

# **INTRODUCTION**

In the fast-paced world of technology, the right opportunity can be life-changing. For many aspiring software developers, cracking a Java interview is the pivotal moment that can set the trajectory of their career. "Cracking the JAVA INTERVIEWS WITH SUMIT" is designed to be your comprehensive guide through this challenging yet rewarding process. The subtitle, "A Good Interview can change your life!", encapsulates the transformative potential that lies within mastering the art of the technical interview.

This book is not just a collection of questions and answers. It is a meticulously crafted resource that aims to provide deep insights into the nature of Java interviews. Sumit, with years of experience both as a candidate and as an interviewer, brings a unique perspective that bridges the gap between theoretical knowledge and practical application. His approach is holistic, covering not only the technical aspects of Java but also the mindset required to excel in an interview setting.

Within these pages, you will find a blend of fundamental concepts, advanced topics, and real-world scenarios that are frequently encountered in interviews. Each chapter is structured to build your understanding progressively, ensuring that you are well-prepared for even the most challenging questions.

In addition to the technical content, this book offers some valuable real interview reports. By fostering a deeper comprehension of Java and its applications, this book aims to equip you with the confidence and competence needed to stand out in any interview.

Whether you are a fresh graduate aiming for your first job, a seasoned professional looking to switch roles, or someone re-entering the workforce, "Cracking the JAVA INTERVIEWS WITH SUMIT" is your essential companion. The practical advice, detailed explanations, and insider tips provided by Sumit will not only help you succeed in your interviews but also inspire you to approach them with a new level of preparedness and enthusiasm.

# Contents

Chapter 1: JAVA Interviews Evolution .....	5
Chapter 2: OOPS.....	9
Chapter 3: Core Java .....	30
Chapter 4: static keyword .....	62
Chapter 5: Java Collections .....	71
Chapter 6: Exception Handling .....	92
Chapter 7: Enums.....	108
Chapter 8: Serialization .....	114
Chapter 9: Generics.....	123
Chapter 10: Java Memory management .....	133
Chapter 11: Output based .....	150
Chapter 12: JAVA 8.....	158
Chapter 13: Java New Versions features .....	175

Chapter 14: Java Multithreading .....	180
Chapter 15 : Junit .....	219
Chapter 16: JAVA Spring/ SpringBoot .....	221
Chapter 17: Hibernate.....	294
Chapter 18: Java Messaging .....	312
Chapter 19: Java Microservices .....	326
Chapter 20: Java design patterns & principles .....	375
Chapter 21: Some common Database Questions.....	397
Chapter 22 : Common Java Stream Coding Problems.....	422
Chapter 23 : Fifteen Methods that interviewers love .....	433
Chapter 23: RESUME TIPS .....	439
Chapter 24: REAL INTERVIEW REPORTS .....	446
Chapter 25: My Story .....	491

# **Chapter 1: JAVA Interviews Evolution**

The nature of Java interviews has evolved significantly over the years, and understanding these changes is essential for candidates preparing for technical roles. The divide between junior and senior candidates is evident, but the expectations, especially around coding rounds, have shifted.

In the past, senior developers with significant industry experience were often exempt from coding rounds. However, this is no longer the case. Today, both junior and senior candidates must undergo a coding round, which has become mandatory in most organizations. This shift has been particularly noticeable in product-based companies, where Data Structures and Algorithms (DSA) proficiency is emphasized in the first round of interviews.

The purpose of this book is to guide you through the technical discussions and interview topics and give you the confidence that you have covered all possible topics in Java that can be discussed in the interview.

## Audience

- Junior Developers: Core Java, Object-Oriented Programming (OOP), and foundational Java concepts are the cornerstones of success.
- Experienced Developers: You must be proficient in all areas covered in this book, especially multithreading, memory management, and other advanced topics, particularly if you are aiming for roles in captive companies.

## What This Book Covers

This book delves into the Java technical discussions. It will cover critical Java concepts that are covered in the interviews. The chapters are divided based on areas of focus for junior and experienced developers:

- Core Java: Covering basic language features, syntax, and concepts that are crucial for junior developers.
- OOP Concepts: A deep dive into encapsulation, inheritance, polymorphism, and abstraction, which are essential for both junior and senior roles.
- Multithreading: Detailed coverage of threads, synchronization, concurrency, and parallelism, which are

particularly important for senior developers targeting captives.

- Memory Management: Understanding garbage collection, memory leaks, and the Java memory model is critical for experienced candidates.

- Spring and Hibernate: Framework-related questions that senior developers need to master, especially for product-based and enterprise-level applications.

- Java 8 and Beyond: With a focus on functional programming, streams, and lambdas, this chapter is vital for developers of all experience levels.

## Interview Focus Areas

### 1. For Junior Developers:

- Mastering core Java and OOP concepts is crucial. Expect questions about basic syntax, exception handling, and the fundamentals of how Java works under the hood.

- Understanding simple multithreading and basic memory management concepts may also come up, but these are not typically the main focus for junior roles.

### 2. For Experienced Developers:

- The entire spectrum of Java is important. From understanding the intricacies of memory management and garbage collection to handling concurrency and multithreading, experienced developers will be tested on how well they can apply their knowledge to solve real-world problems.

- Advanced concepts like Java 8 features, microservices architecture, design patterns, and cloud-based solutions (using Spring Boot, Hibernate, etc.) are often focal points.

- In interviews, multithreading and memory management are emphasized, as these areas are critical to developing efficient, scalable, and reliable systems.

*Prepare to dive into the world of Java interviews with a guide that is as insightful as it is practical. With "CRACKING THE JAVA INTERVIEWS WITH SUMIT," you are taking a significant step toward achieving your career goals and unlocking new opportunities in the ever-evolving tech industry.*

# **Chapter 2: OOPS**

Java is fundamentally an object-oriented programming language, and as such, object-oriented concepts are frequently tested in Java interviews. While traditional questions, such as those exploring the differences between interfaces and abstract classes, remain common, recent years have seen a shift towards more sophisticated queries that delve into advanced design principles and patterns. These questions are designed to assess a candidate's depth of understanding in object-oriented programming (OOP).

It's particularly important for Java interviews targeting developers with 1 to 3 years of experience, as this group is expected to be well-versed in OOP fundamentals, including key concepts like Abstraction, Inheritance, Composition, Class, Object, Interface, and Encapsulation. Understanding and applying these principles is crucial for any developer working within an object-oriented paradigm, making them a focal point during the interview process.

# **1. Which programming paradigms does Java support?**

Answer:

Below programming paradigms:

1. Object-Oriented Programming (OOP): Java emphasizes classes and objects, with support for inheritance, encapsulation, polymorphism, and abstraction.
  2. Imperative Programming: Java allows for programming with explicit statements and commands to change program state, using constructs like loops and conditionals.
  3. Procedural Programming: Java supports procedural code organization, where functions or methods operate on data and control program flow.
  4. Concurrent Programming: Java provides built-in support for multithreading and concurrency, using classes from the `java.util.concurrent` package.
  5. Functional Programming: Introduced in Java 8, Java supports functional programming with lambda expressions, the Stream API, and functional interfaces.
  6. Generic Programming: Java enables type-safe operations on objects through generics, allowing classes and methods to operate on specified types.
-

## **2. Why is Composition preferred over Inheritance?**

Answer:

**VERY IMPORTANT QUESTION** – Explain it with an example :

Let's say you're building an app with different types of users:

Regular users

Admins

SuperUsers with mixed powers

You might think inheritance is the way to go:

```
class User {  
    void login() {}  
    void logout() {}  
}  
class AdminUser extends User {  
    void deleteUser(User user) {}  
}
```

Looks simple, right?

But wait...

What if a SuperUser needs some admin powers, but not all?

What if a Guest should only be able to log in?

Suddenly, your inheritance tree starts to get messy and fragile.

## Can be solved with Composition:

```
class AuthService {  
    void login() {}  
    void logout() {}  
}  
class AdminPrivileges {  
    void deleteUser(User user) {}  
}  
class User {  
    AuthService auth = new AuthService();  
}  
class AdminUser {  
    User user = new User();  
    AdminPrivileges admin = new AdminPrivileges();  
}
```

Now you're not forcing roles into a rigid hierarchy. You're composing users from reusable parts.

## Benefits of Composition Here:

### 1. More Control

Add only the features each role really needs.

### 2. Easy to Extend

Want a ModeratorUser? Just plug in a ModeratorPrivileges class.

### 3. Avoid Deep Inheritance Trees

No guessing where a method comes from. Each part is clear and focused.

### 4. Reusable Components

AuthService works for all user types-write once, use everywhere.

### 5. Less Risky Changes

Updating AdminPrivileges won't accidentally break User

or Guest.

Inheritance says: "Admin is a User"

Composition says: "Admin has login ability + delete rights"

Composition takes a bit more setup but gives you a ton more freedom.

Summary: Composition is preferred over inheritance because:

1. Flexibility: You can change the behavior of a class at runtime by altering its components.
  2. Loose Coupling: Classes are less dependent on the internal details of other classes.
  3. Avoids Inheritance Issues: Prevents problems like the diamond problem and tight coupling.
  4. Encapsulation: Better encapsulates functionality and exposes only necessary parts.
  5. Code Reusability: Allows combining different components for varied behavior.
  6. Easier Maintenance: Changes to components do not affect the class using them, as long as the interface remains the same.
- 

### **3. Is Java a pure object-oriented language?**

Answer:

Java is not a pure object-oriented programming language

Java isn't purely object-oriented due to:

Primitive types (e.g., int, boolean) which aren't objects.

---

**Static members**, breaking encapsulation by not being tied to instances.

**Procedural constructs** like loops and conditionals outside object context.

**No multiple class inheritance**, only allowing interface implementation for multiple inheritance.

**Built-in operators** for primitives, not through object methods.

Smalltalk, Ruby, Eiffel, Self, and Newspeak are examples of pure object-oriented languages.

---

## 4. What is the method hiding in Java?

Answer:

When you declare two static methods with same name and signature in both superclass and subclass then they hide each other i.e. a call to the method in the subclass will call the static method declared in that class and a call to the same method in superclass is resolved to the static method declared in the superclass.

Here's a simple example to illustrate method hiding:

```
class SuperClass {  
    public static void display() {  
        System.out.println("SuperClass display method is called");  
    }  
}  
  
class SubClass extends SuperClass {  
    public static void display() {  
        System.out.println("SubClass display method is called");  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        SuperClass obj = new SubClass();  
        obj.display(); // This will call SuperClass's display method, d  
        SubClass.display(); // This will call SubClass's display method  
    }  
}
```

---

## 5. Can we override the static method in Java?

Answer:

No, you cannot override a static method in Java. Static methods belong to the class rather than an instance of the class.

If you define a static method with the same name and parameters in a subclass, it will hide the superclass's static method, not override it.

---

## 6. Can we override a private or final method in Java?

Answer:

**Private Methods: Private methods cannot be overridden** because they are not accessible outside the class they are defined in. They are implicitly final, meaning they are bound to the specific class in which they are declared

```

class Parent {
    private void show() {
        System.out.println("Parent show()");
    }
}

class Child extends Parent {
    void show() { // this is a NEW method, not overriding
        System.out.println("Child show()");
    }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        p.show(); // COMPILE TIME ERROR
    }
}

```

**Final Methods: Final methods cannot be overridden**  
because the final keyword prevents a method from being changed in any subclass.

```

class Parent {
    final void display() {
        System.out.println("Parent display()");
    }
}

class Child extends Parent {
    // ✗ Compile-time error: cannot override final method
    void display() {
        System.out.println("Child display()");
    }
}

```

## **7. Can we override constructor in Java?**

Answer:

**Constructors cannot be overridden** in Java. Overriding is applicable to methods that are inherited by the subclass.

Constructors are not inherited; each class has its own constructor, which is why they cannot be overridden.

However, constructors can be overloaded within the same class.

---

## **8. Can we override a non-abstract method in an abstract class?**

Answer:

**Yes, you can override a non-abstract method in an abstract class.** Abstract classes in Java can have both abstract and concrete methods.

A subclass can override the concrete methods to provide a more specific implementation.

Example:

```
abstract class AbstractClass {  
    void display() {  
        System.out.println("AbstractClass display");  
    }  
}
```

```
class ConcreteClass extends AbstractClass {  
    @Override
```

```
void display() {  
    System.out.println("ConcreteClass display");  
}  
}
```

---

## **9. Can we override a synchronized method with a non-synchronized one?**

Answer:

**Yes, you can override a synchronized method with a non-synchronized one, and vice versa.** However, doing so can lead to concurrency issues if the synchronization is necessary to ensure thread safety. Removing the synchronized modifier in an overridden method removes the thread safety guarantees provided by the original method.

---

## **10. Can we change the return type of method to subclass while overriding?**

Answer:

Yes, you can, but only from Java 5 onward. This feature is known as covariant method overriding and it was introduced in JDK 5 release.

Example:

```
class Animal {  
    public Animal getAnimal() {  
        return new Animal();  
    }  
}  
  
class Dog extends Animal {  
    // Covariant return type: Dog is a subclass of Animal  
    @Override  
    public Dog getAnimal() {  
        return new Dog();  
    }  
}
```

---

## 11. [FOLLOW- UP QUESTION] Explain Covariant Return Type in method overriding with example?

Answer:

Covariant return type allows a method in a subclass to override a method in the superclass and return a more specific type (a subtype of the original return type).

### Example:

```
class Animal {  
    Animal getAnimal() {  
        return this;  
    }  
}  
class Dog extends Animal {  
    @Override  
    Dog getAnimal() { // Covariant return type  
        return this;  
    }  
}
```

## **12. Can we make a class abstract without an abstract method?**

Answer:

**Yes**, we can make a class abstract in Java even if it doesn't contain any abstract methods.

---

## **13. Can we make a class both final and abstract at the same time?**

Answer:

No, we cannot apply both final and abstract keyword at the class at the same time because they are exactly opposite of each other.

A final class in Java cannot be extended and we cannot use an abstract class without extending and make it a concrete class. A compile time error will come.

```
// This would result in a compile-time error
public final abstract class SomeClass {
    // Class body
}
```

---

## **14. Can we overload or override the main method in Java?**

Answer:

Yes, we can overload the main method in Java.

While you can overload main, the JVM will only look for

and execute the public static void main(String[] args) method when starting the program. Other overloaded versions can be called from within the program but won't serve as entry points.

We can only overload main(), but we cannot override it because the static method cannot be overridden.

**Overloading main** : is possible, but only one version (String[] args) will serve as the entry point for the JVM.

**OVERRIDING main** : isn't possible due to its static nature, but subclasses can define their own main methods which act independently.

---

## **15. What is @Override annotation in method overriding? What would happen if you omit this annotation?**

Answer:

The @Override annotation indicates that a method is intended to override a method in a superclass. It helps catch errors at compile time if the method does not correctly override the superclass method (e.g., due to a typo or incorrect method signature). If omitted, the code will still compile, but there's a risk of accidentally creating an overloaded method instead of overriding it, leading to bugs.

---

## **16. Why can't you use return in a constructor? What would happen if you tried?**

Answer:

A constructor in Java cannot return a value, not even void. Constructors are meant to initialize an object, and their primary purpose is to create an instance of a class. If you try to use return in a constructor, the compiler will throw an error. The absence of a return type is one of the features that distinguish constructors from methods.

---

## **17. What happens if you call super() and this() in the same constructor?**

Answer:

In Java, both super() and this() must be the first statement in a constructor. Therefore, it is illegal to call both in the same constructor.

If you attempt to do so, the code will fail to compile. You can only use one of them in a constructor, depending on whether you are delegating to another constructor in the same class (this()) or calling the superclass constructor (super()).

Attempting to use both super() and this() in the same constructor will result in a compile-time error, similar to:

constructor call must be the first statement in a constructor

or

Call to 'super()' must be first statement in constructor

---

## 18. What happens if you call super() in a constructor of a class, but there is no explicit call to super()?

Answer:

In Java, when you make a:

**Explicit call to super()**: If you explicitly call super() in a constructor, it invokes the no-argument constructor of the superclass.

**No explicit call to super()**: If there's no explicit call to super() in the constructor of a class, Java automatically inserts a call to super() at the beginning of the constructor.

This means:

- If the superclass does not have a no-argument constructor, and you don't explicitly call another superclass constructor, you'll get a compile-time error because Java can't insert the default super() call.
- If the superclass has a no-argument constructor (either explicitly defined or provided by default if no constructors

are defined), then the compiler will insert super() at the start of your constructor .

```
class Superclass {
    public Superclass() {
        System.out.println("Superclass no-arg constructor");
    }
}

class Subclass extends Superclass {
    // No explicit super() call here, but Java would insert super()
    public Subclass() {
        System.out.println("Subclass constructor");
    }
}

public class Test {
    public static void main(String[] args) {
        new Subclass(); // Output will be:
                        // Superclass no-arg constructor
                        // Subclass constructor
    }
}
```

---

## 19. What is the result of comparing two objects with == if they are not the same instance but contain the same value?

Answer:

When comparing two objects in Java using the == operator:

**Reference Comparison:** The == operator checks for **reference equality**. It returns true if both variables point to the exact same object instance in memory, and false otherwise.

**Value Comparison:** If two objects contain the same value but are different instances, == will still return false because they are not the same reference.

For objects to be considered equal based on their values, you should use the `.equals()` method, which compares the contents of the objects according to the class's definition of equality.

The `==` operator should be used when you're checking if two variables refer to the same object instance in memory.

---

## **20. What will happen if you try to invoke a static method on a null object reference?**

Answer:

Invoking a static method on a null object reference in Java does not result in a `NullPointerException`.

Because:

- Static methods belong to the class itself, not to any instance of the class. They can be called without creating an instance of the class.
  - A null object reference means that the variable does not point to any object. However, for static methods, the object reference is not necessary because the method is associated with the class.
- 

## **21. Can we have multiple `main()` methods in a Java class?**

Answer:

Yes, you can have multiple `main()` methods in a class, but

they must have different method signatures (method overloading).

For example, you can have public static void main(String[] args) and public static void main(String arg1, String arg2). However, only the main method with the signature public static void main(String[] args) will be invoked when you run the program.

---

## **22. How does Java's garbage collector work when an object becomes unreachable but still has references?**

Answer:

Java's garbage collector focuses on reachability from GC roots (like local variables, static variables, and threads), not just the existence of references.

If an object isn't reachable from GC roots, it's eligible for collection, even if other unreachable objects reference it or it forms a cycle of references.

This ensures memory is freed from objects no longer needed in the program's execution.

---

## **23. Can We Define an Interface Within a Class?**

Answer:

Yes, you can define an interface within a class, known as a nested interface. This is useful when the interface is closely

related to the outer class and is not intended to be used elsewhere.

Example:

```
public class OuterClass {  
    public interface NestedInterface {  
        void someMethod();  
    }  
}
```

---

## 24. Can We Define a Class Within an Interface?

Answer:

Yes, you can define a class within an interface.

Such a class is implicitly static and public, allowing it to be instantiated without an instance of the interface.

Example:

```
public interface OuterInterface {  
    class InnerClass {  
        public void printMessage() {  
            System.out.println("Hello from  
InnerClass");  
        }  
    }  
}
```

---

## **25. Can an Interface Have Constructors?**

Answer:

No, interfaces cannot have constructors.

Constructors are used to initialize instances, but interfaces do not create instances themselves.

Why?

- Interfaces are not meant to be instantiated directly.
- They are meant to define abstract behavior (method signatures) that must be implemented by a class.

Key Points:

- Interfaces cannot maintain state, so constructors are meaningless.
  - A class implementing an interface is responsible for defining its own constructors.
- 

## **26. How do you decide if an inner class is required?**

Answer:

Use an inner class when:

1. Encapsulation: It tightly couples the class with its outer class.
2. Grouping: The class logically belongs to and is only used by the outer class.
3. Access: It needs to access private members of the outer class.

4. Convenience: It simplifies code by keeping related classes together.

#### Use Cases:

- Helper Classes: For utility classes that assist the outer class.
  - Event Handlers: When implementing event listeners tied to the outer class.
  - Builder Pattern: To create complex objects with an internal builder.
- 
-

# Chapter 3: Core Java

## 27. Explain System.out.println

Answer:

System.out.println is a method in Java used to print messages to the console.

It is a combination of System, which is a final class, out, which is a static final field of type PrintStream in the System class, and println, which is a method of PrintStream. The println method outputs the message followed by a newline character.

In short:

**System:** java.lang.System class.

**out:** Static PrintStream for console output.

**println:** Method to print and add a new line.

---

## 28. Why is System.out considered thread-safe?

Answer:

**System.out is thread-safe** because PrintStream, the class that System.out belongs to, synchronizes all its methods (ally).

Therefore, even in a multi-threaded environment, concurrent access to System.out.println is safe and won't result in interleaved or corrupted output.

---

## **29. Can we execute a program without main() method?**

Answer:

In Traditional Java: A public static void main(String[] args) method is required as the JVM's entry point for execution.

### Java 7+ with Static Blocks:

You can have code in static blocks that runs when the class is loaded, but if there's no main, the program will not continue running after static initialization.

### Java 11 Single-File Execution:

Java 11 introduced a feature where you can run a .java file directly:

```
java MyProgram.java
```

Here, Java implicitly treats the file as if there's a main method, useful for quick scripts or learning.

### Alternative Java Environments:

Applets: Run through lifecycle methods like init() and start(). No main needed.

Servlets: Use init(), service(), etc., managed by a servlet container.

JavaFX: Starts with Application.launch(), which internally might use a main but not explicitly in your code.

Frameworks and Custom Launchers: Some frameworks

might use bytecode manipulation or other JVM capabilities to define alternative entry points or simulate a main method.

In essence, while there are ways to execute some Java code or use Java in environments where main isn't directly used, for a standalone Java application to run and stay running, a main method or an equivalent entry point mechanism is typically necessary.

---

### **30. What happens if you run a Java program compiled with JDK 14 on a JDK 8 runtime?**

Answer:

If you run a Java program compiled with JDK 14 on a JDK 8 runtime, you will encounter an

**UnsupportedClassVersionError**, because the class file version generated by JDK 14 is not compatible with the JDK 8 runtime

---

### **31. Can you run a Java 8 compiled application on a JDK 11 runtime?**

Answer:

Yes, you can run a Java 8 compiled application on a JDK 11 runtime, as **Java provides backward compatibility**.

However, you should ensure that there are no dependencies on APIs or tools that have been removed or deprecated in

later JDK versions.

---

### **32. Can we use the throws keyword with run() or main()? Why?**

Answer:

Yes, both run() and main() can declare throws. However:

1. run() in threads: Declaring throws has no practical effect because exceptions are not propagated back to the calling thread. Instead, they are handled internally by the thread's run-time environment.

2. main() method: Declaring throws allows you to propagate checked exceptions to the JVM, which will terminate the program if not handled.

Example:

```
public static void main(String[] args) throws Exception {  
    throw new Exception("Checked exception");  
}
```

---

### **33. What happens if you call Thread.wait() in the main method?**

Answer:

Calling Thread.wait() directly in the main method (or any method) without proper synchronization will result in a java.lang.IllegalMonitorStateException at runtime.

---

## Why?

wait() is an instance method inherited from Object, not Thread.

You must call wait() inside a synchronized block on the object you're waiting on.

---

## **34. What is String Pool?**

Answer:

Don't go for a Java dev interview if you can't explain what String Pool is..

The String Pool is a special memory area inside the heap where Java stores string literals.

