
- **StringBuilder** → Mutable, not thread-safe, fast. Used in single-threaded scenarios.
- **StringBuffer** → Mutable, thread-safe, slower. Used in multi-threaded environments.

💡 Example:

- **String** for config keys (`"DB_URL"`).
 - **StringBuilder** for JSON/XML building.
 - **StringBuffer** in legacy multi-threaded code.
-

20 Immutability & String

Q: 📖 What is immutability? Is *String* immutable?

A:

- Immutability = Once created, object cannot be changed.
- ✅ Yes, `String` is immutable in Java.

💡 Why?

- Security (passwords, URLs).
 - Caching (String pool).
 - Thread-safety.
-

21 Implement Immutable Object

Q: 📖 How to implement an immutable object?

A:

1. Make class `final`.
2. Fields `private final`.
3. No setters.
4. Return copies of mutable objects.


💡 Example:


```
final class User {
    private final String name;
    private final Date dob;
    public User(String name, Date dob) {
        this.name = name;
        this.dob = new Date(dob.getTime()); // copy
    }
    public Date getDob() { return new Date(dob.getTime()); }
}
```

Used in Java 8 `LocalDate`.

2.2 Rethrow Exception Multiple Times

Q:  Can we rethrow the same exception multiple times?

A:  Yes, but only if caught again.

 Example:

```
try {
    throw new IOException("fail");
} catch (IOException e) {
    throw e; // rethrow
}
```

2.3 Exception Hierarchy

Q:  Describe top three levels of exception hierarchy.

A:

- `Throwable` (root)
 - `Error` (serious, unrecoverable)
 - `Exception` (recoverable)

 Example:

- `OutOfMemoryError` (Error)
 - `IOException` (checked)
 - `NullPointerException` (unchecked)
-

2⁴ Checked vs Unchecked Exceptions

Q: ⚠️ What is the difference?

A:

- **Checked** → Must be handled (compile-time). E.g., `SQLException`.
- **Unchecked** → Runtime exceptions. E.g., `NullPointerException`.

Industry case: JDBC operations throw **checked exceptions**, forcing devs to handle.

2⁵ Throwable Class or Interface?

Q: 🤔 Is Throwable an interface or a class?

A: It's a **class** (parent of Exception & Error).

2⁶ try-finally Without Catch

Q: 🗝️ Can we use try-finally without catch?

A: ✅ Yes.

- Used when cleanup is needed.

💡 Example:

```
try {
    FileInputStream fis = new FileInputStream("a.txt");
} finally {
    // close stream
}
```

2⁷ return in try-finally

Q: 🔄 In `try {return 1;} finally {return 2;}` what will be returned?

A: → `2`.

- Finally block overrides return.
-

2⁸ Exception in try-catch-finally

Q: 🔥 Will finally be executed if exception is thrown & caught?

A: ✅ Yes, `finally` always runs (except `System.exit(0)`).

29 HashMap Internal

Q: 🇮🇹 Why does HashMap use LinkedList inside it instead of ArrayList?

A:

- For collisions handling → LinkedList (or Tree in Java 8).
 - ArrayList unsuitable because insert/remove $O(n)$.
-

30 Calling Private Method Outside Class

Q: 🚫 Can I call a private method outside class?

A: Normally ❌ No.

But ✅ Yes via **reflection**.

💡 Example:

Used in JUnit tests.

31 Transient Variable

Q: 🔑 What is a transient variable?

A:

- A variable not serialized.

Industry case: Password fields in objects marked `transient`.

32 static Keyword

Q: ⚡ What is "static" keyword?

A:

- Belongs to class, not object.
- Used for variables, methods, blocks, nested classes.

💡 Example: `Math.max()`

33 Multiple Inheritance

Q: 🧩 Is multiple inheritance supported in Java?

A: ❌ Not with classes.

✅ Yes with interfaces (Java 8 default methods).

34 Multiple Constructors

Q: 🏗️ Can I call more than one constructor from another constructor?

A: ✅ Yes, using `this()`, but only first line.

3⁵ Can I Override Constructor?

Q: ❌ Can I override constructor?

A: No, constructors are not inherited.

3⁶ Negative hashCode()

Q: ¹²₃₄ Can hashCode() return negative value?