It's all about reusability and memory efficiency.

### How it works:

```
String s1 = "Java";
String s2 = "Java";
System.out.println(s1 == s2); // true
```

Both s1 and s2 point to the same object in the pool. No duplicates created.

### But when you do this:

```
String s3 = new String("Java");
System.out.println(s1 == s3); // false
```

new String() bypasses the pool and creates a new object in the heap.

### Why it exists:

Saves memory by reusing immutable strings.  
Improves performance for repeated string values.  
Thread-safe because strings are immutable.

### Follow up question:

If asked how to make a heap string use the pool, answer:

```
s3 = s3.intern();
```

This forces the string into the pool and returns the pooled reference.

---

## **35. What is Object class?**

Answer:

The Object class is the **root class of the Java class hierarchy**.

Defined in java.lang package.

All classes in Java (built-in or user-defined) directly or indirectly inherit from Object.

If a class does not explicitly extend another class, it implicitly extends Object.

Every class in Java has Object as its ultimate parent.

### Example:

---

```
class A {}      // implicitly extends Object  
class B extends A {}
```

Here, B → A → Object.

---

### 36. Name some important methods in Object class.

Answer:

In Java, every class extends java.lang.Object either directly or indirectly.

That means methods from Object are inherited by all classes.

These methods are not just utilities , they form the foundation of Java's design and often appear in interviews.

Let's go through them in detail :

#### 1. equals(Object obj)

Purpose: Compares two objects for equality.

- Default: behaves like == (checks references).
- Typically overridden to compare content.

Note:

- Always override hashCode() when overriding equals().
- Contract: reflexive, symmetric, transitive, consistent, null-check safe.

#### 2. hashCode()

Purpose: Returns an integer hash for objects (used in hash-

based collections).

- Default: typically derived from memory address.
- When overriding equals(), must override hashCode().

### Contract:

- Equal objects → must have same hashCode.
- Unequal objects → can have same hashCode (collision).

### Important to know:

- Breaking contract leads to issues in HashMap, HashSet.

## **3. toString()**

Purpose: Returns a string representation of object.

- Default: ClassName@HexHashCode.
- Commonly overridden for logging/debugging.

Best Practice: Always override in entities, DTOs, and loggable classes.

## **4. clone()**

Purpose: Creates and returns a copy of the object.

- Requires implementing Cloneable.
- Default: shallow copy (object fields copied by reference).

Deep Copy: Must be implemented manually.

### Follow-up Questions:

Why avoid clone()? → Confusing, shallow copy pitfalls.

Alternatives: copy constructors, builder patterns.

## **5. finalize() (Deprecated in Java 9, Removed in Java 18)**

Purpose: Cleanup before GC destroys object.

- Unreliable: GC may never call it.
- Caused performance & security issues.

Alternatives:

- try-with-resources (AutoCloseable)
- Explicit close() methods

Follow-up Question: Why deprecated? → Non-deterministic, replaced with modern resource management.

## **6. getClass()**

Purpose: Returns runtime Class<?> of an object.

- Useful for reflection, debugging.

## **7. wait(), notify(), notifyAll()**

Purpose: Inter-thread communication on shared objects.

- Must be used inside synchronized block/method.
- wait() → thread waits (releases lock).
- notify() → wakes one waiting thread.
- notifyAll() → wakes all waiting threads.

Follow-up Question: What if not used in synchronize → IllegalMonitorStateException.

---

## **37. What is the default hashCode() of an object in Java?**

**Answer:**

This is a common core Java interview question because it tests your understanding of object identity vs logical

equality, and how hashing works in collections.

In Java, if a class does not override hashCode(), it inherits the default implementation from java.lang.Object.

The **default hashCode() is a native method** (declared as public native int hashCode();).

Its value is typically derived from the object's internal memory address, but it is **not guaranteed to be the actual physical address**.

That's why two distinct objects usually have different hash codes, even if their content is identical.

### Example:

```
Object o1 = new Object();
```

```
Object o2 = new Object();
```

```
System.out.println(o1.hashCode()); // e.g. 2018699554
```

```
System.out.println(o2.hashCode()); // e.g. 1311053135
```

```
System.out.println(o1.equals(o2)); // false
```

Here, the two different objects produce two different hash codes by default.

### Important Points

1. Default hashCode() is consistent for the same object during its lifetime (unless the object is mutated in a way that overrides it).
2. It is not based on object content, but usually derived from the **memory reference**.
3. If you plan to use objects as keys in hash-based collections (HashMap, HashSet), you must override both equals() and hashCode() to ensure logical equality works correctly.
4. If you ever need the *original* identity-based hash code

(even if overridden), you can call **System.identityHashCode(obj)**.

By default, hashCode() from Object returns an integer typically related to the object's memory address, but it's not the actual address , it just ensures each object has a unique identifier unless overridden.

---

## 38. What are Marker Interfaces?

Answer:

An interface with no methods , its only job is to mark a class with some metadata.

It's like putting a tag on a box saying "Fragile" , the tag doesn't do anything itself, but whoever reads it knows how to handle it.

Famous Examples in Java:

Serializable:Marks that a class's objects can be serialized.

Cloneable: Marks that a class's objects can be cloned.

Remote (RMI): Marks that the class can be used in remote method calls.

---

## 39. Why is a char array preferred over String for storing passwords?

Answer:

Old interviewers still ask this question.

Char arrays can be modified and cleared after use, reducing the risk of sensitive data lingering in memory.

In contrast, String objects are immutable and can remain in memory longer, potentially exposing passwords.

Main reasons are:

**Mutability:** char[] can be cleared after use; String remains in memory.

**Memory Security:** You control when sensitive data is erased with char[].

**Garbage Collection:** Unpredictable with String, but you can immediately clear char[].

**String Pool:** Passwords in String might get pooled, increasing exposure risk.

**Best Practices:** Overwriting char[] aligns with secure coding for sensitive data.

---

#### **40. You are investigating an issue and have written a JUnit test - it fails as expected. So you start the debugger and step over several times, but the issue is not reproduced. Why can this happen?**

Answer:

This can happen due to:

1. Timing Issues: The debugger can alter the timing and concurrency behaviour, masking timing-related issues.
2. Side Effects: Debugging can change the state of the application or environment, affecting how the issue

manifests.

3. State Dependency: The issue may depend on specific conditions or data that are not present during debugging.

---

## 41. How do you create a custom annotation?

Answer:

A custom annotation is created using the @interface keyword.

It can include elements, which are essentially methods with optional default values.

### Add Meta-Annotations (Optional but Recommended)

- @Target – where the annotation can be applied (e.g., method, class).
- @Retention – how long the annotation is retained (e.g., runtime, compile-time).

### Example:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyCustomAnnotation {
    String value();
    int priority() default 1;
}
```

This example defines an annotation MyCustomAnnotation that can be applied to methods.

It has a value element and a priority element with a default value of 1.

## **42. What is the purpose of @Retention?**

Answer:

@Retention determines at what point annotation information is discarded.

There are three retention policies:

SOURCE: Annotations are retained only in the source code and discarded during compilation.

CLASS: Annotations are recorded in the class file but discarded by the JVM during runtime.

RUNTIME: Annotations are retained at runtime, allowing them to be accessed through reflection.

---

## **43. What does the @Target annotation do?**

Answer:

@Target defines the kinds of program elements to which an annotation can be applied.

Possible values include:

TYPE: Class, interface, or enum declaration

FIELD: Field (including enum constants).

METHOD: Method declaration.

PARAMETER: Parameter of a method or constructor.

CONSTRUCTOR: Constructor declaration.

LOCAL VARIABLE: Local variable declaration.

ANNOTATION TYPE: Another annotation.

---

#### **44. What is the difference between Class.forName() and ClassLoader.loadClass()?**

Answer:

##### **Class.forName():**

- Loads and initializes the class.
- Syntax: `Class.forName("com.example.MyClass");`
- Static blocks and fields are executed immediately.
- Use when you need the class fully ready to use, like in JDBC.

##### **ClassLoader.loadClass():**

- Loads the class only (no initialization).
- Syntax: `classLoader.loadClass("com.example.MyClass");`
- Class is initialized later, when it's first used.
- Use when you want to delay the initialization for better control.

Feature	<code>Class.forName(String className)</code>	<code>ClassLoader.loadClass(String name)</code>
Purpose	Dynamically load a class, including initialization.	Load a class without initializing it.
Class Loading	Loads the class if not already loaded.	Loads the class if not already loaded.
Initialization	Initializes the class (runs static initializers).	Does not initialize; initialization happens on first active use.
Usage	Commonly for JDBC drivers where initialization is needed to register drivers.	When checking if a class can be loaded without side effects of initialization.
Method Invocation	Invokes the <code>&lt;cinit&gt;</code> method, running static initializers.	Does not invoke <code>&lt;cinit&gt;</code> ; waits for first use.
Return	Returns <code>Class&lt;?&gt;</code> object.	Returns <code>Class&lt;?&gt;</code> object.
Example	<code>Class.forName("com.mysql.cj.jdbc.Driver");</code>	<code>ClassLoader loader = Thread.currentThread().getContextClassLoader(); Class&lt;?&gt; clazz = loader.loadClass("com.example.SomeClass");</code>

---

## 45. What are different types of class loaders in Java?

Answer:

1. Bootstrap ClassLoader: Loads the core Java libraries (e.g., rt.jar).

2. Extension ClassLoader: Loads the classes from the ext directory.

3. System/Application ClassLoader: Loads classes from the classpath.

4. Custom ClassLoaders : User defined class loaders.

Class Loader	Description	Responsibilities	Parent
Bootstrap Class Loader	The topmost class loader; implemented in native code.	Loads core Java libraries (e.g., <code>rt.jar</code> or <code>java.base</code> in Java 9+).	None (root of the hierarchy)
Extension Class Loader	Loads standard extensions from the <code>jre/lib/ext</code> directory or <code>java.ext.dirs</code> .	Responsible for loading extension libraries.	Bootstrap Class Loader
System or Application Class Loader	Also known as the "application class loader", it's the default class loader for the application.	Loads classes from the classpath, including user-defined classes.	Extension Class Loader
Custom Class Loaders	User-defined class loaders extending <code>ClassLoader</code> .	Used for custom class loading logic, like loading classes from non-standard locations or implementing special security measures.	Varies, but often System Class Loader

## 46. Is it possible to load a class by two ClassLoaders?

Answer:

Yes, a class can be loaded by different ClassLoader instances, resulting in separate class objects.

Class Isolation: Each ClassLoader loads its own version of the class, treating them as separate.

Use Case:

Hot Deployment: In environments where classes can be reloaded without restarting the application.

Security: Isolating untrusted code or ensuring separation between different application modules.

Implications: Instances from different ClassLoaders are not equal and method calls across them need special handling.

This can lead to issues if you expect a single class type but get different instances.

---

## **47. How can Java Reflection be used to break encapsulation?**

Answer:

Reflection can be used to access private fields, methods, and constructors by setAccessible(true).

It can lead to security risks, performance overhead, and fragile code that breaks with future updates.

It's used for frameworks or tools where dynamic access is necessary.

---

## **48. What are dynamic proxies in Java, and how can they be used?**

Answer:

Dynamic proxies in Java:

-Allow creating objects at runtime that implement interfaces. These objects can intercept method calls, adding behavior before or after the actual method execution.

How to Use:

Define an Invocation Handler: This handler processes method calls, adding pre/post actions.

Create Proxy: Use `Proxy.newProxyInstance()` with the handler to generate an object that implements the interface(s).

Applications:

AOP: Adding logging, security, or other functionalities without changing original code.

Testing: Creating mock objects.

Lazy Loading: Delaying resource-intensive operations.

Summary:

Dynamic proxies enable runtime modification of method behavior for any interface, facilitating advanced programming techniques like aspect-oriented programming.

---

**49. After deploying a new version of your Java application, you encounter `ClassNotFoundException` or `NoClassDefFoundError`. How would you troubleshoot and resolve this?**

Answer:

Verify that the required class is present in the deployed package (e.g., JAR files).

Check for conflicts between classes loaded by different classloaders, which might be caused by classloader hierarchies or duplicate classes in different jars.

Resolve the issue by adjusting classloader configurations, ensuring that the correct versions of dependencies are being used, and avoiding multiple versions of the same library in the classpath.

---

## 50. What performance optimizations have you implemented in your Java application?

Answer:

This is a very commonly asked Interview question and we should prepared to answer this.

You should answer something like this - I have implemented several performance optimizations, including:

1. Profiling and Monitoring: Used tools like VisualVM and JProfiler to identify and resolve performance bottlenecks.
2. Efficient Data Structures: Replaced inefficient data structures with more appropriate ones (e.g., using HashMap instead of ArrayList for frequent lookups).
3. Database Optimization: Improved query performance by optimizing indexes and using batch processing for database operations.
4. Caching: Implemented caching strategies (e.g., using @Cacheable annotation) to reduce redundant computations and database calls.

- 5. Concurrency Improvements:** Optimized multithreading and used concurrent collections to handle high-throughput scenarios efficiently

**PLEASE Try to include examples from your project**

---

- 51. You have a Java web server in production that got stuck: stopped writing logs and doesn't answer to HTTP requests. How would you investigate?**

Answer:

You can do following:

- 1. Check Logs:** Look for any errors or warnings before it stopped working.
  - 2. Thread Dumps:** Generate a thread dump to identify any deadlocks or threads stuck in long operations.
  - 3. Resource Usage:** Monitor CPU and memory usage to identify any resource exhaustion.
  - 4. Configuration and Dependencies:** Review server configurations and dependencies for issues.
- 

- 52. Explain shallow copy vs deep copy in the context of Java cloning.**

Answer:

**Shallow Copy**

Creates a new instance but does not create new instances

for referenced objects (like nested objects). Both the original and the copy will share references to these nested objects.

Changes to nested objects through one instance will affect the other.

### Deep Copy

Creates a new instance and recursively copies all the nested objects referenced by the original, ensuring each object is entirely independent.

Changes to one instance's nested objects will not affect the other instance.

### Summary

Shallow Copy: Efficient but not fully independent for nested objects.

Deep Copy: Fully independent but requires more effort to implement correctly.

---

## **53. [FOLLOW-UP] How would you implement a deep copy in Java?**

Answer:

A deep copy can be implemented by overriding the `clone()` method and recursively cloning all objects referenced by the original object.

```
public class Person implements Cloneable {  
    private String name;  
    private Address address;  
  
    public Person(String name, Address address) {  
        this.name = name;
```

```

        this.address = address;
    }

    @Override
    protected Object clone() throws
CloneNotSupportedException {
    Person cloned = (Person) super.clone();
    cloned.address = (Address) address.clone(); // Deep
copy
    return cloned;
}
}

public class Address implements Cloneable {
    private String city;
    public Address(String city) {
        this.city = city;
    }

    @Override
    protected Object clone() throws
CloneNotSupportedException {
        return super.clone();}}

```

---

## 54. What is **CloneNotSupportedException**?

Answer:

**CloneNotSupportedException** is a checked exception that is thrown when an object's `clone()` method is called but the object does not implement the `Cloneable` interface.

It ensures that only objects that explicitly allow cloning (by implementing `Cloneable`) can be cloned, providing some level of control over cloning operations.

```
class MyClass {  
    // Not implementing Cloneable  
}  
  
public class Main {  
    public static void main(String[] args) {  
        MyClass obj = new MyClass();  
        try {  
            obj.clone(); // This will throw CloneNotSupportedException  
        } catch (CloneNotSupportedException e) {  
            System.out.println("Cloning not supported: " + e.getMessage());  
        }  
    }  
}
```

---

## 55. What happens when `main` function is run?

Answer:

When you run a Java program, the JVM (Java Virtual Machine) performs the following steps:

### 1. Class Loading:

The `ClassLoader` loads the `.class` files into memory.

### 2. Bytecode Verification:

The bytecode verifier checks the code for any illegal access or violation of Java language rules.

### 3. Memory Allocation:

JVM allocates memory to objects in heap and stack.

### 4. Execution:

The `main` method is invoked by the JVM using the bootstrap loader. Threads are managed by the JVM.

### 5. Execution Engine:

The engine interprets the bytecode or compiles it into native machine code using the JIT (Just-in-Time) compiler.

---

## **56. How Java Really Works Under the Hood ?**

### **What happens after you write System.out.println("Hello")?**

**Answer:**

#### **Step 1: You Write the Code**

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hi");  
    }  
}
```

You save it as http://Hello.java.

#### **Step 2: Compile to Bytecode**

When you run:

```
javac http://Hello.java
```

The Java compiler (javac) converts the code into a .class file.

This file contains bytecode which is not machine code but a special platform-independent format understood by the JVM.

#### **Step 3: JVM Loads and Verifies the Code**

JVM = Java Virtual Machine. It runs your Java app.

When you run:

---

```
java Hello
```

The Java Virtual Machine (JVM) takes over.

The ClassLoader loads the .class file into memory.

The Bytecode Verifier checks the bytecode for:

- Type safety
- Illegal access
- Stack overflows
- Other security issues

This verification step ensures the code is safe to run and hasn't been tampered with.

#### Step 4: JVM Starts Executing the Code

Now JVM can run the bytecode in 2 ways:

- 1 . Interpreter (runs line-by-line, slow but immediate)
2. JIT Compiler (Just-In-Time compiles hot code into native code for speed)

#### What is JIT Compiler?

JIT means "compile just before using" so the JVM watches which parts of your code run a lot (hot spots), and compiles them to machine code.

Types of JIT compilers:

- C1 (Client) - Fast startup, fewer optimizations
- C2 (Server) - Slower compile, highly optimized
- Graal - Newer, Java-based, smarter & supports native images

Modern Java uses Tiered Compilation:

- Start with interpreter
- Then C1
- Then C2 or Graal (based on performance needs)

#### Step 5: Native Code Executes on the CPU

Once compiled, the JVM runs the native code directly on your machine's CPU.

The more the code runs, the faster it gets due to smarter optimizations by the JIT compiler.

---

**57. You have two classes, ClassA and ClassB, each dependent on the other. Both classes' constructors require the other class as a parameter. How would you resolve this circular dependency in Java?**

Answer:

This problem can be resolved by using a design pattern, such as dependency injection, or by refactoring the design to decouple the classes.

Here's an example of how to handle this via setter injection instead of constructor injection:

```
class ClassA {  
    private ClassB classB;  
    public ClassA() {}
```

```
public void setClassB(ClassB classB) {  
    this.classB = classB;  
}  
}  
  
class ClassB {  
    private ClassA classA;  
  
    public ClassB(ClassA classA) {  
        this.classA = classA;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        ClassA a = new ClassA();  
        ClassB b = new ClassB(a);  
        a.setClassB(b); // Circular dependency resolved  
        using setter  
    }  
}
```

Here, ClassA and ClassB do not depend on each other directly in their constructors.

Instead, ClassA is created first, then passed to ClassB, and ClassA gets a reference to ClassB through a setter method.

This avoids the circular dependency in constructors.

---

**58. Consider the following scenario where you have two interfaces with the same default method signature, but different method**

## **bodies. How would you resolve this diamond problem in Java when a class implements both interfaces?**

```
interface A {  
    default void doSomething() {  
        System.out.println("A's implementation");  
    }  
}  
  
interface B {  
    default void doSomething() {  
        System.out.println("B's implementation");  
    }  
}  
  
class C implements A, B {  
    @Override  
    public void doSomething() {  
        // How will you resolve the conflict here?  
    }  
}
```

**Answer:**

Java allows you to resolve this diamond problem by explicitly overriding the conflicting method and specifying which interface's default method you want to use.

```
class C implements A, B {  
    @Override  
    public void doSomething() {  
        A.super.doSomething(); // Call A's implementation  
        explicitly  
    }  
}
```

In this case, you are explicitly choosing A's implementation by using A.super.doSomething() inside the doSomething() method of class C.  
This resolves the ambiguity between the two interfaces.

---

## **59. What are ghost methods, and how do they relate to the Java compiler's optimization?**

Answer:

Ghost methods are methods that the Java compiler creates that aren't explicitly written in the source code.

They're often generated for:

- Accessing private members of an outer class from a nested class.
  - Synthetic methods for anonymous inner classes or lambda expressions.
- 

## **60. What is the role of ClassLoader in dynamic proxy creation How does this mechanism differ from creating a proxy via java.lang.reflect.Proxy?**

Answer:

While java.lang.reflect.Proxy is used for creating interfaces-based dynamic proxies:

**ClassLoader:** In dynamic proxy creation, a custom ClassLoader can be used to load classes at runtime,

allowing for the creation of proxy classes that might not exist at compile time. This approach is more flexible and can create proxies for classes, not just interfaces, but it's more complex to implement.

**Reflect.Proxy:** Easier to use for interface-based proxies, it internally uses a dynamic class loader but abstracts this complexity from the developer.

---

## 61. What is the role of the BufferedReader class. Why is it preferred over FileReader?

Answer:

BufferedReader is a class that provides buffering capabilities for reading text from character input streams. It enhances performance by reading larger blocks of data at once into a buffer, reducing the number of I/O operations needed to fetch characters one by one. This is particularly useful when reading large files, as it minimizes the time spent in I/O operations.

BufferedReader also includes methods like readLine(), which efficiently reads an entire line of text, making it much easier to work with text data compared to FileReader, which reads one character at a time. The buffering mechanism of BufferedReader significantly improves performance when dealing with text files.

---

## 62. What is NIO, and how does it differ from the standard I/O in Java?

Answer:

NIO (New Input/Output) is an advanced I/O API introduced in Java 1.4 that provides a more efficient and flexible framework for handling I/O operations compared to the traditional standard I/O.

The main differences include:

Non-blocking I/O: NIO allows threads to continue processing while waiting for I/O operations to complete, using selectors and channels. This is particularly useful for handling multiple connections (like in server applications) without being blocked on each operation.

Buffers: NIO uses buffers to hold data, allowing for efficient data manipulation and reducing the number of read/write operations.

File Channels: NIO introduces file channels that support memory-mapped files, allowing for more efficient file access and manipulation. Overall, NIO provides improved scalability and performance, particularly in high-throughput applications, by enabling asynchronous processing and better resource management.

---

---

# Chapter 4: static keyword

## 63. When and why to use static keyword?

Answer:

As Java devs, we all use static, but many still get confused about when and why to use it.

### What does static mean?

It simply means "belongs to the class, not to the object." So instead of every object creating its own copy, a static member is shared across all objects.

### Where can we use static?

#### 1. Static Variables (a.k.a. Class Variables)

One copy shared across all objects.

Example: Counting how many objects are created.

```
class Counter {  
    static int count = 0;  
    Counter() {  
        count++;  
    }  
}
```

#### 2. Static Methods

Can be called without creating an object.

Example: Math.max(), Collections.sort()

Can only access static variables and other static methods directly.

### **3. Static Blocks**

Runs once when the class is loaded.

Often used for initializing static variables.

### **4. Static Classes (Nested Classes only)**

A class inside another class can be marked static.

They don't need an outer class object to be created.

```
class Outer {  
    static class Inner {  
        void show() {  
            System.out.println("Static Nested Class");  
        }  
    }  
}
```

#### Why is static useful?

1. Saves memory (shared copy)
2. Utility/helper methods (Math, Collections)
3. Configurations or constants (static final)
4. One-time initialization

NOTE : Don't overuse it , too many static variables can make your code less flexible and harder to test.

---

### **64. How does the static keyword impact garbage collection in Java?**

[HINT: Discuss the lifecycle of static variables in terms of the memory model and the implications

for long-running applications.]

Answer:

Static variables in Java are stored in the method area (part of the heap in modern JVM implementations), and they have the same lifespan as the class itself, meaning they exist until the class is unloaded by the JVM.

In a long-running application, especially in a server environment, static variables can lead to memory leaks if they hold references to objects that are no longer needed but cannot be garbage collected.

This is because static variables remain in memory for as long as the class is loaded, even if the application no longer uses them.

In environments where classes are never unloaded (like in many application servers), this can result in memory being held indefinitely, contributing to potential **OutOfMemoryErrors**.

---

## 65. Can a static method be abstract ?

Answer:

No, a static method cannot be abstract.

- abstract methods are meant to be overridden by subclasses.
  - static methods belong to the class, not to an instance — and cannot be overridden in the traditional polymorphic sense.
-

So this is illegal:

```
abstract class MyClass {  
    abstract static void doSomething(); // Compilation Error  
}
```

Correct Usage:

- Use abstract for instance methods that must be overridden.
- Use static when no instance is needed — but you must provide a method body.

If asked in an interview, emphasize:

"static and abstract are conceptually incompatible. One requires a method body at compile-time, the other forbids it."

---

## 66. Can classes be static?

Answer:

Yes , inner classes can be static.

In Java, only **nested classes** (classes defined inside another class) can be marked static.

A **static nested class** does **not** require an instance of the outer class to be created.

---

## **67. When not to use static keyword in Java and what can be its disadvantages if used?**

Answer:

A question where many candidates go blank...

Let's discuss this:

### Disadvantages of static:

#### 1. Memory Usage:

Static variables stay in memory for the lifetime of the program. Excessive use can lead to higher memory usage.

#### 2. Tight Coupling:

Overusing static methods or variables can make code less flexible and harder to test.

#### 3. Cannot Access Instance Members Directly:

Static methods cannot directly access non-static fields or methods.

#### 4. Thread-Safety Issues:

Shared static variables can cause concurrency issues if not handled properly.

### When NOT to Use static:

#### 1. Instance Specific Data or Behavior:

If a field or method should have a separate value or behavior per object, do not make it static.

---

```
class Employee {
```

```
String name; // Instance-specific, should not be static  
}
```

## 2. Polymorphism Requirements:

Static methods cannot be overridden. Avoid static if you rely on runtime method overriding.

## 3. Tightly Coupled Design:

Overusing static can make classes rigid and hard to test, especially in unit tests.

## 4. Thread-Sensitive Data:

Avoid static variables for data that changes per thread unless properly synchronized, or you risk concurrency issues.

---

## **68. In what scenarios might using a static inner class be preferable to an instance inner class?**

[HINT: Discuss memory usage and design considerations]

Answer:

A static inner class is preferable when the inner class does not need access to the instance members of the outer class.

Since it doesn't hold an implicit reference to the outer class, it can help reduce memory overhead, particularly in scenarios where many instances of the inner class might be created.

---

This design is also beneficial when the inner class is meant to be a utility class or a helper that doesn't rely on the state of the outer class, making the code cleaner and more modular.

Use a **static inner class** when:

1. It doesn't need access to outer class's instance members
  2. You want to reduce memory usage and avoid hidden references.
  3. You're creating utility/helper classes that logically belong inside another class
  4. You want to group related code but maintain independence from an outer class instance
- 

**69. What will be the output of the below code:**

```
class Demo {  
    static {  
        System.out.println("Static block");  
    }  
    {  
        System.out.println("Instance block");  
    }  
    Demo() {  
        System.out.println("Constructor");  
    }  
}
```

```
public static void main(String[] args) {  
    System.out.println("Main starts");  
    new Demo();  
    new Demo();  
}  
}
```

Answer:

*Static block*

*Main starts*

*Instance block*

*Constructor*

*Instance block*

*Constructor*

Explanation:

The static block runs once when the class is loaded.

For each object creation (new Demo()):

The instance block runs first.

Then the constructor runs.

---

**70. What will be the output of the below code**

```
Parent {  
    static void display() {  
        System.out.println("Parent static display");  
    }  
}  
  
class Child extends Parent {  
    static void display() {  
        System.out.println("Child static display");  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Parent obj = new Child();  
        obj.display();  
    }  
}
```

Answer:

The output will be:

**Parent static display**

Explanation:

Static methods are not overridden; they are hidden.

The method called depends on the reference type. Since obj is a reference of type Parent, the display() method of Parent is called, not the one from Child.

---

---

# Chapter 5: Java Collections

## 71. What are the differences between ArrayList and LinkedList in Java? When would you choose one over the other?

Answer:

### ArrayList:

Underlying Structure: Resizable array.

Access Time: O(1) for get() and set() operations.

Insertion/Deletion: O(n) for add() or remove() operations due to shifting elements.

Use Case: Better suited for scenarios with frequent access and infrequent modifications.

### LinkedList:

Underlying Structure: Doubly-linked list.

Access Time: O(n) for get() and set() operations due to traversal.

Insertion/Deletion: O(1) for add() or remove() operations if the position is known (e.g., iterator-based).

Use Case: Better suited for scenarios with frequent insertions and deletions.

Aspect	ArrayList	LinkedList
Access Time	O(1) for random access by index	O(n) for accessing elements by index
Insertion/Deletion at End	O(1) amortized; might involve resizing	O(1) for both adding and removing
Insertion/Deletion at Start	O(n) due to shifting elements	O(1) for both operations
Memory Usage	More efficient for storing elements	More memory due to node references
Use Case	Best for frequent access by index	Ideal for frequent modifications at ends

## **72. How can you make ArrayList Immutable?**

Answer:

To make an ArrayList immutable in Java, you can follow one of these approaches:

1. `Collections.unmodifiableList()`: Wraps an existing ArrayList in an unmodifiable view.
  2. `List.of()`: Creates an immutable list directly (Java 9+).
  3. `Collectors.toUnmodifiable()`: Converts a stream into an immutable list (Java 10+).
  4. Custom Immutable Class: For maximum control, create your own immutable list wrapper.
- 

## **73. Can final ArrayList be changed ?**

Answer:

Yes, a final ArrayList can be changed in terms of its contents—you can add, remove, or modify elements. However, you cannot reassign the final variable to point to a different ArrayList or object.

Example 1:

```
Modifying Contents (Allowed)
final ArrayList<String> list = new ArrayList<>();
list.add("Apple"); // Allowed
list.add("Banana"); // Allowed
list.set(1, "Orange"); // Allowed
System.out.println(list); // Output: [Apple, Orange]
```

## Example 2:

Reassigning the Variable (Not Allowed)

```
final ArrayList<String> list = new ArrayList<>();
list.add("Apple");
```

```
// list = new ArrayList<>(); // Not allowed, causes a
compilation error
```

---

## 74. How can you make ArrayList synchronized?

Answer:

To make an ArrayList synchronized in Java, you can use the Collections.synchronizedList method.

This method returns a synchronized (thread-safe) list backed by the specified list.

```
public class SynchronizedArrayListExample {
    public static void main(String[] args) {
        // Create an ArrayList
        List<String> list = new ArrayList<>();

        // Make the ArrayList synchronized
        List<String> synchronizedList = Collections.synchronizedList(list);

        // Now you can use synchronizedList safely in a multi-threaded environment
        synchronizedList.add("First Element");
        synchronizedList.add("Second Element");
    }
}
```

Also discuss : Synchronization can impact performance, especially in highly concurrent scenarios, because each operation locks the entire list.

The synchronization provided by synchronizedList does not prevent structural modifications to the list during iteration. For such scenarios, you might need to use CopyOnWriteArrayList if multiple threads are reading and

rarely writing, or manually synchronize the iteration.

---

## 75. How to choose initial capacity in an ArrayList constructor in a scenario where the list is repeatedly cleared and reused?

Answer:

Small Capacity: May lead to frequent resizing.

Large Capacity: Avoids resizing but can waste memory.

Optimal Capacity: Balances between avoiding resizing and not wasting memory, leading to efficient reuse.

```
// Small initial capacity, frequent resizing
```

```
ArrayList<Integer> listSmall = new ArrayList<>(10);
```

```
// Large initial capacity, avoids resizing but may waste  
memory
```

```
ArrayList<Integer> listLarge = new ArrayList<>(1000);
```

```
// Balanced initial capacity for expected use, optimized  
performance
```

```
ArrayList<Integer> listOptimal = new ArrayList<>(100);
```

```
// Reusing the list
```

```
for (int i = 0; i < 100; i++) {  
    listOptimal.add(i);
```

```
}
```

```
listOptimal.clear(); // Retains capacity, no need to resize on  
reuse.
```

---

## **76. How does the LinkedList class implement the List, Deque, and Queue interfaces simultaneously?**

Answer:

LinkedList in Java implements List, Deque, and Queue by using a doubly-linked list:

**List:** Supports insertions, deletions, and access at any position, though not as fast for random access as ArrayList.

**Deque:** Provides methods for operations at both ends (addFirst/addLast, removeFirst/removeLast), perfect for LIFO or FIFO structures.

**Queue:** Implements FIFO behavior using Deque methods, where elements are added at the end and removed from the front.

This is achieved through:

**Linked Structure:** Each node links to next and previous, enabling efficient end operations.

**Polymorphism:** LinkedList objects can act as any of these interfaces.

**Method Implementation:** Covers all necessary methods for each interface, optimized for linked list mechanics.

---

## **77. How can we implement an LRU (Least Recently Used) cache using a LinkedList?**

Answer:

An LRU cache can be efficiently implemented using a combination of a doubly linked list and a hash map. The list maintains access order, while the hash map provides

$O(1)$  access and updates.

```
import java.util.*;  
  
public class LRUCache<K, V> {  
    int capacity;  
    Map<K, V> map = new HashMap<>();  
    LinkedList<K> order = new LinkedList<>();  
  
    public LRUCache(int capacity) { this.capacity = capacity;  
    }  
  
    public V get(K key) {  
        if (!map.containsKey(key)) return null;  
        order.remove(key);  
        order.addFirst(key);  
        return map.get(key);  
    }  
  
    public void put(K key, V value) {  
        if (map.containsKey(key)) order.remove(key);  
        else if (map.size() == capacity)  
            map.remove(order.removeLast());  
        map.put(key, value);  
        order.addFirst(key);  
    }  
  
    public static void main(String[] args) {  
        LRUCache<Integer, String> cache = new  
        LRUCache<>(2);  
        cache.put(1, "A"); cache.put(2, "B");  
        cache.get(1); // access 1  
        cache.put(3, "C"); // evicts 2  
        System.out.println(cache.map); // {1=A, 3=C}  
    }  
}
```

### **Explanation:**

HashMap stores key-value pairs for fast access.  
LinkedList tracks usage order (most recent at front).  
On get() or put(), move the key to the front.  
If capacity is full, remove the last key (least recently used).  
Ensures O(1) get and put (amortized).

---

### **78. In what scenarios might a LinkedHashSet outperform a TreeSet, and vice versa?**

Answer:

LinkedHashSet maintains insertion order and has faster performance for insertion and iteration due to its linked list implementation.

TreeSet maintains sorted order and is better suited for scenarios requiring sorted data.

#### LinkedHashSet is better when:

- You need to maintain insertion order.
- Operations like add, remove, and contains are frequent, aiming for O(1) average time.
- Iteration order is important.

#### TreeSet is better when:

- Elements need to be kept in sorted order.
  - You need efficient range queries or nearest element searches.
  - Sorting is more crucial than insertion order.
-

**79. You need to use a HashMap where the keys are complex objects, such as a Person class with attributes like name, age, and address. How would you design this key class to ensure that it works correctly in a HashMap?**

Answer:

To design the Person class as a key in a HashMap:

1. override the hashCode() method to generate a consistent hash code based on significant fields, such as name and age.
  2. override the equals() method to compare the significant fields and ensure that two Person objects are considered equal only if they have the same name, age, and address
  3. ensure that these fields are immutable to maintain consistent behavior of the hash code and equality over the lifetime of the key.
- 

**80. [FOLLOW-UP QUESTION] What are the potential issues with using mutable objects as keys in a HashMap?**

Answer:

When you use a mutable object as a key in a HashMap, if the object's state changes (i.e., its hash code or the result of equals() changes), it might no longer be found in the map because the hash code used to store it is no longer the same

as the hash code used to retrieve it. This is because **HashMap uses the hash code to determine the bucket** where the key-value pair should be stored, and if the hash code changes, **the key might be looked for in the wrong bucket.**

In below example, we first add a MutablePerson to the HashMap with the key “John”, 30. Then, we change the age of this key to 31. When we try to retrieve the value using the modified key, the behavior might be inconsistent because the hash code has changed.

When we try to retrieve with a new MutablePerson object with the same attributes post-modification, it fails because equals() and hashCode() do not match the original key’s state when it was added to the map.

```

public class MutablePersonExample {
    public static void main(String[] args) {
        // Mutable Person class
        class MutablePerson {
            private String name;
            private int age;
            public MutablePerson(String name, int age) {
                this.name = name;
                this.age = age;
            }
            public void setName(String name) {
                this.name = name;
            }
            public void setAge(int age) {
                this.age = age;
            }
            @Override
            public int hashCode() {
                return Objects.hash(name, age);
            }
            @Override
            public boolean equals(Object obj) {
                if (this == obj) return true;
                if (obj == null || getClass() != obj.getClass()) return false;
                MutablePerson person = (MutablePerson) obj;
                return age == person.age && Objects.equals(name, person.name);
            }
            @Override
            public String toString() {
                return "MutablePerson{name='" + name + "', age=" + age + "}";
            }
        }
        // Using MutablePerson as a key in HashMap
        Map<MutablePerson, String> map = new HashMap<>();
        MutablePerson key = new MutablePerson("John", 30);
        map.put(key, "Developer");
        // Modifying the key after it's been added to the map
        key.setAge(31);
        // Trying to retrieve the value with the modified key
        System.out.println(map.get(key)); // Might print null or "Developer" depending on JVM implementation
        // Creating a new MutablePerson object with the same attributes as the modified key
        MutablePerson newKey = new MutablePerson("John", 31);
        System.out.println(map.get(newKey)); // Will print null because hash code and equality check fail
    }
}

```

Solution:

To mitigate this issue, you can Use Immutable Objects: Make the Person class immutable by making fields final and removing setters:

```
public class ImmutablePerson {  
    private final String name;  
    private final int age;  
  
    public ImmutablePerson(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    // No setters, only getters  
    // Override hashCode() and equals() as before  
}
```

---

## 81. [FOLLOW-UP QUESTION] What would happen if you override only the equals() method and not hashCode() in a custom key class used in HashMap?

Answer:

Overriding only the equals() method and not hashCode() can lead to inconsistent behavior.

Two keys that are considered equal by equals() might not have the same hash code, causing them to be placed in different buckets.

This inconsistency can result in incorrect retrieval of values and can break the contract of the HashMap, leading to subtle bugs.

## **82. How would you implement a thread-safe HashMap without using ConcurrentHashMap?**

Answer:

You can make a HashMap thread-safe by:

1. Using Collections.synchronizedMap(new  
HashMap<>()), which wraps the HashMap with  
synchronized methods.
2. Manually synchronizing access to the HashMap by  
using synchronized blocks around the critical sections of  
code that access or modify the map.

Implementing your own lock mechanisms, like using  
ReentrantLock, to control access to the HashMap at a finer  
granularity (e.g., segmenting the map).

---

## **83. What is a hash collision in Java?**

Answer:

A hash collision occurs when two different keys return the same hash code value. Since Java's hashCode() method returns a 32-bit integer, the number of possible objects far exceeds the range of integers, so collisions are inevitable.

Example:

```
String k1 = "FB";
String k2 = "Ea";
System.out.println(k1.hashCode()); // 2236
System.out.println(k2.hashCode()); // 2236
```

Here, both "FB" and "Ea" produce the same hash code

even though they are different strings.

Follow-up Question:

So, if two keys have the same hash code, does that mean they are equal?

Not necessarily, hashCode() being the same only places them in the same bucket. After that, Java uses the equals() method to determine whether the two keys are actually equal.

Follow-up Question:

How does HashMap handle collisions internally?

Before Java 8: collisions were resolved using chaining with a linked list. All colliding entries were placed in the same bucket's linked list, and lookups required traversing that list. In the worst case, it could degrade to  $O(n)$  performance.

From Java 8 onward: if a bucket gets too many collisions (more than 8 entries), the linked list is converted into a red-black tree. (Covered in next question)

---

## **84. What were the changes made to HashMap implementation in Java 8?**

Answer:

In Java 8, HashMap was optimized to improve performance when there are many hash collisions. Instead of using a linked list for bucket storage, Java 8 introduced a balanced tree (red-black tree) structure when the number of elements in a bucket exceeds a certain threshold (default is 8).

This change improves the worst-case time complexity for

operations from  $O(n)$  to  $O(\log n)$ . Additionally, if the bucket size reduces below a certain threshold (default is 6), it converts the tree back to a linked list.

---

## **85. Can you store null keys or values in a TreeMap?**

Answer:

No, you cannot store null keys in a TreeMap because it relies on Comparable or Comparator to sort the keys, and comparing null with any other key would result in a NullPointerException. However, TreeMap allows null values because the sorting only involves keys.

---

## **86. What is the difference between HashMap and IdentityHashMap in terms of how they handle keys?**

Answer:

The key difference between HashMap and IdentityHashMap is in how they compare keys:

HashMap uses the equals() method to compare keys and the hashCode() method to determine the bucket location.

IdentityHashMap uses the == operator to compare keys, meaning that keys are considered equal only if they are the same instance (reference equality).

This is useful in scenarios where logical equality is not

sufficient, and you need to distinguish between different instances of the same value.

---

## **87. How ConcurrentHashMap is Different from HashMap ?**

Answer:

The key differences between ConcurrentHashMap and HashMap in Java:

### **1. Thread Safety:**

HashMap: Not thread-safe. If multiple threads access a HashMap concurrently, and at least one of the threads modifies the map structurally, the behavior is undefined unless external synchronization is provided.

ConcurrentHashMap: Designed for high concurrency. It allows concurrent reads and a certain level of concurrent writes without external synchronization. It uses a divide-and-conquer technique where the map is divided into segments, and each segment can be locked separately, reducing contention.

### **2. Performance Under Concurrency:**

HashMap: If used in a multithreaded environment without proper synchronization, operations like get and put might lead to race conditions or lost updates.

ConcurrentHashMap: Provides better performance under concurrent access scenarios. It allows concurrent reads to occur while a write is in progress, and multiple writes can happen concurrently if they target different segments of the map.

### **3. Null Keys and Values:**

HashMap: Allows one null key and multiple null values.

ConcurrentHashMap: Does not allow null keys or values since version 1.8 (prior to that, it allowed one null key).

### **4. Fail-Fast vs. Fail-Safe:**

HashMap: Uses fail-fast iterators. If the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException.

ConcurrentHashMap: Uses weakly consistent iterators.

These do not throw ConcurrentModificationException, and they reflect the state of the map at some point at or since the creation of the iterator. Elements added during iteration might be reflected, but elements removed after the iterator is created might be still shown.

### **5. Implementation:**

HashMap: Uses a single lock for the entire map, which can lead to contention in a multi-threaded environment.

ConcurrentHashMap: Employs a more advanced mechanism with multiple locks for different parts of the map (segment locking), or in newer versions, uses fine-grained locking and an atomic update mechanism

### **6. Iteration:**

Iterating over a ConcurrentHashMap is safer under concurrent modifications compared to HashMap, though changes might not be immediately visible in iterations.

Below table provides a quick difference between HashMap and ConcurrentHashMap.

Feature	HashMap	ConcurrentHashMap
Thread Safety	Not thread-safe (external synchronization needed)	Thread-safe, optimized for concurrency
Null Keys/Values	Allows one null key and multiple null values	Null keys/values not allowed (since Java 8)
Performance in Concurrency	Poor under concurrent modifications	Excellent, supports concurrent reads and writes
Iterator Behavior	Fail-fast; throws <code>ConcurrentModificationException</code> on structural modification	Weakly consistent; does not throw exceptions due to concurrent modifications
Lock Mechanism	Single lock for the entire map	Multiple locks (segment locking) or fine-grained locking
Suitable For	Single-threaded applications or manually synchronized multi-threaded scenarios	Multi-threaded applications where concurrent access is common

## 88. What is Map.Entry?

Answer:

Candidates use Map.Entry in coding round for iterating over Hashmap but are not able to explain what it is.

Map.Entry is a static nested interface within the Map interface in Java. It represents a key-value pair, or an entry, in a Map.

When you iterate over a Map's entries, you're working with

objects that implement the Map.Entry interface. Each Map.Entry object contains one key and one corresponding value from the Map.

Example in code:

```
public class Example {  
    public static void main(String[] args) {  
        Map<String, Integer> map = new HashMap<>();  
        map.put("one", 1);  
        map.put("two", 2);  
  
        for (Map.Entry<String, Integer> entry : map.entrySet()) {  
            System.out.println("Key = " + entry.getKey() +  
                               ", Value = " + entry.getValue());  
        }  
    }  
}
```

---

## 89. How does Collections.sort() work internally?

Answer:

Collections.sort() method internally uses a modified version of Merge Sort, known as **Timsort**.

Timsort is a hybrid sorting algorithm derived from Merge Sort and Insertion Sort. It takes advantage of existing order in the data (for example, runs of consecutive elements that are already sorted). This makes it more efficient than Merge Sort in real-world data, especially when the data contains pre-sorted sequences.

The main steps of Timsort are:

1. Breaking the list into small segments (called "runs") and

sorting them using Insertion Sort.

2. Merging the runs using a Merge Sort-like approach.

Timsort is used because of its stable sorting behavior and optimized performance for partially sorted data, which is common in practical scenarios.

---

## 90. What would happen if you try to sort a list containing null elements using Collections.sort()?

Answer:

NullPointerException will be thrown unless a custom Comparator is provided that can handle null values.

For example, you can create a Comparator that treats null as less than any non-null value, or vice versa, to avoid this exception.

Here's an example with a Comparator that places null at the end:

```
List<String> list = Arrays.asList("b", "a", null, "c");
Collections.sort(list, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        if (s1 == null) return (s2 == null) ? 0 : 1;
        if (s2 == null) return -1;
        return s1.compareTo(s2);
    }
});
System.out.println(list); // Output might be [a, b, c, null]
```

## **91. Can you sort a list of custom objects using Collections.sort() without providing a Comparator?**

Answer:

Yes, you can sort a list of custom objects without providing a Comparator if the objects implement the Comparable interface.

The Comparable interface requires the class to implement the compareTo() method, which defines the natural ordering of the objects. Collections.sort() will use this natural order to sort the list.

If the objects do not implement Comparable, a ClassCastException will be thrown.

---

## **92. What is the difference between using Collections.sort() and Stream.sorted() in Java 8+ ?**

Answer:

Collections.sort() sorts the list in place, modifying the original list.

On the other hand, Stream.sorted() returns a new stream that is sorted according to the provided comparator or the natural order. It does not modify the original list.

Stream.sorted() is more suitable for use cases where you want to keep the original list unmodified or when working with parallel streams to potentially improve performance

with large datasets.