A: ✅ Yes. Still valid.

3⁷ Garbage Collection & Memory

Q: 🗑️ GC algorithms & types of memory?

A:

- **Heap Memory:** Young, Old, PermGen/Metaspace.
 - **GC Algorithms:** Mark & Sweep, Copying, CMS, G1.
-

3⁸ Exception in Static Block

Q: 💣 What if runtime exception is thrown in static block?

A: → Class initialization fails → `ExceptionInInitializerError`.

3⁹ HashMap Complexity Worst Case

Q: ⌚ Complexity of get() in worst case when all hashCodes are same?

A:

- $O(n)$ in worst case (LinkedList traversal).
 - $O(\log n)$ in Java 8+ (Tree).
-

4⁰ HashSet vs TreeSet

Q: 🌳 HashSet vs TreeSet?

A:

- **HashSet:** Unordered, backed by HashMap, $O(1)$ average.
 - **TreeSet:** Sorted, backed by TreeMap, $O(\log n)$.
-

4¹ Optional Class

Q: 📦 Why would we use Optional class?

A:

- To avoid `NullPointerException`.
- Explicitly represent presence/absence of value.

Industry case: Used in **Spring Data JPA** `findById()` → returns `Optional<User>`.

🌟 Java 8 Interview Q&A

1 Default Methods in Interface

Q: 🤔 *Why were default methods added to interfaces in Java 8?*

A:

- To provide **backward compatibility** → old interfaces can evolve without breaking existing implementations.
- Allow **multiple inheritance of behavior** (not state).

💡 **Example (Industry):**

```
interface Vehicle {  
    default void start() { System.out.println("Starting..."); }  
}  
class Car implements Vehicle {}  
new Car().start(); // ✅ works without redefining
```

👉 Java Collections API introduced default methods like `forEach()` in `Iterable`.

2 Stream vs List

Q: 🔄 *What is the difference between Stream and List?*

A:

- **List**: Data structure holding elements (stored in memory).
- **Stream**: Sequence of elements processed in functional style (not stored, lazy evaluated).

💡 **Example:**

```
List<String> names = Arrays.asList("A", "B", "C");  
Stream<String> s = names.stream().filter(n -> n.startsWith("A"));
```

Industry case: Streams allow **parallel processing** in **big data pipelines**.

3 Intermediate Methods in Streams

Q:  What are intermediate methods in streams?

A:

Methods that return a new Stream → **lazy** evaluation.

- Examples: `filter()`, `map()`, `sorted()`, `distinct()`.

 Example:

```
Stream.of(1,2,3,4)
    .filter(x -> x % 2 == 0) // intermediate
    .map(x -> x * 2);        // intermediate
```

4 What Can't Be Done with Parallel Streams

Q:  What can't be done with parallel streams?

A:

- Operations that depend on **order** (like writing to file sequentially).
- Non-associative reductions (e.g., floating-point sum precision).
- Shared mutable state → race conditions.

Industry case:

Avoid parallel streams when updating **shared collections** in multi-threaded apps.

5 Lambda Expressions

Q:  Describe the three main parts of lambda expressions.

A:

1. Parameters → `(a, b)`
2. Arrow token → `->`
3. Body → `{ return a+b; }`

 Example:

```
(a, b) -> a + b
```

Industry case: Widely used in **Java Streams API** for concise operations.

6 New Date/Time API

Q: 🕒 *Main features of new DateTime API (Java 8)?*

A:

- Immutable and thread-safe.
- Clear API (`LocalDate`, `LocalTime`, `LocalDateTime`).
- Better formatting & parsing (`DateTimeFormatter`).
- Time zones (`ZonedDateTime`).