Feature	<code>Collections.sort()</code>	<code>Stream.sorted()</code>
<b>Operation</b>	In-place sorting, modifies the original list.	Creates a new sorted stream, does not alter the source.
<b>Return</b>	<code>void</code> , no return value; list is sorted directly.	Returns a <code>Stream&lt;T&gt;</code> , which can then be collected into a new collection.
<b>Performance</b>	Efficient for large collections; in-place sort.	Might have overhead due to creating a new stream; less efficient for very large datasets.
<b>Mutability</b>	Mutates the original collection.	Keeps the original data intact, only creates a new sorted view.
<b>Use Case</b>	Ideal for direct, in-place sorting of lists.	Suited for stream pipelines, where sorting is part of multiple operations.

# Chapter 6: Exception Handling

## 93. Explain Exception Hierarchy in Java.

Answer:

Throwable is the root class of the exception hierarchy:

Error: Represents severe JVM errors that are generally not recoverable. Example: StackOverflowError,

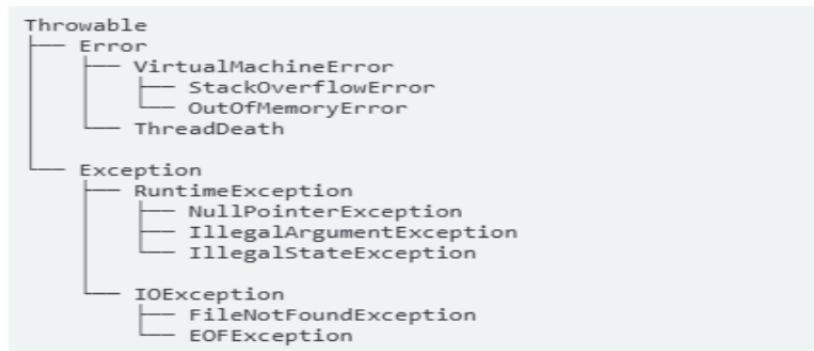
Exception: Represents conditions that the application might want to catch.

RuntimeException: Unchecked exceptions, not required to be declared or handled.

Examples: NullPointerException.

Checked Exceptions: Must be either handled or declared in the throws clause.

Example: IOException



Type of Exception	Exceptions
<b>Runtime Exceptions</b>	ArithmaticException, ArrayIndexOutOfBoundsException, ClassCastException, ConcurrentModificationException, IllegalArgumentException, IllegalStateException, IndexOutOfBoundsException, NullPointerException, NumberFormatException, UnsupportedOperationException
<b>Checked Exceptions</b>	ClassNotFoundException, CloneNotSupportedException, InterruptedException, IOException, NoSuchMethodException, NoSuchFieldException, SecurityException, SQLException, FileNotFoundException, EOFException

## 94. Can you have a try block without a catch block?

Answer:

Yes, but in that case you must have a finally block or a try-with-resources statement.

Example:

```
try {
    System.out.println("In try block");
} finally {
    System.out.println("In finally block");
}
```

## **95. What happens if you do not handle a checked exception?**

Answer:

Checked exceptions must be either:

- Caught using a try-catch block, or
- Declared in the method signature using throws.

If you do neither, the code will not compile — you'll get a compile-time error.

If you declare it but don't catch it, the exception will propagate to the caller.

If the exception reaches the main method and is still unhandled, the program will terminate and print a stack trace.

### **Key Rule:**

Java forces you to deal with checked exceptions at compile time to encourage safe error handling.

---

## **96. Can a constructor throw an exception?**

Answer:

Yes, constructors can throw exceptions.

Constructors are just special methods used to create objects. If something goes wrong during object creation (e.g., invalid input, resource access failure), the constructor can:

- Throw a checked or unchecked exception, just like any method.

- Declare checked exceptions in its signature using throws. However, if an exception is thrown, the object instantiation fails.

```
public MyClass() throws IOException {  
    throw new IOException("Constructor Exception");  
}
```

---

## 97. Can we have multiple catch blocks for a single try block?

Answer:

Yes, and the order of catch blocks matters. The more specific exception should come first.

### Key Rules:

- Catch blocks are evaluated top to bottom.
- The first matching catch block is executed.
- More specific exceptions must come before general ones (e.g., IOException before Exception).

```
try {  
    int a = 10 / 0;  
} catch (ArithmaticException e) {  
    System.out.println("ArithmaticException caught");  
} catch (Exception e) {  
    System.out.println("General Exception caught");  
}
```

---

## 98. Can an exception be thrown from the static block?

**Answer:**

Yes, exceptions can be thrown from the static block, and it will prevent the class from loading. The program won't execute the main method.

**Example:**

```
static {  
    if (true) {  
        throw new RuntimeException("Static block exception");  
    }  
}  
public static void main(String[] args) {  
    System.out.println("Main method");  
}
```

---

## 99. How to avoid NullPointerException?

**Answer:**

NullPointerException is an unchecked exception that occurs when an application attempts to use null in a case where an object is required. This includes calling a method on a null object, accessing or modifying a field of a null object, or taking the length of null as if it were an array.

Ways to avoid NullPointerException:

1. **Null Checks:** Always check for null before accessing an object's methods or properties.
2. **Optional Class:** Use Optional (introduced in Java 8) to handle potentially null values without risking a NullPointerException.
3. **Default Values:** Provide default values to avoid returning null

- 
4. Avoiding null Assignments: Prefer using empty objects, empty collections, or special "no-value" objects instead of null.

---

## 100. Can a finally block be skipped in any case?

Answer:

No, the finally block cannot be skipped under normal circumstances. It is always executed after the try and catch blocks, regardless of whether an exception is thrown or not.

However, there are some scenarios where the finally block might not execute:

1. If the JVM crashes or exits (`System.exit(0)`). If the thread executing the finally block is interrupted or killed.
2. If the code enters an infinite loop or an uncaught exception occurs in a previous block

---

## 101. Can an Error be caught in Java? Should it be caught?

Answer:

Yes, an Error can be caught in Java because it is a subclass of `Throwable`, just like `Exception`. However, it is generally not recommended to catch an Error.

Errors represent serious problems that a reasonable application should not try to catch.

These are usually related to the Java Virtual Machine (JVM) running out of resources, like OutOfMemoryError, StackOverflowError, or other fatal conditions that the application cannot recover from.

Catching such errors can lead to unpredictable behavior and is typically considered bad practice.

The application might not be in a stable state after such an error occurs.

---

## **102. How does the try-with-resources statement work in Java?**

**Answer:**

The try-with-resources statement in Java, introduced in Java 7, is a form of the try statement that automatically closes resources when the try block exits.

Any object that implements `java.lang.AutoCloseable` (which includes `java.io.Closeable`) can be used in a try-with-resources statement.

The resource is closed automatically at the end of the statement, regardless of whether the try block completes normally or abruptly (due to an exception).

**Example:**

```
try (BufferedReader br = new BufferedReader(new  
FileReader("file.txt"))) {  
    String line;  
    while ((line = br.readLine()) != null) {  
        System.out.println(line);  
    }  
}
```

```
    } catch (IOException e) {  
        e.printStackTrace();  
    }
```

Here, BufferedReader is automatically closed after the try block, even if an exception occurs.

---

### 103. If a method throws NullPointerException in the superclass, can we override it with a method that throws RuntimeException?

Answer:

Yes, RuntimeException is a superclass of NullPointerException, so a method overriding one that throws NullPointerException can throw RuntimeException or its subclasses.

```
class SuperClass {  
    void doSomething() throws NullPointerException {  
        // Implementation that might throw NullPointerException  
    }  
}  
  
class SubClass extends SuperClass {  
    @Override  
    void doSomething() throws RuntimeException {  
        // This is allowed because you're throwing an unchecked exception  
        // that's broader than NullPointerException  
    }  
}
```

---

### 104. What will be the output?

```
Parent {  
    void show() throws IOException {  
        System.out.println("Parent");  
    }  
}  
  
class Child extends Parent {  
    @Override  
    void show() throws FileNotFoundException {  
        System.out.println("Child");  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Parent obj = new Child();  
        try {  
            obj.show();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Answer:

The output will be: **Child**

The Child class's show() method can throw FileNotFoundException, which is a subclass of IOException. In overridden methods, the child method can declare the same exceptions or more specific ones.

---

## 105. What will be the output of this code?

```
public class Test {  
    public static void main(String[] args) {  
        System.out.println(testMethod());  
    }  
  
    static int testMethod() {  
        try {  
            return 1;  
        } finally {  
            return 2;  
        }  
    }  
}
```

Answer:

**The output will be:2**

Even though try block has a return statement, the finally block will override it.

The value returned by the finally block is returned as the final result of the method.

---

## 106. What is the difference between **NoClassDefFoundError** and **ClassNotFoundException**?

Answer:

Very old question but still get asked.

- **NoClassDefFoundError:** Thrown if a class was present at compile-time but is missing at runtime. This can occur if the class file is deleted or corrupted.
- **ClassNotFoundException:** Thrown when attempting to load a class dynamically using Class.forName() or similar methods, and the class is not found in the classpath.

Aspect	NoClassDefFoundError	ClassNotFoundException
Type	Error	Exception
Cause	Class missing at runtime but present during compile-time.	Class not found when loading via name at runtime.
Scenario	Classpath issues post-compilation.	Dynamic loading with incorrect or missing class name.
Handling	Not catchable; ensure class availability at runtime.	Catchable; use try-catch for handling.
Example	<pre>NoClassDefFoundError: MissingClass</pre>	<pre>try {     Class.forName("Nonexistent"); } catch (ClassNotFoundException e) { ... }</pre>

---

## 107. Logs say OutOfMemoryError - how would you investigate?

Answer:

You can talk about below points if this question is asked in an interview:

1. Heap Dump Analysis: Analyze heap dumps using tools like Eclipse MAT or VisualVM to find memory leaks.
  2. Memory Usage Patterns: Check memory usage patterns and garbage collection logs for unusual behavior.
  3. Code Review: Review code for potential leaks or inefficient memory usage.
  4. Increase Heap Size: Temporarily increase heap size to confirm if it's a memory-related issue.
- 

**108. User clicks on a button and gets NullPointerException, but same case works correctly in development environment. There are no log statements in code that help analyze the problem. How would you investigate and what step would you take to ease debugging such problems in the future?**

Answer:

You can talk about following:

1. Reproduce in Staging: Try to reproduce the issue in a staging environment with similar data and configuration.

2. Add Logging: Add detailed logging around the suspected areas to capture the context of the error.

3. Error Handling: Improve error handling and validation to provide more informative error messages.

4. Use Monitoring Tools: Implement APM tools to capture runtime errors and performance metrics.

---

## **109. How do you create a custom exception in Java?**

Answer:

To create a custom exception, extend either `Exception` (for checked exceptions) or `RuntimeException` (for unchecked exceptions).

### // Custom Checked Exception

```
public class MyCustomCheckedException extends  
Exception {  
    public MyCustomCheckedException(String message)  
    {  
        super(message);  
    }  
}
```

### // Custom Unchecked Exception

```
public class MyCustomUncheckedException extends  
RuntimeException {  
    public MyCustomUncheckedException(String  
message) {  
        super(message);  
    }  
}
```

Use `throw` to trigger the exception, and handle it with a `try-catch` block if necessary.

---

## **110. How can you ensure that the original exception is preserved when handling exceptions in a CompletableFuture chain?**

Answer:

To preserve the original exception in a CompletableFuture chain:

#### Exception Chaining:

Use exception chaining by throwing a new exception in exceptionally or whenComplete that wraps the original exception.

```
future.exceptionally(ex -> {
    throw new RuntimeException("Custom message",
    ex);
});
```

Preserve Context: This way, the original stack trace and context are not lost, aiding in debugging.

---

## **111. What are Suppressed Exceptions?**

Answer:

This is a tricky Java interview question because most developers know about exceptions but haven't explored suppressed exceptions, which were introduced in Java 7 with try-with-resources.

When you use try-with-resources or nested try blocks, multiple exceptions can occur.

Primary Exception → the one thrown in the try block.

Secondary (Suppressed) Exception → the one thrown while closing the resource (in close() method).

By default, Java attaches the secondary exception to the primary exception instead of replacing it, so you don't lose the original cause.

These secondary exceptions are called suppressed

exceptions.

### Example Without Suppressed Exceptions

```
class MyResource implements AutoCloseable {  
    @Override  
    public void close() {  
        throw new RuntimeException("Exception in close()");  
    }  
}  
public class SuppressedExample {  
    public static void main(String[] args) {  
        try {  
            MyResource r = new MyResource();  
            throw new RuntimeException("Exception in try block");  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Here, only "Exception in try block" is printed, but the close() exception is lost.

### Example With Suppressed Exceptions (try-with-resources)

```
class MyResource implements AutoCloseable {  
    @Override  
    public void close() {  
        throw new RuntimeException("Exception in close()");  
    }  
}  
public class SuppressedExample {  
    public static void main(String[] args) {  
        try (MyResource r = new MyResource()) {  
            throw new RuntimeException("Exception in try block");  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
System.out.println("Primary Exception: " + e);
for (Throwable suppressed : e.getSuppressed()) {
    System.out.println("Suppressed: " + suppressed);
```

### Output

Primary Exception: java.lang.RuntimeException:  
Exception in try block  
Suppressed: java.lang.RuntimeException: Exception in  
close()

Now you see the primary exception is preserved and the close exception is marked as suppressed.

---

### **112. Which exception will be thrown ?**

```
public static void main(String[] args) throws Exception {
    try {
        throw new RuntimeException("fail");
    } finally {
        throw new Exception("from finally");}}
```

Answer:

Output: Exception in thread "main" java.lang.Exception:  
from finally

### Explanation:

The try block throws a RuntimeException("fail").

But **before that exception can propagate**, the finally block runs.

The finally block throws a **new checked exception**:  
Exception("from finally"). Java **suppresses the original exception** (RuntimeException("fail")) and throws the one from finally.

---

---

## Chapter 7: Enums

### 113. What is an enum in Java and How do you declare an enum in Java?

Answer:

An enum in Java is a special data type that enables for a variable to be a set of predefined constants. It's introduced in Java 5 and is implemented as a class type.

You declare an enum using the enum keyword similar to how you declare classes:

```
public enum Day {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
}
```

---

### 114. How does Java handle enum constants internally? Can you explain how they are represented in memory?

Answer:

Java handles each enum constant as a singleton instance of the enum class.

When an enum is compiled, Java creates a class that extends `java.lang.Enum`. Each constant, such as `Color.RED`, is a static, final instance of this class, created during class loading. Each constant has a name, ordinal (position), and, if specified, its own fields and methods.

Java also stores the constants in a static array, making it

easy to retrieve them via values().

This structure makes enum constants efficient, type-safe, and easily accessible.

---

## 115. What are some use cases where using enum is preferable over static final constants?

Answer:

Using enum is often better than static final constants in these cases:

Type Safety: enum restricts values to defined constants, preventing unintended values.

Grouping: enum organizes related constants in a single type, improving readability.

Custom Fields and Methods: Each enum constant can have its own fields and methods, unlike static final.

Utility Methods: enum provides values(), ordinal(), and valueOf() methods for easy iteration and lookup.

Serialization Safety: Ensures only one instance of each constant, even after deserialization.

Switch Compatibility: Works seamlessly with switch statements, making the code cleaner and more readable.

Use Case	Enum	Static Final Constants
Type Safety	Ensures type safety at compile-time; prevents typos.	No compile-time type safety; typos can lead to runtime errors.
Built-in Methods	Comes with <code>values()</code> , <code>valueOf()</code> , <code>ordinal()</code> for easy management.	No built-in methods; must be manually implemented.
Switch Statements	Ideal for switch; compiler checks for completeness.	Less safe in switch; no compile-time check for all cases.
Complex Constant Behavior	Each enum constant can have unique behavior or data.	All constants share class behavior; can't differentiate without extra structures.
Singleton Pattern	Simple and inherently thread-safe implementation.	More code needed for thread safety and serialization safety.
Serialization	Automatically serializable with guaranteed single-instance preservation.	Requires extra steps like <code>readResolve()</code> for singleton behavior in serialization.

## 116. Can an enum extend another class in Java?

Answer:

No, enums cannot extend another class in Java Because every enum in Java implicitly extends `java.lang.Enum`, and Java doesn't support multiple inheritance of classes.

---

However, an enum can implement interfaces.

---

## **117. How do you iterate over all values of an enum?**

Answer:

You can iterate over all values of an enum using the values() method, which returns an array of all enum constants.

Every Java enum gets a static values() method.

It returns an array of all enum constants in the order they were declared.

You can loop through it using a for-each loop.

Example:

```
for (Day day : Day.values()) {  
    System.out.println(day);  
}
```

---

## **118. What is the significance of the Enum<E extends Enum<E>> declaration in the Enum class?**

Answer:

The Enum<E extends Enum<E>> declaration enforces that each enum type is a subtype of Enum<E> where E is the specific enum type itself.

This ensures type safety by preventing incorrect

assignments and allowing the Enum class to provide methods that work specifically with the enum type.

It ensures **type safety** for enum constants and methods operating on them.

- E represents the actual enum type (e.g., Day, Color).
  - E extends Enum<E> ensures that only **enum types** can extend Enum.
- 

## 119. How can we implement singleton and strategy pattern using enum?

Answer:

An enum **singleton** is implemented as follows:

```
public enum Singleton {  
    INSTANCE;  
    public void doSomething() {  
        // logic here  
    }  
}
```

This approach is preferred because it is inherently thread-safe, provides serialization guarantees, and ensures that there is only one instance of the enum.

Enums can implement the **strategy pattern** by defining abstract methods that are implemented differently for each enum constant:

```
public enum Operation {  
    ADD {  
        @Override  
        public int apply(int a, int b) {  
            return a + b;  
        }  
    },  
    SUBTRACT {  
        @Override  
        public int apply(int a, int b) {  
            return a - b;  
        }  
    };  
  
    public abstract int apply(int a, int b);  
}
```

This allows each enum constant to encapsulate its behavior.

---

# **Chapter 8: Serialization**

## **120. What is Serialization?**

**Answer:**

Serialization in Java is the process of converting **an object into a byte stream** so that it can be easily stored in a file, sent over a network, or saved in a database.

Later, this byte stream can be converted back into the original object. That reverse process is called **deserialization**.

### **Why do we need it?**

- To save the state of an object permanently (for example, caching or saving session data).
- To send objects across a network in distributed systems.
- To persist data for later use.

### **How does it work?**

- A class must implement the `java.io.Serializable` interface (it is a marker interface with no methods).
- The `ObjectOutputStream` class is used for serialization.
- The `ObjectInputStream` class is used for deserialization.

### **Example:**

```
class Student implements Serializable {  
    int id;  
    String name;  
    Student(int id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
}
```

```
}

// Serialization
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("student.txt"));
out.writeObject(new Student(1, "Alex"));
out.close();

// Deserialization
ObjectInputStream in = new ObjectInputStream(new
FileInputStream("student.txt"));
Student s = (Student) in.readObject();
in.close();
```

### Important Points

1. Static fields are not serialized because they belong to the class, not the object.
2. Transient fields are skipped during serialization.
3. serialVersionUID is used to verify compatibility during deserialization.

### In short

Serialization = converting object to byte stream.

Deserialization = converting byte stream back to object.

---

## **121. What happens if you do not define serialVersionUID in a Serializable class?**

Answer:

Candidates go blank when asked what happens if we don't define serialVersionUID in Serializable class .

---

Let's understand it :

First, What is serialVersionUID ?

When a class implements Serializable, Java converts its objects into bytes (serialization) and back (deserialization). But what if the class definition changes after you serialized an object?

This is where serialVersionUID comes in.

The Role of serialVersionUID:

It is a unique identifier for a Serializable class. During deserialization, JVM checks if the serialVersionUID of the class matches the one stored in the serialized object.  
If they don't match -> InvalidClassException.

Example

```
class User implements Serializable {  
    private static final long serialVersionUID = 1L;  
    String name;  
}
```

Here, serialVersionUID = 1L is explicitly defined. Even if you recompile the class, old serialized objects can still be read, as long as fields are compatible.

If you don't define it, JVM will generate one automatically based on class structure.

The problem: even a small change (adding a field, changing method order) may change the generated UID,

breaking deserialization.

If serialVersionUID is not explicitly declared, Java will generate it at runtime based on various aspects of the class.

This generated value can differ between different Java compiler implementations, potentially leading to InvalidClassException during deserialization if the class structure changes.

**NOTE:**

Always declare serialVersionUID explicitly in Serializable classes.

It makes your class backward compatible across versions. If omitted, you risk random InvalidClassException when class changes.

---

**122. What is the transient keyword and how does it impact serialization**

Answer:

The transient keyword is used to indicate that a field should not be serialized.

During the serialization process, transient fields are skipped and their values are not included in the serialized byte stream.

---

**123. What are the differences between**

## Externalizable and Serializable interfaces?

Answer:

The Serializable interface is a marker interface with no methods, relying on the JVM's default serialization mechanism.

Externalizable, on the other hand, requires the implementation of `writeExternal()` and `readExternal()` methods, giving the developer full control over the serialization process. Externalizable can be more efficient, as it allows you to skip fields that do not need to be serialized.

Serializable	Externalizable
Automatic serialization via reflection.	Manual control over serialization process.
Limited control, uses <code>transient</code> for exclusion.	Full control over what and how to serialize.
Potentially slower due to reflection.	Can be optimized for performance.
Uses <code>serialVersionUID</code> for versioning.	No automatic versioning; manual management needed.
Easy to implement; no methods required.	Requires implementation of <code>writeExternal</code> and <code>readExternal</code> .

- **Serializable** for simple, automatic serialization needs.
- **Externalizable** when needing detailed control over serialization.

---

### 124. What is `readResolve()` method in serialization?

Answer:

The `readResolve()` method is used to replace the object read from the stream with another object during deserialization.

This is particularly useful for singleton classes or when you want to return a cached instance instead of creating a new one during deserialization.

```
public class Singleton implements Serializable {
    private static final Singleton instance = new Singleton();

    // Private constructor to prevent instantiation
    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }

    // Method to ensure that deserialization returns the same instance
    private Object readResolve() throws ObjectStreamException {
        return instance;
    }
}
```

---

## 125. Can you serialize static fields in Java?

Answer:

No, static fields are not serialized because they belong to the class, not to any specific instance. Static fields can be initialized to their default values or handled separately if needed.

---

## **126. What happens if an exception is thrown during the serialization process?**

Answer:

If an exception is thrown during serialization, the process will fail, and the partially serialized object may be left in an inconsistent state.

To avoid this, proper exception handling should be implemented, and any resources should be cleaned up.

Additionally, when using streams, you should close them in a finally block or use try-with-resources to ensure that all resources are properly managed.

---

## **127. What happens if your Serializable class contains a member which is not serializable? How do you fix it?**

Answer:

If your Serializable class contains a **non-serializable field**, Java will **throw a java.io.NotSerializableException at runtime** when you try to serialize the object.

Imagine your class has a field of a type that doesn't implement Serializable. When serialization reaches that field, it fails.

Fix:

1. Make the Field Serializable

Ensure the class of the field also implements Serializable.

2. Mark the Field as transient

---

Use transient if the field doesn't need to be serialized (e.g., loggers, DB connections).

### 3. Use Custom Serialization

---

**128. You have a parent class Parent and a child class Child. The parent class has a static field and the child class has instance fields. If the child class object is serialized and later deserialized, what happens to the static field in the parent class?**

Answer:

Static fields are not serialized, as they belong to the class, not the instance. When a class is deserialized, the static field will hold the value it had at the time of deserialization, not the value that the object had during serialization.

Example:

```
class Parent implements Serializable {  
    static String staticField = "Static Field";  
}  
  
class Child extends Parent {  
    private String childField;  
  
    public Child(String childField) {  
        this.childField = childField;  
    }  
}
```

```
public class SerializationTest {  
    public static void main(String[] args) throws Exception {  
        Child child = new Child("Child Field");  
  
        // Serialize the object  
        ObjectOutputStream out = new  
ObjectOutputStream(new FileOutputStream("test.ser"));  
        out.writeObject(child);  
        out.close();  
  
        // Change static field value after serialization  
        Parent.staticField = "Modified Static Field";  
  
        // Deserialize the object  
        ObjectInputStream in = new ObjectInputStream(new  
FileInputStream("test.ser"));  
        Child deserializedChild = (Child) in.readObject();  
        in.close();  
  
        // Static field is not serialized, so it keeps the latest value  
        System.out.println("Static Field: " +  
Parent.staticField); // "Modified Static Field"  
    }  
}
```

When serialized, the static field will hold "Modified Static Field", showing that static fields are not affected by the process of serialization.

---

---

# Chapter 9: Generics

## 129. What does Generics mean?

Answer:

Generics in Java allow you to write classes, interfaces, and methods that can work with different data types while providing type safety and reusability.

Without generics, you often need to cast objects, which can lead to runtime errors. With generics, type checks happen at compile time, making your code safer and cleaner.

### Why do we need Generics?

1. Type safety - prevents `ClassCastException` at runtime.
2. Code reusability - one generic class or method can work with any data type.
3. Readability - no need for explicit casting.

### Example without Generics:

```
List list = new ArrayList();
list.add("Hello");
String s = (String) list.get(0); // explicit cast
```

### Example with Generics:

```
List<String> list = new ArrayList<>();
list.add("Hello");
String s = list.get(0); // no cast needed
```

## Generic Class Example:

```
class Box<T> {  
    private T value;  
    public void set(T value) { this.value = value; }  
    public T get() { return value; }  
}
```

```
Box<Integer> intBox = new Box<>();  
intBox.set(100);  
System.out.println(intBox.get()); // 100
```

## Generic Method Example:

```
public <T> void print(T value) {  
    System.out.println(value);  
}  
  
print("Java"); // String  
print(123); // Integer
```

## Key Points

Generics are checked at compile time (stronger type checks).

Type parameters like `<T>`, `<E>`, `<K,V>` are just placeholders for actual types.

Helps avoid runtime errors by catching issues earlier.

## In short

Generics let you write flexible, reusable, and type-safe code in Java.

---

## **130. What is the difference between List<?>, List<Object>, and List<? extends Object>?**

**Answer:**

List<?>: A list that can hold elements of any type, but you cannot add elements to it except null (unknown type).

List<Object>: A list that can hold elements of any type, but it expects elements to be of type Object or a subclass.

List<? extends Object>: A list that can hold elements of a type that is a subclass of Object. It's similar to List<?> but indicates an upper bound of Object.

---

## **131. What is the difference between covariance and contravariance in Java generics ?**

**Answer:**

Covariance (? extends T): Allows a generic type to be a subtype of a specific type (T).

**Example:**

List<? extends Number> can hold List<Integer>, List<Double>, etc.

Contravariance (? super T): Allows a generic type to be a supertype of a specific type (T).

**Example:**

List<? super Integer> can hold List<Integer>, List<Number>, List<Object>, etc.

### **Usage:**

Use covariance (? extends T) when you need to read data.  
Use contravariance (? super T) when you need to write data.

---

## 132. Can you pass List<String> to a method which accepts List<Object>?

Answer:

NO , In Java, you **cannot** directly pass a List<String> to a method expecting List<Object> due to **invariance** in generics.

List<String> isn't compatible with List<Object> for type safety.

### Workarounds:

- 1.Use wildcards for read-only operations: List<? extends Object>.
  - 2.Copy the list to a List<Object> if modification is needed.
- 

## 133. What is PECS Principle in Generics?

Answer:

The PECS principle is a classic Java Generics interview question. It is asked to check if you understand bounded wildcards (? extends and ? super) and when to use them in collections.

PECS = Producer Extends, Consumer Super

It tells you when to use ? extends and when to use ? super in Generics.

### ? extends T → Producer

- Use when the collection is a producer of data (you only read from it).
- You can safely read items as type T.
- But you cannot add new elements (except null).

```
List<? extends Number> numbers = new ArrayList<Integer>();  
Number n = numbers.get(0); // Safe to read  
// numbers.add(10); // Compile error
```

### ? super T → Consumer

- Use when the collection is a consumer of data (you write into it).
- You can safely add objects of type T or its subclasses.
- But when reading, you only get Object.

```
List<? super Integer> integers = new ArrayList<Number>();  
integers.add(10); // Safe to add Integer  
integers.add(20); // Safe  
Object obj = integers.get(0); // Returns Object, not Integer
```

---

## 134. What is TypeErasur?

Answer:

Type erasure is the process by which the Java compiler removes all information related to type parameters and type arguments within a generic type declaration. This happens during compilation, not at runtime.

The compiler strips away type parameters from generic types, replacing them with their bounds or Object if unbounded.

```
List<String> list = new ArrayList<>();  
// At runtime, this is essentially:  
List list = new ArrayList();
```

### Key Points:

Compile-Time: Type checking and safety are enforced.

Run-Time: Generic type information is lost; List<String> becomes List.

---

## 135. What is a generic type inference?

Answer:

Generic type inference allows the compiler to automatically determine the type arguments for a generic method or constructor based on the context in which it is used.

### Example:

```
public static <T> List<T> singletonList(T value) {  
    return Collections.singletonList(value);  
}  
List<String> list = singletonList("Hello");
```

The compiler infers that T is String based on the argument passed to the method. In Java 8 and later, the <> diamond operator can be used to simplify

this.

---

**136. Suppose you need to overload a method to handle both a List<Integer> and a List<Double>. Can you overload methods with these types?**

Answer:

In Java, method overloading with generic types is restricted due to type erasure, which removes generic type information at runtime.

Both List<Integer> and List<Double> are erased to List<Object> during compilation, leading to a conflict.

For example:

```
public void method(List<Integer> list)
{
    // Do something with Integer list
}
public void method(List<Double> list) {
    // Do something with Double list
}
```

These methods will cause a compile-time error because after type erasure, both methods would look like:

```
public void method(List<Object> list) {
    // Do something
```

```
}
```

---

### 137. Why can't we create an array of generic types in Java?

Answer:

Creating an array of generic types is prohibited because of type erasure. When the generic type is erased to Object or another bound, the array loses its type safety, leading to potential runtime ClassCastException.

For example, T[] would be treated as Object[], which could allow inserting elements of any type, breaking the type safety of the array.

---

### 138. I have a generic method called merge that merges two collections into one:

```
public static <T> Collection<T>
merge(Collection<T> a, Collection<T> b)
{
    /.../
}
```

If I call this method like below, will it work ?

```
List<Number> numbers = merge(new
ArrayList<Integer>(), new
ArrayList<Double>());
```

Answer:

The code fails because the types Integer and Double are not compatible; the **type inference** cannot determine a single type T that satisfies both Integer and Double.

To fix this, you could change the method signature to use wildcards:

```
public static <T extends Number> Collection<T>
merge(Collection<? extends T> a, Collection<? extends T> b)

{  
    /.../  
}
```

Now the method will accept collections of different subtypes of Number, inferring the common supertype (Number in this case).

---

**139. You are required to write a utility method that accepts an arbitrary number of arguments of any type and returns a list containing those arguments. However, the method should ensure type safety and avoid issues with heap pollution.**  
**Implement this method using generics.**

Answer:

A type-safe, generic utility method that accepts any number of arguments of any type and returns a List containing those arguments, avoiding heap pollution:

```
public class Utils {  
    @SafeVarargs  
    public static <T> List<T> toList(T... elements) {  
        return Arrays.asList(elements);  
    }  
}
```

#### Explanation:

- The method is generic `<T>`, so it works for any type.
- It uses varargs (`T... elements`) to accept an arbitrary number of arguments.
- The `@SafeVarargs` annotation suppresses warnings related to potential heap pollution from varargs of generic types.
- Returns a fixed-size `List<T>` backed by the array (`Arrays.asList`).

#### Example usage:

```
List<String> stringList = Utils.toList("one", "two", "three");  
List<Integer> intList = Utils.toList(1, 2, 3, 4);
```

This implementation is type-safe and avoids heap pollution issues.

# **Chapter 10: Java Memory management**

These questions help assess a candidate's ability to read and interpret GC logs, understand the implications of different GC strategies, and make informed decisions to optimize garbage collection in Java applications. This section is important when you are going for an interview of investment banking domain.

---

## **140. How Are Strings Represented in Memory?**

Answer:

In Java, strings are represented as objects of the String class. Each string is stored in the heap memory, and internally, a string is backed by a character array (char[]), which holds the actual string data. Java strings are immutable, meaning that once created, their content cannot be changed.

This immutability is achieved using a special memory area called the string pool. The string pool allows for memory optimization by storing string literals only once. When a new string is created, the JVM checks if the string already exists in the pool. If it does, it reuses the reference to the existing string. If not, it creates a new string in the pool. This approach not only saves memory but also helps with performance, as it avoids unnecessary string duplications.

Java 9+ Optimization:

Java 9 introduced a byte[] + encoding flag for optimization (Compact Strings).

Saves memory for strings that only use Latin-1 characters.

### Summary:

Heap Memory: Strings are stored in the heap.

String Pool: A memory optimization area where identical string literals are stored.

Immutability: Strings cannot be changed after creation.

---

## **141. Is it possible to resurrect an Object that became eligible for garbage collection?**

Answer:

Yes, it is possible to resurrect an object that has become eligible for garbage collection, but only once. This can be achieved using the finalize() method. When the garbage collector finds an object with no more references, it marks the object for garbage collection and calls its finalize() method before collecting it.

If, during the execution of the finalize() method, the object assigns a reference to itself (e.g., to a static variable), it becomes reachable again and is “resurrected.” However, once an object has been finalized, it will not be finalized again if it later becomes unreachable, making further resurrection impossible.

It’s important to note that using finalize() for resurrection is highly discouraged in modern Java development. The finalize() method introduces unpredictable behavior and performance issues. Additionally, it has been deprecated

since Java 9 in favor of alternatives like try-with-resources or explicit resource management.

### Summary:

**Resurrection:** An object can be resurrected once during the finalize() method.

**Discouraged:** The use of finalize() is considered bad practice due to its unpredictability.

---

## 142. What are the default garbage collectors in different Java versions?

Answer:

Java uses several garbage collectors, with defaults that have evolved across versions:

**Java 7 and earlier:** The default garbage collector was the Parallel GC (also known as the Throughput Collector). It aims to maximize application throughput by utilizing multiple CPU threads.

**Java 8:** The Parallel GC remained the default.

**Java 9 to Java 17:** The G1 GC (Garbage-First Garbage Collector) became the default. G1 is designed to optimize both pause times and throughput, making it suitable for applications that require consistent performance.

**Java 18+:** The default garbage collector is still G1 GC, but alternative collectors like ZGC (designed for low-latency) and Shenandoah GC are available for specific use cases that require extremely low pause times.

### Summary :

- Java 7 and earlier: Parallel GC (Throughput Collector).
- Java 8: Parallel GC.

- Java 9 - Java 17: G1 GC (Garbage-First).
  - Java 18+: G1 GC (default), with ZGC and Shenandoah as alternatives.
- 

## 143. What are Strong, Weak, Soft and Phantom References and their Role in Garbage Collection?

Answer:

### Strong References:

A strong reference is the default type of reference in Java. Any object with a strong reference cannot be garbage collected as long as the reference exists. It is the most common reference type, and most objects are strongly referenced.

### Weak References:

A weak reference does not prevent an object from being garbage collected. If the only references to an object are weak, it becomes eligible for garbage collection, even if memory is not low. Weak references are often used in scenarios like implementing memory-sensitive caches.

### Soft References:

Soft references are similar to weak references but with one key difference: objects with only soft references are not immediately garbage collected when they become unreachable. They are collected only when the JVM is running low on memory. This makes them useful for caching purposes where you want to retain objects as long as there is enough memory.

### Phantom References:

Phantom references are the weakest type of reference and are used to determine when an object has been definitively removed from memory. Unlike soft and weak references, a phantom-referenced object is already finalized and cannot be resurrected. Phantom references are used in conjunction with reference queues to clean up resources after an object has been collected.

### Summary:

**Strong References:** Prevent garbage collection.

**Weak References:** Allow garbage collection when no strong references exist.

**Soft References:** Collected only when memory is low.

**Phantom References:** Used for post-garbage collection cleanup.

---

## **144. What happens in the memory when we use `new` Keyword?**

Answer:

When you use the new keyword in Java, the following steps happen in memory:

**1. Heap Allocation:** A new object is created in the heap memory. The heap is where all Java objects are stored, and it's managed by the JVM.

**2. Reference in Stack:** A reference to the object is created and stored in the stack memory. The stack holds the references (addresses) to objects in the heap.

**3. Constructor Execution:** The constructor of the object is invoked to initialize the object. The constructor assigns values to instance variables and performs any required

setup.

**4. Reference to the Object:** After the constructor completes, the reference points to the memory location of the newly created object, allowing you to interact with the object via that reference.

Summary:

**Heap Memory:** The object is created in the heap.

**Stack Memory:** A reference to the object is stored in the stack.

**Constructor:** Initializes the object when created.

---

## 145. What are the different types of Garbage Collectors in Java?

Answer:

Java provides several types of garbage collectors, each optimized for different use cases:

- Serial Garbage Collector: A simple, single-threaded garbage collector that pauses all application threads during garbage collection. It is suitable for single-threaded environments or small applications.

- Parallel Garbage Collector (Throughput Collector): Uses multiple threads to speed up garbage collection. It is designed for high-throughput applications that can tolerate longer pauses for garbage collection.

- CMS (Concurrent Mark-Sweep) Garbage Collector: Reduces garbage collection pauses by doing most of the work concurrently with application threads. It is suitable

for applications requiring low latency.

- G1 (Garbage First) Garbage Collector: Aims to provide both high throughput and low latency by dividing the heap into regions and collecting the ones with the most garbage first. It is designed for applications running on multi-core processors with large heaps.

- Z Garbage Collector (ZGC): A low-latency garbage collector capable of handling large heaps with minimal pause times. It is designed for ultra-low latency applications.

- Shenandoah Garbage Collector: Similar to ZGC, Shenandoah focuses on ultra-low pause times, making it suitable for large heap applications with strict latency requirements.

---

## **146. What performance optimizations have you done in your Java project?**

Answer:

This is a very commonly asked interview question and it's better to have an answer prepared for this . An ideal answer should look like below , you might have to relate this to your project somehow:

### **1. Profiled and Analyzed Performance:**

- Used tools like JProfiler and VisualVM to identify bottlenecks, memory leaks, and inefficient code paths.

### **2. Optimized Database Queries:**

- Improved SQL query performance by adding indexes,

optimizing queries, and using efficient data access patterns with JPA/Hibernate.

**3. Implemented Caching:**

- Used caching mechanisms (e.g., Ehcache, Redis) to reduce database load and speed up frequent data retrieval.

**4. Tuned Garbage Collection:**

- Configured JVM garbage collection settings (e.g., G1 GC, ZGC) to reduce pause times and optimize heap usage.

**5. Improved Concurrency:**

- Used concurrent collections and optimized thread usage to improve multi-threaded performance and reduce contention.

**6. Enhanced Code Efficiency:**

- Refactored code to reduce complexity, remove redundant operations, and use efficient algorithms and data structures.

**7. Load Testing and Scaling:**

- Conducted load testing to identify performance issues under high traffic and scaled the application horizontally (e.g., load balancers, microservices).

**8. Asynchronous Processing:**

- Implemented asynchronous processing for tasks like I/O operations to improve responsiveness and throughput.

**9. Optimized Network Communication:**

- Reduced network latency and improved performance by using efficient data serialization formats (e.g., Protobuf) and minimizing network calls.

**10. Used Profiling and Monitoring Tools:**

- Leveraged monitoring tools (e.g., Prometheus, Grafana) to continuously monitor performance and quickly identify and address issues.

## **147. What coding standards do you follow as a Java developer?**

Answer:

Another very commonly asked interview question , try to pick points from your project and relate it to points below:

1. Consistent Style: Follow a unified style guide and use tools like Checkstyle.
  2. Design Patterns: Apply appropriate patterns (e.g., Singleton, Strategy).
  3. Modular Architecture: Organize code into modular, SOLID-compliant components.
  4. Exception Handling: Use specific exceptions and meaningful messages.
  5. Resource Management: Use try-with-resources to manage resources efficiently.
  6. Testing: Write comprehensive unit and integration tests.
  7. Performance: Profile and optimize performance, use caching.
  8. Documentation: Provide clear Javadoc comments and inline explanations.
  9. Code Reviews: Conduct thorough reviews focusing on quality and standards.
  10. CI/CD: Integrate and deploy code frequently using automated pipelines.
- 

## **148. What are different areas in Java Memory?**

Answer:

Below are different memory areas in Java:

### Heap Area

- Stores: Objects and class instances
- Managed by: Garbage Collector
- Subdivided into:
  - Young Generation (Eden + Survivor spaces)
  - Old Generation (Tenured)
- Most memory-related issues (e.g., OutOfMemoryError) occur here.

### Stack Area

- Stores: Method call frames, local variables, references
- One stack per thread (thread-private)
- Memory is allocated/deallocated in LIFO order
- Error: StackOverflowError if recursion is too deep

### Method Area (or Metaspace in Java 8+)

- Stores: Class metadata, static variables, method info
- In Java 8+, moved to native memory (Metaspace)
- Grows dynamically (up to system memory)

### Program Counter (PC) Register

- Stores: Address of the current JVM instruction for each thread
- One per thread (thread-private)
- Used to resume execution correctly after a method call or jump

### Native Method Stack

- Supports: Native (non-Java) method execution (e.g., JNI)
- Uses native libraries (C/C++)
- Separate from Java stacks

Memory Area	Purpose
Heap	Objects, class instances
Stack	Method calls, local variables
Method Area / Metaspace	Class metadata, static methods
PC Register	Instruction pointer
Native Method Stack	Native (C/C++) method calls

---

## 149. [FOLLOW-UP] How is memory allocated in these areas?

Answer:

- Heap Memory: Managed by the JVM and allocated dynamically during runtime for objects.
  - Stack Memory: Allocated for each thread when methods are invoked, with frames containing local variables and method references.
  - Method Area: Pre-allocated during JVM initialization and grows as more classes are loaded.
- 

## 150. What are Memory Leaks in Java and how to prevent them?

Answer:

A memory leak in Java occurs when objects are no longer needed by the application but are still referenced,

preventing the garbage collector from reclaiming their memory. Over time, this can lead to OutOfMemoryErrors and degrade application performance.

**To prevent memory leaks:**

- Ensure that objects are dereferenced (e.g., setting them to null) when they are no longer needed.
  - Use appropriate collection classes (e.g., WeakHashMap for caches) that allow objects to be garbage collected when they are no longer in use.
  - Be cautious with listeners, callbacks, and static fields that may inadvertently hold references to objects, preventing their collection.
- 

**151. What is Metaspace in Java and how does it differ from PermGen?**

Answer:

Metaspace is the memory area where class metadata is stored in Java 8 and later versions, replacing the Permanent Generation (PermGen).

Unlike PermGen, which had a fixed maximum size, Metaspace dynamically resizes based on application needs, utilizing native memory.

This change eliminates many of the memory management issues associated with PermGen, such as OutOfMemoryError: PermGen space, making class metadata management more efficient.

Feature	Metaspace (Java 8+)	PermGen (Java 7-)
Location	Native memory	Java heap
Size	Dynamic (no default limit)	Fixed ( <code>-XX:MaxPermSize</code> )
OOM Risk	Low (native memory)	High (fixed size)
GC	Efficient class unloading	Limited (full GC only)
Error	<code>OutOfMemoryError: Metaspace</code>	<code>OutOfMemoryError: PermGen space</code>

Metaspace is a significant improvement over PermGen, offering dynamic memory allocation, better garbage collection, and reduced risk of memory errors. It is designed to handle the needs of modern Java applications with dynamic class loading, making PermGen obsolete as of Java 8.

---

## 152. How can we monitor Garbage Collection activities in Java?

Answer:

To monitor garbage collection activities succinctly:

1. Using JVM Options (for logs)
2. JVisualVM (GUI tool)
  - Bundled with JDK
  - Monitor memory, GC activity, heap usage in real time
3. JConsole
  - Another JDK GUI tool
  - Shows memory pools and GC statistics
4. Garbage Collection Logs + GCViewer
  - GC logs can be analyzed using GCViewer

5. FR (Java Flight Recorder) – Java 11+  
Powerful profiling + GC tracking  
Enable with:  
-XX:StartFlightRecording=filename=recording.jfr
6. Third-party Tools  
Prometheus + Grafana  
New Relic, AppDynamics, Dynatrace

### Summary:

Use JVM flags for logging, JVisualVM/JConsole for live monitoring, or advanced tools like JFR or Prometheus for production environments.

---

## **153. Is Memory Size of Heap fixed? How is Memory Allocated in Heap?**

Answer:

The heap size in Java is not fixed and can be configured via JVM arguments like ` -Xms` (initial heap size) and ` -Xmx` (maximum heap size).

- Allocation: Memory allocation is managed by the JVM based on the program's need.

- New objects → Eden space
- Surviving objects → Moved to Survivor → Eventually to Old Gen

- Who Allocates: The operating system allocates memory to the JVM, and the JVM allocates it to objects during runtime.

### Heap is Divided Into:

- Young Generation (Eden + Survivor)
- Old Generation (Tenured)

### Garbage Collection manages memory:

- Minor GC: Cleans Young Gen
  - Major/Full GC: Cleans Old Gen
- 

## **154. Examine the following garbage collection output and answer these questions:**

1. What type of garbage collector is being used?
2. How much memory was freed in the young generation?
3. What was the total heap usage change after the GC?
4. How long did the garbage collection pause take?

Garbage Collection Output:

```
[GC  
[ParNew: 1800K->90K(1800K), 0.0552314 secs]  
22320K->20610K(65536K), 0.0553213 secs]  
[Times: user=0.04 sys=0.01, real=0.06 secs]
```

Answer:

1. Type of Garbage Collector:

The garbage collector being used is ParNew, which is typically used for the young generation in the parallel collector, often part of the CMS (Concurrent Mark Sweep) collector setup in Java.

## 2.Memory Freed in Young Generation:

The young generation (ParNew) went from 1800K to 90K.  
Therefore, the memory freed is:

$$1800\text{K} - 90\text{K} = 1710\text{K}$$

## 3.Total Heap Usage Change:

The total heap usage before GC was 22320K, and after GC it was 20610K. The change in heap usage is:

$$22320\text{K} - 20610\text{K} = 1710\text{K}$$

This matches the memory freed in the young generation, suggesting no significant change in the old generation during this GC cycle, or the change there was minimal or not shown in this snippet.

## 4.Garbage Collection Pause Time:

The real time taken for the garbage collection pause is given as:

$$\text{real}=0.06 \text{ secs}$$

This includes both the ParNew collection and any minor overheads included in the timing but primarily reflects the stop-the-world pause for the young generation collection.