💡 **Example:**

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plusDays(1);
```

Industry case: Used in **financial apps** to avoid timezone bugs that `java.util.Date` had.

⚡ Multithreading Interview Q&A

1 Thread vs Process

Q: 🧵 *What is the difference between Thread and Process?*

A:

- **Process** 🏢 → Independent program in execution with its own memory space.
- **Thread** 🧵 → Lightweight sub-task of a process. Shares memory with other threads in the same process.

💡 **Industry Example:**

- Chrome browser → each **tab** is a process, but within a tab, multiple **threads** handle UI, rendering, and networking.
-

2 Thread.start() vs Thread.run()

Q: 🚦 What is the difference between `start()` and `run()`?

A:

- `start()` → Creates a **new thread** and calls `run()`.
- `run()` → Just executes code in the **same thread** (like a normal method).

💡 Example:

```
Thread t = new Thread() -> System.out.println(Thread.currentThread().getName());
t.start(); // ✅ runs in new thread
t.run();   // ❌ runs in main thread
```

Industry case: Calling `run()` directly in web server thread would block requests.

3 volatile Keyword

Q: ⚡ What is the purpose of volatile keyword?

A:

- Ensures **visibility** of changes across threads.
- Prevents caching of variable in thread's local memory.

💡 Example:

```
volatile boolean running = true;
```

Used in **producer-consumer systems** to stop threads gracefully.

4 Semaphore

Q: 🚧 What is a semaphore?

A:

- A concurrency utility to control access to a resource using permits.
- Can be **binary (mutex)** or **counting**.

💡 Example:

```
Semaphore s = new Semaphore(2); // 2 permits
```

```
s.acquire(); // thread acquires
s.release(); // releases
```

Industry case: Limit database connection pool size.

5 sleep() vs wait()

Q: 🛏 What is the difference between `sleep()` and `wait()`?

A:

- `sleep(ms)` → Thread pauses for time, does not release lock.
- `wait()` → Thread releases lock and waits until `notify()/notifyAll()`.

💡 Example:

- `sleep()` → Retry logic with delays.
 - `wait()` → Producer-Consumer coordination.
-

6 Exception from Another Thread

Q: 🚨 How can I get an exception from another thread?

A:

- Use **Future + Callable** → exception is captured and rethrown when calling `future.get()`.

💡 Example:

```
ExecutorService es = Executors.newSingleThreadExecutor();
Future<Integer> f = es.submit(() -> { throw new RuntimeException("boom"); });
f.get(); // throws ExecutionException
```

7 Exception Without Handler

Q: ⚠ What happens if an exception is thrown from a thread without an exception handler?


A:

- JVM terminates the thread.
- Can set **UncaughtExceptionHandler** to handle.

Industry case: In microservices, thread pool tasks log errors using

`Thread.setDefaultUncaughtExceptionHandler()`.

8 Thread States


Q:  What are the thread states?

A:

- NEW
- RUNNABLE
- BLOCKED
- WAITING
- TIMED_WAITING
- TERMINATED

💡 **Example:** Used in **VisualVM** to monitor JVM threads.

9 Restarting a Thread

Q:  A thread has finished work. Can I start it again?

A: ❌ No. Once terminated, thread cannot be restarted → `IllegalThreadStateException`.

👉 Need to create a new `Thread` object.

10 Thread-Safe Singleton

Q:  How to implement thread-safe singleton?

A:

- Double-checked locking with `volatile`.

```
class Singleton {
    private static volatile Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) instance = new Singleton();
            }
        }
        return instance;
    }
}
```

Industry case: Used in logging frameworks.

1 1 JVM Stacks

Q: 🧠 How many stacks are there in JVM?

A:

- Each thread has its **own stack**.
 - So number of stacks = number of threads.
-

1 2 Future vs Callable

Q: 📦 Future vs Callable?

A:

- **Callable** → task that returns value (like Runnable + return).
- **Future** → placeholder for result of Callable.

💡 Example:

```
Future<Integer> f = executor.submit(() -> 42);  
System.out.println(f.get()); // 42
```

🧵 Multithreading Interview Q&A

1 Thread Class vs Runnable Interface

Q: 🤔 Difference between extending `Thread` and implementing `Runnable`?

A:

- **Extending Thread** → Not recommended if you need multiple inheritance.
- **Implementing Runnable** → Preferred; separates task from execution.

💡 Example:

```
class MyTask implements Runnable {  
    public void run() { System.out.println("Task running"); }  
}new Thread(new MyTask()).start();
```

Industry case: Runnable is widely used in **ThreadPools (ExecutorService)**.

2 start() vs run()

Q: 🔄 Difference between `start()` and `run()` methods?

A:

- `start()` → Creates a new thread, calls `run()` asynchronously.
- `run()` → Normal method call, no new thread created.

💡 Example:

```
Thread t = new Thread(() -> System.out.println("Hello"));
t.run();    // Runs in main thread ❌
t.start();  // Runs in new thread ✅
```

3 Thread States

Q: 🇮🇹 What are the different thread states in Java?

A:

- NEW 🆕 → created but not started.
- RUNNABLE 🏃 → ready to run.
- RUNNING ⚡ → executing on CPU.
- BLOCKED 🛑 → waiting for monitor lock.
- WAITING ⌚ → indefinitely waiting.
- TIMED_WAITING ⌚ → waiting with timeout.
- TERMINATED 🛑 → finished.

Industry case: Debugging **deadlocks** requires analyzing thread states in tools like JVisualVM.

4 Volatile vs Atomic

Q: ⚡ Difference between `volatile` and `Atomic` variables?

A:

- `volatile` → Guarantees **visibility** of changes across threads. No atomicity.
- `AtomicXXX` (e.g., `AtomicInteger`) → Provides **atomic operations** (CAS-based).

💡 Example:

```
volatile boolean flag = true;AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet(); // atomic
```

5 Semaphore

Q: 🚦 What is a Semaphore in Java?

A:

- A synchronization aid controlling **number of permits** to access a resource.
- Useful for **rate-limiting** or **connection pooling**.

💡 Example:

```
Semaphore sem = new Semaphore(3); // max 3 permits
sem.acquire();try {
    // critical section
} finally {
    sem.release();
}
```

Industry case: Used in **database connection pools** to allow only N connections.

6 Deadlock

Q: ⚠️ What is a deadlock? How to avoid it?

A:

- When two/more threads wait for each other's locks → infinite wait.

Avoid by:

- Lock ordering ✅
 - Using `tryLock()` with timeout ✅
 - Minimizing synchronized blocks ✅
-

7 Executor Framework

Q:  Why use `ExecutorService` instead of creating threads manually?

A:

- Manages a pool of threads.
- Reuses threads → efficient.
- Provides task submission with `Future`.

💡 Example:

```
ExecutorService service = Executors.newFixedThreadPool(5);
Future<Integer> result = service.submit(() -> 42);
System.out.println(result.get());
service.shutdown();
```

Industry case: Used in **web servers** to handle multiple requests efficiently.

Spring Core Interview Q&A

1. Bean Scopes

Q: What are the different bean scopes in Spring?

A:

- **singleton** → One shared instance per Spring container (default).
- **prototype** → New instance every time bean is requested.
- **request** → One bean per HTTP request (web apps).
- **session** → One bean per HTTP session.
- **application** → One bean per ServletContext.
- **websocket** → One bean per WebSocket lifecycle.

Cross Q: If a prototype bean is injected into a singleton bean, will it behave as prototype?

👉 Ans: No. By default, Spring injects the same prototype instance at injection time. To truly get new instances, we need `@Lookup` method or `ObjectFactory/Provider`.

2. Singleton vs Prototype

Q: Difference between Singleton and Prototype beans?

A:

- **Singleton**: Single instance, cached and reused by container. Efficient, but shared state may cause thread-safety issues.
- **Prototype**: New instance every time. Useful for stateful objects but may impact performance.

Cross Q: Which scope would you choose for a DAO bean?

👉 **Ans:** Singleton, because DAOs are stateless and safe to share.

3. Dependency Injection (DI)

Q: What is Dependency Injection?

A: A design pattern where Spring container provides dependencies instead of the class creating them. Promotes loose coupling, testability, and flexibility.

Types of DI:

- **Constructor Injection** (`@Autowired` constructor)
- **Setter Injection** (`@Autowired` setter method)
- **Field Injection** (`@Autowired` directly on fields – not recommended for testability).

Cross Q: Which injection type do you prefer?

👉 **Ans:** Constructor Injection → ensures immutability, better for mandatory dependencies, easier unit testing.

4. Bean Lifecycle

Q: Explain the lifecycle of a Spring bean.

A:

1. Bean instantiated.
2. Dependencies injected.
3. `BeanNameAware` / `BeanFactoryAware` callbacks (if implemented).
4. `@PostConstruct` or `InitializingBean.afterPropertiesSet()`.
5. Bean is ready for use.
6. On shutdown → `@PreDestroy` or `DisposableBean.destroy()`.

Cross Q: How to customize bean initialization?

👉 **Ans:** Use `init-method` in XML or `@PostConstruct` in annotation config.

5. @Autowired

Q: What is `@Autowired`?

A: Marks a constructor, field, or setter to be automatically injected with a matching bean from the Spring container.

Cross Q: What happens if multiple beans of the same type exist?

👉 **Ans:** Use `@Qualifier("beanName")` or `@Primary` to resolve conflict.

6. Circular Dependency

Q: What is a circular dependency in Spring?

A: When Bean A depends on Bean B and Bean B depends on Bean A.

Spring resolves it using **setter injection** or by breaking the cycle with `@Lazy`.

Cross Q: Will constructor injection cause circular dependency failure?

👉 **Ans:** Yes, constructor injection fails in circular cases. Setter or field injection can resolve it.

🌟 Spring Boot Interview Q&A

1. IOC (Inversion of Control) in Spring Boot

Q: What is IOC in Spring Boot?

A: Inversion of Control (IoC) means the object creation and dependency management is handled by Spring's IoC Container instead of manually creating objects using `new`.

👉 Achieved via **ApplicationContext** and **@ComponentScan**.

Cross Q: How is IoC implemented in Spring Boot?

- Using **Dependency Injection (DI)** via annotations like `@Autowired`, `@Component`, `@Service`, `@Repository`, `@Configuration`.

Example:

```
@Service
public class OrderService {
    private final PaymentService paymentService;

    @Autowired
    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

2. AOP (Aspect-Oriented Programming)

Q: What is AOP in Spring Boot?

A: AOP allows separating cross-cutting concerns (like logging, transactions, security) from business logic.

Cross Q: What are the key concepts in AOP?

- **Aspect** → Module containing cross-cutting concern.
- **Advice** → Code to execute (Before, After, Around).
- **JoinPoint** → Point in execution (like method call).
- **Pointcut** → Expression that matches JoinPoints.

Example:

```
@Aspect
@Component
public class LoggingAspect {
    @Before("execution(* com.example.service.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("Before: " + joinPoint.getSignature());
    }
}
```

3. Transactions in Spring Boot

Q: How does Spring Boot handle transactions?

A: By using **@Transactional** annotation. It provides declarative transaction management.

Cross Q: What happens if a transaction fails?

- It **rolls back** automatically (for RuntimeExceptions by default).

Example:

```
@Service
public class BankService {
    @Transactional
    public void transferMoney(Account from, Account to, double amount) {
        from.debit(amount);
        to.credit(amount);
    }
}
```

4. Profiles in Spring Boot

Q: What are profiles in Spring Boot?

A: Profiles allow you to define different configurations for different environments (dev, test, prod).

Cross Q: How do you activate a profile?

- Via `application.properties`:

```
spring.profiles.active=dev
```


Example:

```
@Configuration
@Profile("dev")
public class DevConfig {
    @Bean
    public String devBean() {
        return "Dev Profile Bean";
    }
}
```

5. Spring Boot Actuator

Q: What is Spring Boot Actuator?

A: A module that provides production-ready features like health checks, metrics, info, and monitoring.

Cross Q: Common Actuator endpoints?

- `/actuator/health` → App health
- `/actuator/metrics` → Metrics
- `/actuator/info` → Info about app

Example:

```
management.endpoints.web.exposure.include=health,metrics,info
```

6. Spring Boot Security

Q: How is security handled in Spring Boot?

A: Using **Spring Security Starter**. By default, all endpoints are secured with basic auth.

Cross Q: How do you configure security in Spring Boot?

- Extend `WebSecurityConfigurerAdapter` (in Spring Security 5.7 and earlier).
- Or use **SecurityFilterChain Bean** in Spring Boot 3.

Example (Spring Boot 3.x):

```
@Configuration
public class SecurityConfig {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http
```

```
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
        )
        .httpBasic()
        .build();
    }
}
```

✅ That covers **Spring Boot Core Interview Q&A (IOC, AOP, Transactions, Profiles, Actuator, Security)** with examples and cross-questions.

✅ Hibernate Interview Q&A

Q1. Is Session object thread-safe?