---

## 155. Examine the the following output with the G1 garbage collector (-XX:+UseG1GC):

### Garbage Collection Output:

[GC pause (G1 Evacuation Pause) (mixed)  
2048M->1024M(4096M), 0.1200000 secs]

**[GC pause (G1 Evacuation Pause) (mixed)  
1024M->512M(4096M), 0.1100000 secs]**

## **What do the GC pauses signify, and how can you reduce the duration of these pauses?**

Answer:

- These pauses are due to the G1 collector's evacuation process, where live objects are moved to a new region, and the old region is reclaimed.
  - The "mixed" indicates that both young and old regions are being collected.
  - To reduce the pause duration, you can tune the -XX:MaxGCPauseMillis parameter, reduce the heap occupancy trigger for mixed GCs, or adjust the -XX:G1HeapRegionSize.
- 
-

# Chapter 11: Output based

**156. What will be the output of the following code snippet?**

```
String s1 = new String("test") + new  
String("test");  
String s2 = "testtest";  
System.out.println(s1 == s2);  
System.out.println(s1.equals(s2));
```

Answer:

**s1 == s2 prints false.** s1 creates a new String object, while s2 refers to a string literal from the string pool.

**s1.equals(s2) prints true.** Both strings have the same value.

---

**157. What will be the output of the following code snippet?**

```
class Test {  
    void method(String s) {  
        System.out.println("String");  
    }  
  
    void method(String... s) {  
        System.out.println("Varargs");  
    }  
}
```

```
 }

public static void main(String[] args) {
    Test t = new Test();
    t.method("Hello");
}
}
```

Answer:

**String.**

The method with single String argument is preferred over the method with String... s (varargs).

---

### 158. What will be the output of the following code snippet?

```
class Test {
    void method(int... nums) {
        System.out.println("Int varargs");
    }

    void method(Integer... nums) {
        System.out.println("Integer varargs");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.method(1, 2, 3);
    }
}
```

Answer:

error: reference to method is ambiguous

t.method(1, 2, 3);

  ^

Method 1 is a direct match via int...

Method 2 is also acceptable because of autoboxing to Integer...

Since **both are valid** and **neither is more specific**, the compiler throws:

### Error: reference to method is ambiguous

---

### 159. What will be the output of the following code snippet?

```
class Test {  
    void method(double d) {  
        System.out.println("Double");  
    }  
  
    void method(int i) {  
        System.out.println("Int");  
    }  
  
    public static void main(String[] args) {  
        Test t = new Test();  
        t.method(5);  
    }  
}
```

Answer:

**Int.**

The method with int is chosen because it is more specific than double.

---

**160. What will be the output of the following code snippet?**

```
class Test {  
    void method(Object o) {  
        System.out.println("Object");  
    }  
  
    void method(String s) {  
        System.out.println("String");  
    }  
  
    public static void main(String[] args) {  
        Test t = new Test();  
        t.method(null);  
    }  
}
```

Answer:

**String.**

The method with String is preferred over the method with

Object because String is more specific.

---

### **161. What will be the output of the following code snippet?**

```
class Test {  
    void method(int... nums) {  
        System.out.println("Int varargs");  
    }  
  
    void method(int num) {  
        System.out.println("Int");  
    }  
  
    public static void main(String[] args) {  
        Test t = new Test();  
        t.method(1);  
    }  
}
```

Answer:

**Int.**

The method with a single int is preferred over the varargs method when there is an exact match.

---

### **162. What will be the output of the following code snippet?**

```
try {
    System.out.println("Inside try block");
    System.exit(0);
} catch (Exception e) {
    System.out.println("Exception caught: " + e);
} finally {
    System.out.println("In finally block");}
}
```

Answer:

### **Inside try block**

After printing "Inside try block", the program will terminate due to `System.exit(0)`. Therefore, neither the `catch` nor the `finally` block will execute.

- `System.exit(0)` is a method that terminates the JVM immediately, which means that once it is called, no further code is executed, including the `finally` block.
- The `finally` block is normally guaranteed to execute after `try` and `catch`, but `System.exit(0)` prevents this from happening.

---

### **163. What will be the output of the following code snippet?**

```
class A {
    static {
        System.out.println("Class A loaded");
```

```
        }
    }

class B extends A {
    static {
        System.out.println("Class B loaded");
    }
}

class C extends B {
    static {
        System.out.println("Class C loaded");
    }
}

public class TestClass {
    public static void main(String[] args) {
        B b;
        System.out.println("Inside main");
        b = new B();
    }
}
```

Answer:

The output will be:

**Inside main**  
**Class A loaded**  
**Class B loaded**

Explanation:

The reference B b; does not trigger any class loading, so no static block is executed.

The message Inside main is printed.

When b = new B(); is executed, class B is loaded, which triggers the static blocks of its parent class A first and then B. Since class C is not used, its static block is never executed.

# Chapter 12: JAVA 8

## 164. What is a Functional Interface?

Answer:

A functional interface is an interface in Java which:

- Has exactly one abstract method.
- Can be annotated with @FunctionalInterface.
- May include default and static methods.
- Enables use of lambda expressions and method references.

Some important Functional Interfaces

Interface	Method	Description
Runnable	<code>run()</code>	No result, no exceptions.
Callable<V>	<code>call()</code>	Returns result, can throw exceptions.
Supplier<T>	<code>get()</code>	Provides a result.
Consumer<T>	<code>accept(T)</code>	Consumes an input.
BiConsumer<T, U>	<code>accept(T, U)</code>	Consumes two inputs.
Function<T, R>	<code>apply(T)</code>	Maps T to R.
BiFunction<T, U, R>	<code>apply(T, U)</code>	Maps two inputs to an output.
Predicate<T>	<code>test(T)</code>	Tests an input, returns boolean.
BiPredicate<T, U>	<code>test(T, U)</code>	Tests two inputs, returns boolean.

## **165. What is the difference between Lambda vs. Anonymous Classes?**

Answer:

Lambda Expressions: More concise and readable.

```
Runnable r = () -> System.out.println("Hello");Anonymous  
Classes: More verbose.Runnable r = new Runnable() {  
    public void run() {  
        System.out.println("Hello");  
    }  
};
```

Benefits of Lambdas: Less boilerplate code, clearer intent, and better support for functional programming.

Method References Usage: Provides a shorthand for lambda expressions where the lambda just calls a method.

```
Function<String, Integer> parseInt = s -> Integer.parseInt(s);  
// Lambda
```

```
Function<String, Integer> parseIntRef = Integer::parseInt; //  
Method Reference
```

Comparison: Method references can improve readability, but both have similar performance characteristics

---

## **166. Are streams slower or faster than conventional for loop?**

Answer:

Yes, Java Streams can be slower than conventional for loops in certain scenarios due to the overhead involved in creating streams, invoking functional interfaces, and managing intermediate operations. This overhead can be noticeable, especially with small datasets or simple operations.

However, streams offer advantages in readability and maintainability, and they can outperform for loops when dealing with large datasets or when parallel processing is utilized (`parallelStream()`), which allows the workload to be spread across multiple CPU cores.

#### In summary:

- For loops are typically faster for small, simple operations.
- Streams may be slower due to overhead but can be more efficient with large datasets, particularly when parallelized.

The choice between streams and for loops should balance performance needs with code clarity and maintainability.

---

### **167. [FOLLOW UP] Then what is the advantage of using Stream API?**

Answer:

1. Readable: Streams offer a more readable and concise syntax.
2. Chaining: Allows method chaining for complex operations.
3. Lazy Evaluation: Only processes elements when

necessary, optimizing performance.

4. Parallel Processing: Easily parallelize with parallelStream() for faster execution.
  5. Functional Style: Leverages functional programming with lambda expressions.
- 

## 168. Can you name few predefined Functional Interfaces in Java?

Answer:

1. Function<T, R>: Takes one argument (T) and returns a result (R).
  - Method: R apply(T t)
2. BiFunction<T, U, R>: Takes two arguments (T and U) and returns a result (R).
  - Method: R apply(T t, U u)
3. Predicate<T>: Takes one argument (T) and returns a boolean.
  - Method: boolean test(T t)
4. BiPredicate<T, U>: Takes two arguments (T and U) and returns a boolean.
  - Method: boolean test(T t, U u)
5. Consumer<T>: Takes one argument (T) and returns no result.
  - Method: void accept(T t)
6. BiConsumer<T, U>: Takes two arguments (T and U) and returns no result.

- Method: void accept(T t, U u)
7. Supplier<T>: Produces a result of type T with no input.  
- Method: T get()
8. UnaryOperator<T>: A Function that takes and returns the same type T.  
- Method: T apply(T t)
9. BinaryOperator<T>: A BiFunction that takes two arguments of the same type T and returns the same type T.  
- Method: T apply(T t1, T t2)
10. Comparator<T>: Compares two objects of type T.  
- Method: int compare(T o1, T o2)
11. Runnable: Represents a task that takes no arguments and returns no result.  
- Method: void run()
- 

## 169. You have the following code snippet:

```
Stream<String> stream =  
    Stream.of("a", "b", "c").filter(s -> s.startsWith("a"));  
    stream.forEach(System.out::println);  
    stream.forEach(System.out::println); // What happens here?
```

Answer:

Streams in Java can only be consumed once. Attempting to reuse the stream will result in an **IllegalStateException**.

The second call to forEach will throw an IllegalStateException because the stream has already been

consumed.

---

## 170. You have the following code snippet:

```
List<String> data = Arrays.asList("apple", "banana",  
"cherry");  
Stream<String> stream =  
data.stream().map(String::toUpperCase);
```

**What will happen if no terminal operation is invoked on this stream?**

Answer:

Streams are lazily evaluated, so intermediate operations like map don't execute until a terminal operation is invoked.

The map operation will not be executed, and no transformations will occur since no terminal operation is present.

---

## 171. Explain the difference between Stream API map and flatMap?

Answer:

map: Transforms each element in the stream.

The map() function is used when you want to transform each element of a stream into another value, one-to-one.

```
List<String> strings = Arrays.asList("a", "b");  
List<Integer> lengths =
```

```
strings.stream().map(String::length).collect(Collectors.toList());
```

flatMap: Flattens nested structures.

The flatMap() function is used when each element can be transformed into multiple elements (like a list), and you want to flatten all the results into a single stream.

```
List<List<String>> nested =  
    Arrays.asList(Arrays.asList("a", "b"), Arrays.asList("c",  
        "d"));  
List<String> flat =  
    nested.stream().flatMap(Collection::stream).collect(Coll  
        ectors.toList());
```

---

## 172. Explain the difference between peek() and map(). In what scenarios should peek() be used with caution?

Answer:

map() is a transformation operation that produces a new stream by applying a function to each element.

peek(), on the other hand, is mainly used for debugging or performing side effects without modifying the stream.

It should be used with caution because side effects may lead to non-deterministic behavior, especially in parallel streams.

Since peek() doesn't modify the data, it's not ideal for business logic but can be useful for logging intermediate

---

steps.

---

### **173. Can an interface with multiple default methods still be a functional interface?**

Answer:

Yes, an interface with multiple default methods can still be a functional interface as long as it has only one abstract method, since default methods do not count towards the abstract method count.

---

### **174. Describe a situation where you might use a custom functional interface instead of using built-in ones like Predicate, Function, or Consumer.**

Answer:

You might use a custom functional interface when you need to define behavior that doesn't fit the standard patterns of Predicate, Function, or Consumer.

For example, if you need a method that accepts three parameters and returns a result (like `TriFunction<A, B, C, R>`), there is no built-in Java functional interface for this specific case, so you would define a custom interface.

---

### **175. [FOLLOW-UP] How would you implement a custom FunctionalInterface**

## **that takes three arguments and returns a result?**

Answer:

You can create a custom functional interface for scenarios where you need three arguments, as Java doesn't provide a built-in interface for that.

Example:

```
@FunctionalInterface  
public interface TriFunction<T, U, V, R> {  
    R apply(T t, U u, V v);  
}
```

Usage

```
TriFunction<Integer, Integer, Integer, Integer> sum = (a, b,  
c) -> a + b + c;  
int result = sum.apply(1, 2, 3); // result = 6
```

This differs from built-in interfaces like BiFunction, which only take two arguments, so a custom interface is needed for more complex cases.

---

## **176. How BiFunction, BiConsumer, and BiPredicate are different from their single-parameter counterparts - Function, Consumer, and Predicate, and give a scenario where you would use them ?**

Answer:

The "Bi" versions of functional interfaces accept two parameters:

**BiFunction<T, U, R>**: Takes two parameters (T and U) and returns a result (R). Useful for cases like combining two objects into a third, such as adding two numbers or merging data.

**BiConsumer<T, U>**: Takes two parameters and performs an action but returns no result. Useful when you need to perform an operation using two inputs, like logging two related pieces of information.

**BiPredicate<T, U>**: Takes two parameters and returns a boolean. Useful for situations like comparing two objects or checking relationships between them, such as checking if two strings are anagrams.

---

## **177. What is the difference between UnaryOperator<T> and Function<T, R>? When would you prefer using UnaryOperator over Function?**

Answer:

**UnaryOperator<T>** is a specialization of **Function<T, T>** where both the input and output are of the same type.

You would prefer using **UnaryOperator** when you have a scenario where the input and output types are the same, such as incrementing an integer or converting a string to uppercase.

It makes the code more readable and semantically clearer.

Feature	Function<T, R>	UnaryOperator<T>
Type Parameters	T (Input), R (Result, can differ from T)	T (Input and Result are of the same type)
Method Signature	R apply(T t)	T apply(T t)
Purpose	Maps one type to another type.	Applies an operation on a type to produce the same type.
Use Case	When transforming input to a different type of output.	When you want to operate on and return the same type.
Example	Converting String to Integer (Function<String, Integer> )	Incrementing an Integer (UnaryOperator<Integer> )
Specialization	General purpose function interface.	Specialized Function where input and output are identical.
When to Prefer	When the output type differs from the input type or when you need more flexibility in type transformation.	When you're dealing with operations where the result is the same type as the input, enhancing readability and ensuring type consistency.

## 178. In what scenarios might you prefer using method references (:) over lambda expressions in Java 8?

Answer:

Method references are often preferred when they improve code readability, such as when passing an existing method

instead of writing a new lambda expression.

For instance, `String::toUpperCase` is more readable than `s -> s.toUpperCase()`.

Performance-wise, there's typically no significant difference between method references and lambda expressions, as both are translated to similar bytecode under the hood, though this depends on the JVM optimizations.

---

## 179. What is the difference between a lambda expression and an anonymous inner class. Can lambda expressions access non-final local variables?

Answer:

Aspect	Lambda Expression	Anonymous Inner Class
Syntax	Concise: <code>(args) -&gt; expression</code>	Verbose: <code>new Interface() { method() { ... } }</code>
Purpose	For functional interfaces (single method).	For any interface or class, multiple methods.
Type	Method implementation, not a class.	New unnamed class instance.
Scope	Captures <code>this</code> as enclosing class.	Has own <code>this</code> (refers to the anonymous class).
Flexibility	Limited to single method implementation.	Can implement multiple methods or extend classes.
Performance	Lighter, uses <code>invokedynamic</code> (more optimized).	Heavier, creates a new class instance.

Both lambda expressions and anonymous inner classes can only access final or effectively final local variables.

- X** They cannot modify local variables from the enclosing method.
-

---

## **180. What are higher-order functions in Java**

### **8. How can you implement a higher-order function using functional interfaces?**

Answer:

A higher-order function is a function that either takes one or more functions as arguments or returns a function.

In Java, you can use functional interfaces to implement higher-order functions.

For example, a function that returns a Function<String, String> that appends a suffix:

```
public static Function<String, String> addSuffix(String suffix) {  
    return s -> s + suffix;  
}
```

#### Usage

```
Function<String, String> addExclamation = addSuffix("!");  
String result = addExclamation.apply("Hello"); // Output:  
"Hello!"
```

---

## **181. What is the use of the BinaryOperator<T> interface. When would you choose BinaryOperator over BiFunction?**

Answer:

BinaryOperator<T> is a specialization of BiFunction<T, T,

`T>`, where both arguments and the return type are of the same type.

You would choose `BinaryOperator` when performing operations where both inputs and outputs are the same, like adding two numbers, concatenating strings, or merging collections.

Example:

```
BinaryOperator<Integer> sum = (a, b) -> a + b;  
int result = sum.apply(5, 10); // result = 15
```

---

**182. Explain the difference between `Optional.map` and `Optional.flatMap` with an example OR Why do we need `flatMap` when dealing with `Optional`?**

**Answer:**

`Optional.map` wraps the result inside another `Optional` if the mapping function itself returns an `Optional`, whereas `flatMap` prevents nested `Optional<Optional<T>>`.

```
Optional<String> name = Optional.of("John");
```

```
Optional<Optional<String>> mapResult =  
name.map(Optional::of); // Nested Optional
```

```
Optional<String> flatMapResult =  
name.flatMap(Optional::of); // Flattened Optional
```

---

---

**183. Explain the concept of short-circuiting operations in streams. What would be the result of the following stream pipeline, and why?**

```
Stream.of("one", "two", "three", "four")
    .map(String::toUpperCase)
    .filter(s -> s.length() > 3) .findFirst();
```

Answer:

**Optional[THREE]**

The stream pipeline will return an Optional containing "THREE".

findFirst is a short-circuiting operation that stops processing once the first matching element is found, so the stream will not continue to process after "four" is encountered.

Short-circuiting in Java Streams refers to the ability of some intermediate or terminal operations to stop processing as soon as a certain condition is met, thereby not processing all elements of the stream. This is particularly useful for optimizing performance when working with large or infinite streams.

Some Operations that Short-Circuit:

Method	Description
<b>anyMatch</b>	Stops if predicate matches any element.
<b>allMatch</b>	Stops if any element fails to match predicate.
<b>noneMatch</b>	Stops at first match of predicate.
<b>findFirst</b>	Returns first element or <code>Optional.empty()</code> .
<b>findAny</b>	Returns any element or <code>Optional.empty()</code> .
<b>limit(n)</b>	Stops after <code>n</code> elements.

---

## 184. What is a Spliterator, and how does it relate to the internal working of streams?

Answer:

A Spliterator is an interface introduced in Java 8 that provides a way to traverse and partition elements of a source for processing, particularly in the context of streams.

### Role in Streams:

Spliterator is used internally by streams to split the source into smaller parts, which can then be processed in parallel.

This is crucial for parallel streams where tasks need to be distributed across multiple threads.

```
Spliterator<String> spliterator = listspliterator();
```

---

**185. You've refactored a large codebase to use Java Streams for better readability. However, after deploying the changes, you notice a significant performance degradation. What could be the cause, and how would you address this?**

Answer:

The performance degradation could be due to several factors:

**Boxing/Unboxing:** If streams are used with primitive types but the code relies on boxing/unboxing, it can cause overhead.

**Excessive Object Creation:** Streams can create many intermediate objects if not optimized, especially with operations like filter or map.

**Lack of Parallelism:** Streams can be run in parallel, but if used incorrectly or with an unsuitable data structure, parallel streams can degrade performance instead of improving it.

To address this, you might revert to more manual loop constructs for performance-critical sections, use primitive streams (IntStream, LongStream), or ensure that parallel streams are appropriately applied

---

# Chapter 13: Java New Versions features

Over the years, Java has evolved significantly, with each version introducing features aimed at simplifying development, enhancing performance, and improving security. For interviewers, candidate's understanding of these new features isn't just about keeping up-to-date; it's about assessing their adaptability, learning curve, and depth of knowledge in Java.

**Version**	**Feature**
**11**	HTTP Client API, String Methods, ZGC, `var` Lambdas
**12**	EE Module Removal, Nashorn Deprecated
**13**	Switch Expressions (Preview)
**14**	Text Blocks (Preview)
**15**	Records (Preview), NPE Enhancements
**16**	Sealed Classes (Preview)
**17**	Pattern Matching for `instanceof`
**18**	Sealed Classes (Final), FFM API (Incubator)
**19**	Pattern Matching for switch (Preview), PRNGs
**20**	Simple Web Server, UTF-8 Default Charset
**21**	Virtual Threads (Preview), Structured Concurrency
**22**	Scoped Values (Incubator)
**23**	Virtual Threads (Final), Sequenced Collections
	String Templates (Preview), GC Enhancements
	Value Objects (Preview)
	FFM API (Final), GC Enhancements

---

## 186. What is the Java Module System introduced in Java 9, and its usage?

Answer:

The Java Module System (Project Jigsaw) allows developers to break down large applications into smaller, manageable modules. Each module can explicitly state which other modules it requires and which packages it exports.

This improves encapsulation, helps in reducing the application footprint, and prevents classpath issues by resolving dependencies at compile time.

---

## 187. What is a record in Java, and its usage?

**Answer:**

A record is a special kind of class introduced in Java 14, designed to model immutable data.

It automatically generates boilerplate code such as constructors, equals(), hashCode(), and toString() methods based on the fields declared in the record.

**Features:**

- Immutable by Default.
- Auto-generates constructor, toString(), equals(), hashCode().

**Syntax:**

```
record Person(String name, int age) { }
```

**Purpose:**

- Ideal for DTOs or simple data structures.

**Benefits:**

- Less code to write.

- Clear data-centric classes.
- Supports functional programming paradigms.

### Example:

record Point(double x, double y) creates an immutable class with x and y as its components.

```
public record Point(double x, double y) {}

public class Main {
    public static void main(String[] args) {
        Point p = new Point(1.0, 2.0);
        System.out.println(p.x() + ", " + p.y());
    }
}
```

---

## 188. What is a sealed class, introduced in Java 15 and it's usage?

Answer:

Sealed classes restrict which classes can extend or implement them.

This is done by explicitly specifying permitted subclasses using the permits keyword.

### Purpose:

- Control inheritance and implementation.
- Ensure all subclasses are known, useful for exhaustive pattern matching.

### Example:

```

public sealed class Shape permits Circle, Rectangle {
    public abstract double area();
}

final class Circle extends Shape {
    double radius;
    Circle(double r) { this.radius = r; }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

final class Rectangle extends Shape {
    double width, height;
    Rectangle(double w, double h) { this.width = w; this.height = h; }

    @Override
    public double area() {
        return width * height;
    }
}

```

### Keywords:

- sealed: Used to declare the class or interface.
- permits: Lists permitted subclasses or implementors.
- final: Classes that can't be further extended.
- non-sealed: Classes that can still be extended.

### Benefits:

- Enhances type safety.
- Supports exhaustive pattern matching.
- Improves modularity and code design.

---

## **189. What is a hidden class, introduced in Java 15 and it's usage?**

**Answer:**

Hidden classes are classes that are not discoverable through normal reflection and are intended to be used by frameworks that dynamically generate classes at runtime.

**Purpose:**

- Enhance security by restricting class access.
- Improve performance in dynamic class generation scenarios.

**Features:**

- Not visible through typical class loading or reflection.
- Often used internally by JVM for lambda expressions or similar features.

**Benefits:**

- Better encapsulation.
- Improved memory management.
- Increased security.

**Usage:**

- Typically generated by frameworks or JVM for specific, temporary needs.
-

# Chapter 14: Java Multithreading

- 190. You have threads T1, T2, and T3, how will you ensure that thread T2 run after T1 and thread T3 run after T2?**

Answer:

For ensuring the sequence of execution T1, T2, and T3 in Java we can use the join() method of the Thread class. This starts each thread sequentially and waits for each thread to finish before starting the next one, guaranteeing the desired sequence.

```
public class ThreadSequence {  
    public static void main(String[] args) {  
        Thread t1 = new Thread(() -> System.out.println("T1 is running"));  
        Thread t2 = new Thread(() -> System.out.println("T2 is running"));  
        Thread t3 = new Thread(() -> System.out.println("T3 is running"));  
        try {  
            t1.start(); t1.join();  
            t2.start(); t2.join();  
            t3.start(); t3.join();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

## **191. Can we start a thread twice in Java?**

Answer:

The answer is No. Thread can only start once. If you try to start it for a second time, it will throw an exception, i.e., `java.lang.IllegalThreadStateException`.

---

## **192. Can we run a thread twice in Java?**

Answer:

The answer is Yes. When you call the `run()` method, it doesn't create a new thread. The `run()` method is treated as a normal method and pushed into the main stack, so the main thread would execute it. So, it's not multi-threading.

---

## **193. Why wait, notify and notifyAll is defined in Object Class and not on Thread class in Java?**

Answer:

In Java all object has a monitor. Threads waits on monitors so, to perform a wait, we need 2 parameters:

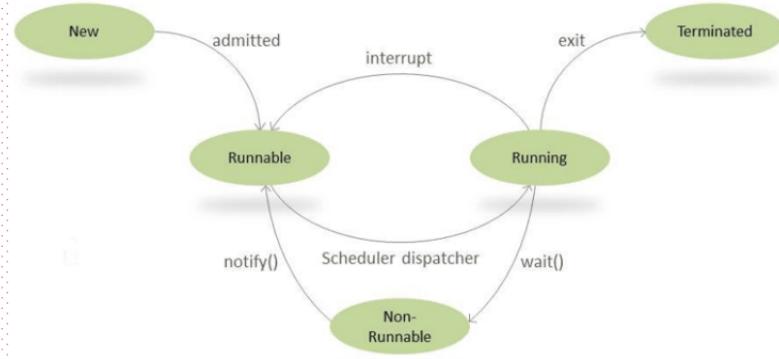
- a Thread
- a monitor (any object)

In the Java design, the thread can not be specified, it is always the current thread running the code. However, we can specify the monitor (which is the object we call wait on).

Wait and notify are communication mechanism between two threads in Java.

Threads needs lock and they wait for lock, they don't know which threads hold lock instead they just know the lock.

***wait() forces the current thread to wait until some other thread invokes notify() or notifyAll() on the same object.***



---

## 194. What are the different ways to achieve synchronization in Java?

Answer:

If the interviewer asks this .

Tell below 6:

### 1. Synchronized Methods:

Guarantees that only one thread can access a method at a time.

### 2. Synchronized Blocks:

Synchronizes only specific parts of a method, offering

- more fine-grained control.
  - 3. ReentrantLock:  
Provides more flexibility and control than synchronized methods, allowing explicit locking and unlocking.
  - 4. ReadWriteLock:  
Allows multiple threads to read data simultaneously, but only one thread can write at a time.
  - 5. Atomic Classes:  
Provides thread-safe operations on variables
  - 6. Volatile Keyword:  
Ensures that changes to a variable are visible to all threads, providing a guarantee of visibility.
- 

## **195. What is the difference between IllegalMonitorStateException and InterruptedException in Java?**

Answer:

These two exceptions are related to multithreading but occur in very different situations.

### **IllegalMonitorStateException**

This is an unchecked exception (RuntimeException). It occurs when a thread calls wait(), notify(), or notifyAll() without holding the monitor (lock) of that object. The reason is that inter-thread communication requires proper ownership of the lock, and if you do not own it, the JVM throws this exception.

### **Example:**

`Object lock = new Object();`

```
lock.wait(); // Throws IllegalMonitorStateException
```

### Correct usage:

```
synchronized (lock) {  
    lock.wait(); // Works fine  
}
```

### **InterruptedException**

This is a checked exception.

It occurs when a thread is in a waiting, sleeping, or joining state and another thread interrupts it.

The idea is to let the sleeping or waiting thread know that it should wake up and handle the interruption.

### Example:

```
Thread t = new Thread(() -> {  
    try {  
        Thread.sleep(5000);  
    } catch (InterruptedException e) {  
        System.out.println("Thread was interrupted");  
    }  
});  
  
t.start();  
t.interrupt(); // Causes InterruptedException
```

### In short:

IllegalMonitorStateException happens because the thread did not follow synchronization rules.

InterruptedException happens because another thread signaled interruption while it was waiting or sleeping.

## **196. What is the difference between Optimistic and Pessimistic Locking?**

Answer:

When multiple users or services update the same data, we need a strategy to avoid conflicts.

### Pessimistic Locking:

Lock first, update later. Other transactions wait.

Safe for high-conflict scenarios but can block threads.

### Optimistic Locking :

Assume conflicts are rare. Check before commit using a version field.

High throughput, no blocking, but may need retries.

### When to Use:

Pessimistic: Critical data, frequent conflicts (e.g., banking).

Optimistic: Low-conflict, scalable systems (e.g., e-commerce inventory).

---

## **197. What is the difference between Concurrency and Parallelism?**

Answer:

Candidates often confuse these two terms, but they are not the same.

### Concurrency

Concurrency means dealing with multiple tasks at the same time.

It does not always mean tasks are running at the exact same moment, but they are making progress together.

A single CPU core can achieve concurrency by switching between tasks quickly (context switching).

In Java, concurrency is often managed using threads, executors, and synchronization mechanisms.

### Example:

```
Thread t1 = new Thread(() -> System.out.println("Task 1"));
Thread t2 = new Thread(() -> System.out.println("Task 2"));
t1.start();
t2.start();
```

### Parallelism

Parallelism means actually running multiple tasks at the exact same time.

This requires multiple CPU cores or processors, where each core executes a different task simultaneously.

In Java, parallelism is achieved using parallel streams, the Fork/Join framework, or thread pools when the hardware has multiple cores.

### Example:

```
IntStream.range(1, 5)
    .parallel()
    .forEach(i -> System.out.println(i + " - " +
    Thread.currentThread().getName()));
```

---

## 198. How is the Fork/Join framework different from traditional thread pools?

Answer:

Fork/Join Framework: A framework designed for parallel processing by dividing tasks into smaller sub-tasks (fork) and then combining their results (join).

Difference: Unlike traditional thread pools, Fork/Join is designed to work with tasks that can be split into smaller tasks and allows for more efficient use of threads with work-stealing algorithms.

Aspect	Fork/Join Framework	Traditional Thread Pools
Task Division	Work-stealing algorithm for load balancing	Single queue, no automatic balancing
Task Type	Recursive, divide-and-conquer tasks	Independent tasks
Granularity	Fine-grained tasks	Coarser granularity
API and Usage	<code>ForkJoinPool</code> , <code>fork()</code> , <code>join()</code>	<code>ExecutorService</code> , <code>submit()</code> , <code>execute()</code>

---

## 199. What is the difference between CountDownLatch and CyclicBarrier in Java, and in what scenarios would you use each?

Answer:

CountDownLatch: Used to make one or more threads wait

until a set of operations in other threads complete (count down to zero). It cannot be reset once the count reaches zero.

CyclicBarrier: Used to make a set of threads wait for each other to reach a common barrier point. It can be reset, allowing it to be reused.

Usage:

Use CountDownLatch when you have a one-time event, such as waiting for multiple services to start, and

CyclicBarrier when you have repetitive tasks that need to synchronize at a certain point, like simulation steps.

Criteria	CountDownLatch	CyclicBarrier
<b>Reusability</b>	Not reusable; one-shot use.	Reusable; can be reset for multiple uses.
<b>Initialization</b>	Initialized with a count.	Initialized with the number of parties.
<b>Count Mechanism</b>	Count is manually decremented.	No count decrement; all parties must arrive.
<b>Barrier Action</b>	No action upon count reaching zero.	Optional action executed when all parties arrive.
<b>Wait Method</b>	<code>await()</code> until count is zero.	<code>await()</code> for all parties to arrive.
<b>Use Case</b>	One-time wait for multiple events to occur.	Multiple synchronization points in thread execution.

---

## 200. How do you handle thread interruption in Java?

Answer:

Thread Interruption: Allows one thread to signal another thread to stop what it is doing and do something else.

Handling: Discuss checking the interrupted status using Thread.interrupted() or Thread.isInterrupted() and reacting appropriately, such as by throwing an InterruptedException or performing cleanup tasks.

Significance: The interrupt() method is used to interrupt a thread, and the thread must handle the interruption correctly, especially in blocking operations like sleep(), wait(), or I/O.

---

## **201. How do you check if a Thread holds a lock or not?**

Answer:

The solution lies in Java's Thread class, which includes a method named holdsLock().

This method checks whether the current thread has acquired the monitor lock for a given object, returning true if it has, and false otherwise.

This functionality allows developers to query the lock status programmatically, which can be crucial for debugging or optimizing thread synchronization in concurrent programming scenarios.

---

## **202. How to get a thread dump in Java**

Answer:

Here's how to get a thread dump in Java:

Windows:

Press **Ctrl + Break** in the console where Java runs.

Linux:

Use `kill -3 <PID>` where `<PID>` is your Java process ID.

Universal (jstack):

Find the PID with `jps`.

Run `jstack <PID>` to get the dump.

This gives you a snapshot of all threads' states for debugging.

---

## 203. Difference Between `synchronized` and `ReentrantLock`?

Answer:

Below are the differences:

`synchronized`: Implicit, simpler, automatically releases the lock, but does not provide much flexibility.

`ReentrantLock`: More flexible, supports fairness, lock polling, and interruptible locks. It also provides more control with methods like `tryLock()`, `lockInterruptibly()`, and `unlock()`.

Criteria	<code>synchronized</code>	<code>ReentrantLock</code>
<b>Syntactic Sugar</b>	Keyword-based, simpler syntax.	Requires explicit lock and unlock calls.
<b>Flexibility</b>	Less flexible; no timeout or interrupt.	Highly flexible with options like <code>tryLock()</code> .
<b>Fairness</b>	Non-fair only.	Can be made fair or unfair.
<b>Interruptibility</b>	Not interruptible when waiting for lock.	Can be interrupted ( <code>lockInterruptibly()</code> ).
<b>Condition Variables</b>	Basic <code>wait()</code> , <code>notify()</code> , <code>notifyAll()</code> .	Rich <code>Condition</code> objects for complex signaling.
<b>Exception Handling</b>	Implicit in structure.	Requires explicit try-finally for safety.

**204. You have a scenario where multiple threads are contending for a lock, and you want to ensure that the lock is acquired in a fair order (i.e., first-come, first-served). How would you implement this in Java?**

Answer:

You can achieve fairness by using the fairness policy with `ReentrantLock`. A fair lock ensures that threads acquire the lock in the order they requested it.

The `ReentrantLock(true)` ensures that threads will acquire the lock in a fair, first-come, first-served order. This

prevents starvation but may reduce throughput due to context switching.

---

## 205. When is the volatile keyword used?

Answer:

The volatile keyword is used in Java to indicate that a variable's value will be modified by multiple threads. It ensures visibility and ordering of variable updates across threads. A volatile variable is directly read from and written to main memory.

Use cases:

To prevent caching issues in multithreading.

When threads must access the most recent value of a variable.

Example:

```
class SharedResource {  
    volatile boolean flag = false;  
}
```

---

## 206. Difference Between visibility and atomicity in multithreading?

Answer:

Below are the differences:

Visibility: Changes made by one thread are visible to others using volatile.

Atomicity: Operations are indivisible (e.g.,  
AtomicInteger.incrementAndGet()).

Scenario: A volatile int counter ensures visibility, but  
counter++ is not atomic since it involves multiple steps  
(read-modify-write).

---

## 207. What is the difference Between concurrency and parallelism ?

Answer:

**Concurrency:**

Overlapping execution of tasks.  
Shared CPU time; tasks might not run at once.  
Uses synchronization for thread interaction.

**Parallelism:**

Simultaneous execution of tasks.  
Multiple tasks run at the same time on different cores.  
Enhances performance with more cores.

**In Java:**

**Concurrency**: Thread, synchronized, Lock.

**Parallelism**: ForkJoinPool, ParallelStream.

---

## 208. What happens when an exception occurs inside a synchronized block?

Answer:

When an exception occurs inside a synchronized block, it behaves like any other exception in Java, but there are some important nuances:

- 1. Monitor Released:** If an exception occurs inside a synchronized block, the lock is released immediately.
- 2. Exception Propagation:** The exception propagates up the call stack, but the lock is still released.
- 3. Other Threads:** Other threads can acquire the lock once it is released.

This can lead to unexpected behavior if not handled properly, as other threads might access the object in an inconsistent state.

---

## 209. What are use cases of ThreadLocal variables in Java?

Answer:

**ThreadLocal** in Java is used when you need to have a variable that is local to each thread, meaning each thread can have its own instance of a variable.

Here are some brief use cases:

- **User Context:** Store per-thread user session data.
- **DB Connections:** Manage a database connection per thread.
- **Formatting:** Keep thread-specific formatters like `SimpleDateFormat`.
- **Metrics:** Track performance metrics for each thread's operations.

- Thread-Safe Instances: Use for objects that should be thread-confined but globally accessible within the thread.
- Web Requests: Hold request or session data in web applications.
- Local Cache: Implement a cache that's not shared across threads.
- Testing: Isolate test data or mocks per thread in concurrent tests.
- Logging: Add thread-specific context to log entries.

Remember, while `ThreadLocal` provides isolation, it's crucial to clean up `ThreadLocal` variables when they're no longer needed (especially in thread pool scenarios) to prevent memory leaks.

---

## 210. Write Producer/Consumer Problem using wait and notify?

Answer:

Use `wait()` to pause the producer or consumer if conditions are not met, and `notify()` or `notifyAll()` to wake up threads when conditions change.

This is often implemented using a shared queue and synchronization.

```
synchronized (queue) {  
    while (queue.isEmpty()) {  
        queue.wait(); // Wait until items are available  
    }  
}
```

```
// Consume item  
queue.notifyAll(); // Notify producer  
}
```

---

## 211. Explain the role of ExecutorService in the Executor Framework. What methods does it provide?

Answer:

ExecutorService is a subinterface of Executor that provides methods for managing the lifecycle of asynchronous tasks. It includes methods for:

- Task Submission: submit() to submit tasks and receive Future objects.
  - Shutdown: shutdown() and shutdownNow() to initiate a graceful or immediate shutdown.
  - Task Completion: invokeAll() and invokeAny() for executing multiple tasks and collecting results.
  - Scheduling: schedule() in ScheduledExecutorService for delayed and periodic task execution.
- 

## 212. What is the difference between submit() and execute() methods in the Executor Framework?

Answer:

The execute() method is defined in the Executor interface and is used to execute a Runnable task without returning any result. The submit() method, available in ExecutorService, can execute both Runnable and Callable

tasks and returns a Future object, allowing the caller to retrieve the result of the task or handle exceptions.

### Pointwise comparison:

#### execute():

- No return value.
- For Runnable tasks only.
- Less task control.

```
ExecutorService executor = Executors.newFixedThreadPool(1);
executor.execute(() -> {
    // Task that doesn't return a result
});
```

#### submit():

- Returns a Future.
- Works with both Runnable and Callable.
- Allows task management, cancellation, and result retrieval.

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(() -> {
    return 42;
});
Integer result = future.get(); // Get the result from Future
```

---

## 213. What is the RejectedExecutionHandler in ThreadPoolExecutor? How can you customize it?

**Answer:**

The RejectedExecutionHandler is an interface used by the ThreadPoolExecutor to handle situations where a task cannot be executed because the thread pool is at its capacity (either the core pool size and the maximum pool size are exhausted, or the queue is full).

#### **Default RejectedExecutionHandler Strategies:**

1. AbortPolicy (default): Throws a RejectedExecutionException when the task cannot be accepted.
  2. CallerRunsPolicy: Executes the task on the calling thread (i.e., the thread that made the request).
  3. DiscardPolicy: Ignores the task (i.e., it discards the task).
  4. DiscardOldestPolicy: Discards the oldest unprocessed task in the queue and tries to add the new task.
- 

#### **214. Explain the internal working of ThreadPoolExecutor and how it manages tasks in its different states?**

**Answer:**

ThreadPoolExecutor manages tasks using a pool of worker threads.

The internal process includes:

- **Core Threads:** Initially, core threads are created to handle incoming tasks. If all core threads are busy, tasks are added to the BlockingQueue.

- Queue Management: If the BlockingQueue becomes full, and the number of threads is below maximumPoolSize, new threads are created to handle the overflow. If both the queue is full and the pool is at max size, the RejectedExecutionHandler is invoked.
  - Worker States: Threads within the pool can be in various states like RUNNING, SHUTDOWN, STOP, TIDYING, and TERMINATED. These states manage the lifecycle of tasks and worker threads, particularly during shutdown or abnormal termination.
  - Keep-Alive Time: Idle threads exceeding the core pool size are terminated after a keep-alive time if the pool is not busy, helping in resource optimization.
- 

## **215. How does the Java Executor Framework handle task interruption, and what are best practices for managing interruptions in tasks?**

Answer:

The Executor Framework handles task interruptions primarily through the Future.cancel() method, which interrupts the running task if it is still active.

Best practices for managing interruptions include:

- Checking Interruption Status: Periodically check Thread.currentThread().isInterrupted() within tasks to handle interruption requests gracefully.

- Catching InterruptedException: Tasks should catch InterruptedException and either rethrow it or handle it in a way that stops the task execution cleanly.
  - Resource Cleanup: Ensure that tasks clean up any resources (like closing files or releasing locks) before terminating.
  - Interruptible Methods: Use interruptible methods like Thread.sleep(), Object.wait(), and BlockingQueue.take() to allow tasks to respond to interruption signals appropriately.
- 

## 216. How does ConcurrentHashMap work internally?

Answer:

One of the most asked interview questions in multithreading/concurrency .

- Segmented Locking (Pre-Java 8): The map was divided into segments, each with its own lock, allowing multiple threads to access different segments concurrently.
- Bucket-Level Locking (Post-Java 8): Instead of segments, Java 8 uses bucket-level locks and CAS (Compare-And-Swap) operations for atomic updates. This allows fine-grained locking, improving concurrency.
- Lock-Free Reads: Most read operations (`get()`, `containsKey()`) are lock-free and rely on volatile reads for visibility.

- TreeBin Structure: If a bucket has too many elements, the linked list in the bucket is converted into a balanced red-black tree for faster access.
  - Weakly Consistent Iterators: Iterators are weakly consistent, meaning they reflect the map's state at some point without being affected by ongoing modifications.
- 

**217. You have a scenario where multiple threads are contending for a lock, and you want to ensure that the lock is acquired in a fair order (i.e., first-come, first-served). How would you implement this in Java?**

Answer:

Use **ReentrantLock** with **Fairness** when you need a simple, built-in solution where fairness is preferable but not critical. Java's ReentrantLock allows for a fairness policy. When instantiated with true, it attempts to grant locks in the order requests were made, reducing the likelihood of thread starvation.

Custom Solutions might be necessary for scenarios with very specific fairness requirements or where performance is crucial, and you can optimize the locking mechanism.

Use Condition from `java.util.concurrent.locks` to manage waiting and signaling, ensuring threads wait for their turn.

---

## **IMPORTANT MULTITHREADING TERMS**

Understanding multithreading in Java is essential for creating efficient and scalable applications.

Below are the basic and advanced multithreading topics that are crucial for interviews.

It's important to know all these terms for interviews and also the interview questions for multithreading/concurrency covered later.

### **Basic Topics**

#### **1. Application, Process, and Thread**

- Application: A program that performs tasks, executed by a process.
- Process: A self-contained execution environment with its own memory space.
- Thread: A lightweight, smaller unit of execution within a process, sharing the same memory space as other threads in the process.

#### **2. How to Create a Thread**

Threads can be created by either:

- Extending the Thread class and overriding its run() method.

- Implementing the Runnable interface and passing it to a Thread instance.

Example:

```
public class MyThread extends Thread {  
    public void run() {  
        System.out.println("Thread running");  
    }  
}
```

```
Runnable myRunnable = () ->  
System.out.println("Runnable running");  
Thread thread = new Thread(myRunnable);  
thread.start();
```

### 3. How to Join a Thread

The join() method allows one thread to wait for another thread to complete its execution.

Example:

```
Thread thread = new Thread(() ->  
System.out.println("Running"));  
thread.start();  
thread.join(); // Main thread waits for this thread to  
finish
```

### 4. How to Interrupt a Thread

The `interrupt()` method is used to signal a thread that it should stop its execution. The thread can check its interrupted status using `isInterrupted()` or handle `InterruptedException`.

Example:

```
Thread thread = new Thread(() -> {
    while (!Thread.currentThread().isInterrupted()) {
        // Perform task
    }
});
thread.start();
thread.interrupt();
```

## 5. Thread Synchronization

Synchronization ensures that only one thread can access a critical section of code at a time, preventing data corruption.

Example:

```
synchronized (lockObject) {
    // Critical section
}
```

## 6. Happens-before Relationship

The happens-before relationship defines the order of operations in multithreading to ensure visibility and ordering of operations. It guarantees that one action is

visible to another.

Example:

```
int a = 1;  
int b = 2;  
a = b; // Happens-before relationship ensures 'a'  
       sees the updated value of 'b'
```

## 7. Locks Granularity

Granularity refers to the scope of locks. Fine-grained locks affect smaller code sections and increase concurrency, while coarse-grained locks affect larger code sections and are simpler but less concurrent.

## 8. Volatile Variables

The volatile keyword ensures that changes to a variable are visible to all threads immediately, preventing caching issues.

Example:

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

## 9. Atomic Variables

Atomic variables, provided in the `java.util.concurrent.atomic` package, support lock-free thread-safe operations.

Example:

```
AtomicInteger count = new AtomicInteger(0);  
count.incrementAndGet();
```

## 10. Wait/Notify and the Producer-Consumer Pattern

`wait()`, `notify()`, and `notifyAll()` methods facilitate communication between threads, often used in the Producer-Consumer pattern to manage resource production and consumption.

Example:

```
synchronized (lockObject) {  
    while (condition) {  
        lockObject.wait(); // Wait  
    }  
    lockObject.notify(); // Notify  
}
```

## 11. ThreadLocal Variables

`ThreadLocal` provides thread-local variables, which are isolated to the thread that sets them, preventing interference from other threads.

Example:

```
ThreadLocal<Integer> threadLocal =
```

```
ThreadLocal.withInitial(() -> 1);
int value = threadLocal.get();
```

## 12. Race Condition

A race condition occurs when the outcome depends on the sequence or timing of uncontrollable events, often resulting in incorrect behavior.

Example:

```
int count = 0;
count++; // Risk of race condition if multiple
threads execute this simultaneously
```

## 13. Deadlock

Deadlock is a situation where two or more threads are blocked forever, each waiting for the other to release resources.

Example:

```
synchronized (lock1) {
    synchronized (lock2) {
        // Deadlock risk if another thread acquires
        locks in reverse order
    }
}
```

## 14. Starvation

Starvation occurs when a thread is perpetually denied access to resources because other threads continually acquire those resources.

Example:

```
synchronized (lock) {  
    // Resource starvation if high-priority threads  
    keep accessing this block  
}
```

## 15. Livelock

Livelock occurs when threads keep changing states in response to each other, preventing progress, but remaining active.