A: No. Hibernate `Session` is **not thread-safe**. Each thread should use its own session.

👉 Reason: Session maintains persistence context (1st level cache), so concurrent modifications by multiple threads can cause inconsistent state.

Q2. It is necessary to run three queries in parallel with Hibernate. Will it require three session objects or one session object will be enough?

A: You will need **three separate sessions** (one per thread) since Session is not thread-safe.

Q3. How to map one entity to two tables?

A: Use `@SecondaryTable` annotation.

```
@Entity
@Table(name="user")
@SecondaryTable(name="user_details")
class User {
    @Id Long id;
    @Column(table="user_details") String address;
}
```

Q4. What is a first-level cache?

A: It's the **Session-level cache** in Hibernate. Every entity once loaded is stored in session.

- Avoids hitting the DB again within the same session.
 - Cleared when session is closed.
-

Q5. Explain all states of entity.

1. **Transient** → New object, not associated with Session.
 2. **Persistent** → Associated with Session, part of 1st-level cache.
 3. **Detached** → Was persistent, but Session is closed.
 4. **Removed** → Marked for deletion.
-

Q6. Explain N+1 problem.

A: When fetching parent entities, Hibernate issues **1 query for parents + N queries for each child collection**.

👉 Fix using `JOIN FETCH`, `EntityGraph`, or batch fetching.

Q7. How to implement distributed cache in Hibernate?

A: Use **2nd level cache providers** (EHCache, Infinispan, Hazelcast) with cluster support.

- Configure cache in `hibernate.cfg.xml`.
 - Mark entities/collections as `@Cacheable`.
-

Q8. Different types of fetching.

- **EAGER** → loads immediately with parent.
 - **LAZY** → loads only when accessed.
 - **JOIN FETCH** (query level).
 - **Batch Fetching**.
-

Q9. Difference between `get()` and `load()`?

- `get()` → Hits DB immediately, returns `null` if not found.
 - `load()` → Returns proxy, DB hit when accessed, throws `ObjectNotFoundException` if not found.
-

Q10. What is automatic dirty checking?

A: Hibernate automatically detects changes in persistent objects and updates DB on `flush()` / `commit()` without explicitly calling `update()`.

Q11. What is Query Cache in Hibernate?

A: A 2nd-level cache that stores **query results** (not entities). Requires enabling both query cache + 2nd level cache.

Q12. What are the inheritance mapping strategies?

- **SINGLE_TABLE** – All classes in one table (fast, less normalized).
 - **JOINED** – Separate table per subclass, joined with parent.
 - **TABLE_PER_CLASS** – Each class has its own table.
-

Q13. What will happen if a lazy loaded entity is accessed in a detached state?

A: It throws `LazyInitializationException`.

👉 Solution:

- Use `JOIN FETCH` before session close.
 - Or `Open Session In View` pattern.
 - Or DTO projection.
-

Q14. When is an id set for an entity?

A: Depending on strategy:

- `GenerationType.IDENTITY` → after insert.
 - `GenerationType.SEQUENCE` or `AUTO` → before insert (using sequence).
-

Q15. In what cases `equals()` and `hashCode()` should be overridden when using Hibernate?

A: When entity is used in **Set/Map** or compared logically.

- Use **business key** (not DB id, since id may be null before persistence).
-

Q16. Distributed transactions / XA transactions.

A: Use **JTA (Java Transaction API)** with XA-capable transaction managers (e.g., Atomikos, Bitronix, Narayana) to span multiple DBs/resources.

Q17. Lazy loading collection not loaded but session closed. How to load this collection?

A:

- Use `JOIN FETCH` before closing session.
- Use `Hibernate.initialize(entity.getCollection())` inside transaction.
- Or `@Transactional` method wrapping lazy access.

Database & SQL Interview Q&A

Q1. Explain all types of joins.

- **INNER JOIN** → common records.
 - **LEFT JOIN** → all left + matching right.
 - **RIGHT JOIN** → all right + matching left.
 - **FULL OUTER JOIN** → all from both sides.
 - **CROSS JOIN** → cartesian product.
 - **SELF JOIN** → join table with itself.
-

Q2. What is an index?

A: A data structure (usually B-Tree) that speeds up searches on columns.

Q3. Why not use index for every column?

- Slows down inserts/updates (index maintenance).
 - Takes extra memory.
 - Index is useful only for frequently searched columns.
-

Q4. Explain Isolation levels.

1. **READ UNCOMMITTED** → dirty reads allowed.
 2. **READ COMMITTED** → prevents dirty reads (default in most DBs).
 3. **REPEATABLE READ** → prevents non-repeatable reads.
 4. **SERIALIZABLE** → strictest, prevents phantom reads.
-

Q5. SQL vs NoSQL databases.

- **SQL** → structured, relational, ACID, schema-based.
 - **NoSQL** → flexible schema, horizontal scaling, BASE consistency, supports JSON docs, key-value, graph.
-

SQL Query Questions

- Find number of customers per location

```
SELECT location_id, COUNT(*) FROM customer GROUP BY location_id;
```

- **Many-to-many relationship** → create **junction table**.

```
CREATE TABLE student_course (  
  student_id INT,  
  course_id INT,  
  PRIMARY KEY(student_id, course_id)  
);
```

- **Select * From A, B** → Cartesian product ($A \times B$).
- **Delete duplicates**

```
DELETE FROM table t1 WHERE t1.id NOT IN (  
  SELECT MIN(id) FROM table GROUP BY column_with_duplicates  
);
```

- **Prepared Statement** → precompiled SQL with placeholders (`?`).
- **Benefits:** prevents SQL Injection, better performance via caching.
- **HAVING** → filter after aggregation.

```
SELECT dept, COUNT(*) FROM emp GROUP BY dept HAVING COUNT(*) > 5;
```

✓ REST Interview Q&A

Q1. What is JAX-RS?

Java API for RESTful Web Services (framework like Jersey, RESTEasy).

Q2. What is a Resource in REST?

An entity exposed via URI (e.g., `/users/1`).

Q3. Difference between GET and POST?

- GET → safe, idempotent, data in URL.
 - POST → not idempotent, data in body.
-

Q4. Parts of HTTP request?

- Request Line (method + URI).
 - Headers.
 - Body.
-

Q5. Ways to send arguments in REST?

- Query string (`/users?id=1`).
 - Path variable (`/users/1`).
 - Headers.
 - Body (POST/PUT).
-

Q6. How to force XML instead of JSON?

Set Accept header → `Accept: application/xml`.

Q7. How to make REST endpoints secure?

- HTTPS.
 - JWT/OAuth2 tokens.
 - Basic Auth / API Keys.
 - Role-based access (Spring Security).
-

Q8. What is content negotiation?

Deciding format (JSON, XML, etc.) based on `Accept` header.

Q9. What is idempotent operation?

Operation that gives same result no matter how many times called (e.g., GET, PUT, DELETE).

Q10. Stateful REST service?

By default REST is stateless, but state can be managed via **token**, **session store**, **Redis**.

✓ Design Patterns Q&A

Q1. SOLID with Java example.

- Single Responsibility → Service class only handles business logic.
 - Open/Closed → extend new behavior via subclassing, no modify.
 - Liskov Substitution → Subclass can replace parent without breaking.
 - Interface Segregation → multiple small interfaces instead of one big.
 - Dependency Inversion → Depend on abstraction, not concrete.
-

Q2. Singleton Pattern.

Ensure only one instance of a class.

```
public class Singleton {  
    private static final Singleton INSTANCE = new Singleton();  
    private Singleton(){}  
    public static Singleton getInstance(){ return INSTANCE; }  
}
```

Q3. Chain of Responsibility.

Pass request along chain until one handles it (e.g., servlet filters, logger chains).

Q4. Factory vs Builder.

- **Factory** → creates objects, hides creation logic.
 - **Builder** → constructs complex objects step by step.
-

Q5. Template Method Pattern.

Defines algorithm skeleton in abstract class, subclasses override steps.

Q6. Composition vs Inheritance.

👉 Prefer **composition** (flexible, avoids tight coupling).

Example: `Car has Engine` vs `Car extends Engine`.

Q7. Reuse methods of class A & B in new class.

Use **composition**: new class has objects of A and B.

Q8. Functional vs Imperative programming.

- Imperative → step by step instructions (`for` loops).
 - Functional → focus on **what**, not how (streams, lambdas).
-