Example:

```
while (condition) {  
    // Threads actively responding but not making  
    progress  
}
```

## Advanced Topics

### 1. Lock and ReentrantLock

ReentrantLock provides more sophisticated locking

capabilities than synchronized blocks, including try-locks and timed locks.

Example:

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try {
    // Critical section
} finally {
    lock.unlock();
}
```

## 2. Read-Write Lock

ReadWriteLock allows multiple readers or a single writer, enhancing performance when read operations are more frequent than write operations.

Example:

```
ReadWriteLock rwLock = new
ReentrantReadWriteLock();
rwLock.readLock().lock();
try {
    // Read operation
} finally {
    rwLock.readLock().unlock();
}
```

## 3. Condition Variables

Condition variables are used with locks to manage complex thread coordination and communication.

Example:

```
Condition condition = lock.newCondition();
condition.await(); // Wait
condition.signal(); // Notify
```

#### 4. Semaphore

Semaphore controls access to a resource pool by allowing a set number of threads to access it concurrently.

Example:

```
Semaphore semaphore = new Semaphore(3); // Allow up to 3 permits
semaphore.acquire();
try {
    // Access resource
} finally {
    semaphore.release();
}
```

#### 5. CyclicBarrier and Parallel Sum

CyclicBarrier allows a set number of threads to wait for each other to reach a common barrier point, useful in parallel computations like sum calculations.

Example:

```
CyclicBarrier barrier = new CyclicBarrier(4, () -> {  
    // Action after all threads reach the barrier  
});
```

## 6. CountDownLatch and Merge-Sort Algorithm

CountDownLatch allows one or more threads to wait until a set of operations are completed, which can be used in merge-sort to wait for sub-tasks to complete.

Example:

```
CountDownLatch latch = new  
CountDownLatch(2); // Wait for 2 threads  
latch.await(); // Wait  
latch.countDown(); // Count down
```

## 7. Exchanger

Exchanger allows two threads to exchange objects, useful for scenarios where tasks must swap data between threads.

Example:

```
Exchanger<String> exchanger = new  
Exchanger<>();  
String data = exchanger.exchange("Data");
```

## **8. Phaser**

Phaser provides a more flexible mechanism than CyclicBarrier and CountDownLatch for coordinating threads, especially in complex parallel tasks.

Example:

```
Phaser phaser = new Phaser(1); // Register main  
thread  
phaser.register(); // Register additional threads  
phaser.arriveAndAwaitAdvance(); // Synchronize  
threads
```

## **9. CopyOnWrite Collections**

CopyOnWrite collections (e.g., CopyOnWriteArrayList) create a new copy of the collection on each write operation, providing thread safety with less contention for read operations.

Example:

```
CopyOnWriteArrayList<String> list = new  
CopyOnWriteArrayList<>();  
list.add("Element");
```

## **10. NonBlocking Queues**

Non-blocking queues (e.g., ConcurrentLinkedQueue) provide thread-safe operations without blocking threads, enhancing concurrency.

Example:

```
ConcurrentLinkedQueue<String> queue = new  
ConcurrentLinkedQueue<>();  
queue.offer("Element");
```

## 11. Blocking\_Queues

BlockingQueue implementations (e.g., LinkedBlockingQueue) support operations that block when the queue is full or empty, useful for producer-consumer scenarios.

Example:

```
BlockingQueue<String> queue = new  
LinkedBlockingQueue<>();  
queue.put("Element"); // Blocks if queue is full
```

## 12. ConcurrentMap

ConcurrentMap provides a thread-safe map with atomic operations for common map methods.

Example:

```
ConcurrentMap<String, Integer> map = new
```

```
ConcurrentHashMap<>();  
map.put("key", 1);
```

## 13. Map-Reduce Algorithm

The Map-Reduce algorithm processes large data sets by dividing the task into a "map" phase, where data is processed in parallel, and a "reduce" phase, where results are aggregated.

Example:

```
// Map phase  
Map<Integer, List<String>> mapResults =  
    data.stream()  
        .collect(Collectors.groupingBy(String::length));  
  
// Reduce phase  
Map<Integer, Long> wordCount =  
    mapResults.entrySet().stream()  
        .collect(Collectors.toMap(Map.Entry::getKey, e -> (long) e.getValue().size()));
```

## 14. Executors

The Executor framework provides a high-level API for managing and controlling thread execution, allowing for task submission and management of thread pools.

Example:

```
ExecutorService executor =  
    Executors.newFixedThreadPool(10);  
    executor.submit(() -> System.out.println("Task  
executed"));  
    executor.shutdown();
```

## 15. Scheduled Tasks

Scheduled tasks can be executed periodically or at a fixed rate using the ScheduledExecutorService.

Example:

```
ScheduledExecutorService scheduler =  
    Executors.newScheduledThreadPool(1);  
    scheduler.scheduleAtFixedRate(() ->  
        System.out.println("Scheduled task"), 0, 1,  
        TimeUnit.SECONDS);
```

## 16. ThreadPoolExecutor and ThreadFactory

ThreadPoolExecutor is a versatile thread pool implementation that allows custom configuration. ThreadFactory can be used to create new threads with custom properties.

Example:

```
ThreadPoolExecutor executor = new  
    ThreadPoolExecutor(5, 10, 60, TimeUnit.SECONDS,  
    new LinkedBlockingQueue<>());  
    executor.setThreadFactory(runnable -> new
```

```
Thread(runnable, "CustomThread"));
executor.execute(() -> System.out.println("Task
executed"));
```

## 17. Fork-Join Pool

ForkJoinPool is designed for parallel processing of tasks that can be divided into smaller subtasks, using the fork-join framework for efficient execution.

### Example:

```
ForkJoinPool forkJoinPool = new ForkJoinPool();
forkJoinPool.submit() -> {
    // Task execution
});
forkJoinPool.shutdown();
```

## 18. CompletableFuture

CompletableFuture provides a way to write asynchronous, non-blocking code with a fluent API, handling future computations and combining multiple futures.

### Example:

```
CompletableFuture.supplyAsync(() -> "Hello")
    .thenAccept(result -> System.out.println(result));
```

## 19. Parallel Streams

Parallel streams in Java allow for parallel processing of

collections, utilizing multiple threads to perform operations concurrently.

Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    numbers.parallelStream().forEach(n ->
        System.out.println(n));
```

## 20. Spinlock and Busy Wait

A spinlock repeatedly checks a condition while consuming CPU resources. Busy wait refers to the loop that continuously checks for a condition to become true.

Example:

```
AtomicBoolean lock = new AtomicBoolean(false);
while (!lock.compareAndSet(false, true)) {
    // Spin wait
}
```

## 21. Lock-Free and Wait-Free Algorithms

Lock-free algorithms ensure that at least one thread will make progress, while wait-free algorithms guarantee that every thread will complete its operation in a finite number of steps.

Example:

Lock-free algorithms include AtomicInteger operations, and wait-free algorithms are more complex and often involve specialized data structures.

## **22. Throughput and Latency in Concurrent Applications**

Throughput measures the amount of work done in a given time frame, while latency measures the time taken to complete a single task. Optimizing both is crucial for performance.

## **23. Profiling**

Profiling involves analyzing the performance of an application to identify bottlenecks, resource usage, and optimize code. Tools like JVisualVM and YourKit are commonly used.

### **Example:**

Run your application with a profiler to collect data on method execution times, memory usage, and thread behavior.

## **24. Microbenchmarks with JMH**

Java Microbenchmarking Harness (JMH) is a toolkit for writing accurate and reliable benchmarks, helping to measure the performance of Java code with high precision

# **Chapter 15 : Junit**

## **218. What is the difference between Stub and Mock in Unit testing?**

Answer:

Stub: Provides predefined answers to calls during test, focusing on input-output behavior.

Mock: Not only provides answers but also verifies that interactions (like method calls) happened as expected.

**Stub = fake data provider**

**Mock = behavior verifier**

Both are test doubles, but used for different testing goals.

---

## **219. Can you test a private method using JUnit?**

Answer:

**Not recommended**, as private methods are implementation details.

Testing private methods directly isn't recommended because it tests implementation rather than behavior, potentially making tests brittle.

**But** ,Can be done using reflection, but often signifies a need to refactor or test through public methods.

---

## **220. How to test Exception in JUnit?**

Answer:

Use @Test(expected = ExceptionClass.class)

or

newer JUnit 5's assertThrows()

---

## **221. How would you test asynchronous methods with JUnit?**

Answer:

- JUnit 5: Use assertTimeout() or CompletableFuture for timeouts or completion checks.
  - Awaityility: A library for more complex async assertions.
- 

## **222. How would you mock static methods using JUnit.**

Answer:

PowerMock: Used with JUnit to mock static methods, but it's often seen as a design smell.

JUnit 5 with Mockito: Mockito now supports mocking static methods, but use sparingly as it might indicate code smell.

---

---

# **Chapter 16: JAVA Spring/ SpringBoot**

## **223. Why do we need Spring?**

Answer:

Don't just say "less boilerplate."

Cover these points about Spring :

### **1. Built on Strong Design Principles**

- Spring uses Inversion of Control (IoC) and Dependency Injection (DI).
- This means objects are not created manually; Spring handles it.

Result: loose coupling and better testability.

### **2. Manages Core Application Concerns**

Spring provides built-in support for:

- REST APIs
- Database access (JDBC, JPA)
- Transaction management
- Security

This lets developers focus on business logic.

### **3. Highly Modular Architecture**

Spring is not one big monolith.

Use only what you need:

- Spring Web for APIs
- Spring Data for DB operations
- Spring Security for auth

#### **4. Spring Boot Simplifies Everything**

- Auto-configuration
- Embedded server (like Tomcat)
- Production-ready setup with minimal config
- It reduces setup time and makes projects easier to start and maintain.

#### **5. Production-Ready by Default**

- Built-in logging
  - Profiles for different environments
  - Health checks and metrics
  - It's mature and widely used in large-scale apps.
- 

### **224. What exactly is a Spring Bean?**

Answer:

Don't just say: "It's an object."

Technically true, but you need to go deeper.

Cover these points :

1. A Spring Bean is just a Java object

But what makes it special is: Spring creates and manages it for you.

2. Managed by what?

By the Spring IoC container (Inversion of Control). It controls the object's lifecycle.

3. Why do we need it?

Instead of doing new MyBean() everywhere, Spring gives

it to you wherever needed , automatically.

#### 4. How do we tell Spring to create a Bean?

- Use annotations like @Component, @Service, @Repository, @Controller on your class
- Or use @Bean method inside a @Configuration class

#### 5. Where are all beans stored?

Inside something called the ApplicationContext , basically Spring's container or "bean warehouse".

---

### **225. What is bootstrapping in Spring Boot? Explain its importance**

Answer:

Bootstrapping in Spring Boot refers to the process of starting up and initializing a Spring Boot application. During bootstrapping, Spring Boot performs the following key tasks:

1. Spring Context Initialization: The application context is created and initialized. It scans for components, configurations, and beans.
2. Configuration Loading: Loads properties from application.properties or application.yml files and applies them.
3. Embedded Server Startup: If a web application is configured, it starts an embedded server (like Tomcat, Jetty, or Undertow).

4. Dependency Injection: Instantiates and injects dependencies into beans using Spring's IoC container.

Importance of Bootstrapping:

Provides automatic configuration, reducing boilerplate code.

Simplifies the startup process, enabling quick prototyping and deployment.

Integrates embedded servers, so no external server setup is needed.

Establishes the foundation for the application's lifecycle and environment setup.

Example:

```
@SpringBootApplication  
public class MyApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(MyApplication.class, args);  
    }  
}
```

In this example, `SpringApplication.run()` bootstraps the application by initializing the Spring context and starting the embedded server.

---

**226. What is the difference between these Java/Spring jargon like POJO, DTO, Entity, Bean, Repository?**

Answer:

Let's understand them:

## 1. POJO (Plain Old Java Object)

Just a regular Java class

No annotations, no frameworks

Only fields, constructors, and getters/setters

Think of it as the most basic data structure

## 2. Bean

A POJO that Spring manages

Must follow JavaBean rules (like having a no-arg constructor)

Registered in Spring via annotations like @ Component, @ Service, or @ Repository

All Beans are POJOs but Spring is in charge of creating and injecting them.

## 3. DTO (Data Transfer Object)

A POJO specifically used to carry data between layers

Contains only data - no logic, no annotations required

Commonly used between Controller and Service layers

Every DTO is a POJO, but not every POJO is a DTO.

## 4. Entity

A POJO that represents a database row

Annotated with @ Entity and used with JPA/Hibernate

Maps class fields to database table columns

## 5. DAO (Data Access Object)

A class that manually handles database operations

Typically uses JDBC or Hibernate to write queries

## 6. Repository

Spring's modern replacement for DAO

---

Uses @ Repository and Spring Data JPA  
No need to write queries , Spring can auto-generate them  
Cleaner, more maintainable way to access data

## 7. Service

Contains the business logic of your application  
Where decisions, calculations, and rules are applied  
Annotated with @ Service

## 8. Controller

Handles HTTP requests and responses  
Acts as the entry point for web APIs  
Annotated with @ Controller or @ RestController

## 9. Component

The base annotation for any Spring-managed class  
@ Service, @ Repository, and others are specialized  
versions of it  
Used to register general-purpose Spring beans

## 10. Configuration

A class that defines how beans are wired together  
Annotated with @ Configuration  
Often contains methods that create beans for the Spring  
context

---

## **227. What exactly is a Spring Container?**

Answer:

At the heart of Spring is the Spring Container.  
It's what creates, manages, and wires your objects (called  
beans).

---

## What is it?

The container is the engine of Spring's Inversion of Control (IoC).

Instead of you creating objects with new, the container creates them and injects dependencies where needed.

It also manages their full lifecycle - creation, initialization, destruction.

## Types of Containers

Spring provides different container implementations, the most common are:

1. BeanFactory - the simplest container, lazy loads beans, minimal features.
2. ApplicationContext - more advanced, eagerly loads beans, supports internationalization, event publishing, and AOP.

In real applications, we mostly use ApplicationContext.

---

## **228. What is the difference between @Component, @Repository, @Service, and @Controller annotations in Spring?**

Answer:

All of these annotations are specializations of

@Component and are used to define Spring-managed beans.

- @Component: General-purpose annotation indicating a Spring component.
  - @Repository: Indicates a Data Access Object (DAO). It also provides additional capabilities related to persistence exceptions.
  - @Service: Indicates a service layer class. It doesn't provide additional functionalities but is used to convey the intent.
  - @Controller: Indicates a Spring MVC controller. It handles HTTP requests and returns views.
- 

## 229. What is the difference between @Configuration and @SpringBootConfiguration?

Answer:

Both are about telling Spring: “This class has beans you should manage.”

But their use is slightly different.

### What is @ Configuration?

Comes from Spring Framework (not specific to Boot).  
Put it on a class to say: this class defines beans.  
Beans are created using @Bean methods.

### Example:

```
@Configuration  
public class MyConfig {  
  
    @Bean  
    public String appName() {  
        return "Hello Spring!";  
    }  
}
```

Spring will create a bean named appName and put it in the application context.

### What is @ SpringBootConfiguration?

Comes from Spring Boot.

It is built on top of @ Configuration.

Used on the main class of your Spring Boot app.

### Example:

```
@SpringBootApplication // includes @  
SpringBootConfiguration inside  
public class MyApp {  
    public static void main(String[] args) {  
        SpringApplication.run(MyApp.class, args);  
    }  
}
```

Here, @SpringBootApplication -> includes @SpringBootConfiguration.

This marks the entry point of the app.

**@SpringBootConfiguration is just a special @ Configuration.**

### The difference is purpose:

---

@Configuration: for your custom configs (e.g., DB, caching, security).

@SpringBootConfiguration: for the main Spring Boot app class.

---

## 230. What is the difference between @Component and @Bean?

Answer:

Don't say just one is **class-level** and one is **method-level**.

Cover these points :

1. **@Component** : Use when you wrote the class

You simply annotate your class with @ Component.

Spring will automatically detect it during component scanning.

No need to manually define or return anything.

Example:

```
@ Component  
public class UserService {  
}
```

Now Spring will auto-register UserService as a bean.

---

2. **@Bean** : Use when you don't control the class

You define a method inside a class marked with @ Configuration

You return the object you want Spring to manage.

This is helpful when the class you're using comes from a third-party library or needs custom config during creation.

### Example:

```
@ Configuration  
public class AppConfig {  
  
    @ Bean  
    public RestTemplate restTemplate() {  
        return new RestTemplate();  
    }  
}
```

Spring registers the returned RestTemplate as a bean.

### 3. When should you use which?

If you're writing your own service or component use @ Component.

If you're using someone else's class, or want to customize how the object is created use @ Bean.

If you need fine control (like passing constructor arguments or changing config) go with @ Bean.

Think of @ Component as "auto-register" and @ Bean as "manual register".

Both result in a Spring-managed bean, but how they get there is different.

---

## 231. How would you configure connection pooling in a Spring Boot application?

Answer:

Spring Boot uses HikariCP for connection pooling out of the box.

**Default HikariCP:** Include spring-boot-starter-jdbc or spring-boot-starter-data-jpa.

Configure in application.properties:

```
spring.datasource.hikari.maximum-pool-size=10  
spring.datasource.hikari.minimum-idle=5
```

This sets up the pool with a maximum of 10 connections and a minimum of 5 idle connections.

---

## 232. What's the default bean scope in Spring, and how does it differ from Prototype?

Answer:

The default scope for a bean in Spring is **Singleton**, where only one instance is created per Spring IoC container.

**Singleton:** One instance per context.

**Prototype:** A new instance each time the bean is requested.

---

Aspect	Singleton	Prototype
<b>Definition</b>	One instance per Spring container.	New instance created each time the bean is requested.
<b>Default Scope</b>	Yes (default if no scope is specified)	No (must be explicitly set)
<b>Usage</b>	For stateless beans or shared state beans.	For stateful beans or when isolation is required.
<b>Instance Creation</b>	One instance for all clients.	New instance per client request.
<b>Resource Usage</b>	Generally lower, one object in memory.	Higher, multiple objects in memory.
<b>Performance</b>	Better for stateless operations, no instantiation overhead after first call.	Can be slower due to repeated instantiation.
<b>Lifecycle Management</b>	Spring manages lifecycle callbacks once.	Lifecycle callbacks not managed by Spring by default.
<b>State Management</b>	Shared state across all clients.	Separate state for each client.
<b>Configuration</b>	No annotation needed or <code>@Scope("singleton")</code>	<code>@Scope("prototype")</code>
<b>Example Use Case</b>	Service layer beans, data source, config beans.	User session beans, beans with mutable state.

## 233. What are the different scopes in Spring?

Answer:

Spring supports several bean scopes:

- Singleton: (Default) A single instance per Spring IoC container.

- Prototype: A new instance is created each time the bean is requested.
- Request: A single instance per HTTP request (Web-aware scope).
- Session: A single instance per HTTP session (Web-aware scope).
- GlobalSession: A single instance per global HTTP session (Web-aware scope).
- Application: A single instance per ServletContext (Web-aware scope).

Scope	Description
<code>singleton</code> (default)	One shared instance per Spring container.
<code>prototype</code>	A new instance is created <b>each time</b> the bean is requested.
<code>request</code> (web)	One instance per HTTP request (valid only in web-aware apps).
<code>session</code> (web)	One instance per HTTP session.
<code>application</code> (web)	One instance per <code>ServletContext</code> (web app).
<code>websocket</code> (web)	One instance per WebSocket lifecycle.

---

## 234. What is the difference between ApplicationContext and BeanFactory in the Spring framework?

Answer:

**BeanFactory:** Basic container, lazy initialization, minimal features.

**ApplicationContext:** Extends BeanFactory, eager initialization, includes additional features like AOP, event handling, and internationalization.

Feature	BeanFactory	ApplicationContext
<b>Additional Features</b>	Basic	Advanced (events, i18n, auto-registration, annotation support)
<b>Bean Loading</b>	Lazy by default	Eager loading common
<b>Resource Loading</b>	Manual	Automatic
<b>Event Publication</b>	Not supported	Supported
<b>Annotation Support</b>	Limited	Extensive
<b>Context Awareness</b>	Limited	Enhanced
<b>Common Use Case</b>	Memory-sensitive	Most applications

---

## 235. What is the difference between @Autowired and @Inject annotation in Spring?

Answer:

@Autowired:

Spring-specific.

Has required attribute for injection control.

### @Inject:

Java standard (JSR-330), supported by Spring.

No required attribute, but defaults to required in Spring.

### Use:

@Autowired for Spring-centric projects.

@Inject for broader compatibility or standard adherence.

Feature	@Autowired	@Inject
Origin	Spring	JSR-330 (Standard)
Required Attribute	Has <code>required</code> attribute	No <code>required</code> attribute
Optional Dependencies	<code>required=false</code> for optional	Use <code>Provider&lt;T&gt;</code> for optional
Qualifier Annotation	Uses <code>@Qualifier</code>	Uses <code>@Named</code>
Framework Support	Spring-specific	Multi-framework support

---

## 236. Can you explain the complete request handling flow in Spring MVC, starting from when a client sends an HTTP request until the response is returned?

Answer:

This is one of the **most common Spring interview questions** because it tests whether you really understand the **internals** of the framework , not just how to write @Controller or @RestController. A candidate who can explain this flow clearly shows that they can debug tricky

issues like request not reaching the controller, wrong mapping, filters blocking, or views not rendering.

Here's how the request travels inside Spring MVC:

#### 1. Client → HTTP Request

A user hits a URL (say /users/123). The request first enters the application.

#### 2. Servlet Container → Filters

Before Spring even sees it, Servlet Filters (like authentication, logging, CORS) can intercept the request. These are configured at the web.xml or through @WebFilter/FilterRegistrationBean.

#### 3. DispatcherServlet (Front Controller)

Every Spring MVC app has a single DispatcherServlet. This servlet receives all requests and acts as the central hub.

#### 4. Handler Mapping

DispatcherServlet checks its HandlerMapping beans to decide which controller method should handle this request. Example: /users/123 → UserController.getUserById()

#### 5. Handler Interceptors (Pre-Handle)

Before the controller method executes, Interceptors can run logic like checking headers, adding attributes, or blocking invalid requests.

#### 6. Controller Execution

The mapped controller method executes. It processes the request, often talking to services and repositories.

---

Example: `getUserById(123)` fetches a user object.

## 7. Handler Interceptors (Post-Handle)

After the controller finishes but before the response is rendered, interceptors can modify the ModelAndView.

## 8. View Resolver / HttpMessageConverter

If it's a traditional MVC app, the ViewResolver picks the correct view (e.g., JSP, Thymeleaf).

If it's a REST API (`@RestController`), Spring uses HttpMessageConverters to serialize the response (e.g., JSON via Jackson).

## 9. DispatcherServlet Sends Response

The resolved response is handed back to the DispatcherServlet.

## 10. Filters (Response Phase)

Filters can again intercept during response, e.g., adding security headers, compressing payload, etc.

## 11. Client Receives Response

The final response (JSON, HTML, XML, etc.) goes back to the browser or client.

If you only say:

"Request goes to DispatcherServlet, then controller, then response comes back."

That's a junior-level answer.

If you say:

"Request first passes through Filters, then DispatcherServlet, then HandlerMappings, then

Interceptors, then Controller, then ViewResolver/MessageConverter, then back through Interceptors and Filters."

That's a senior-level answer.

---

## 237. What are the differences between @RestController and @Controller?

Answer:

- `@Controller` is used to mark a class as a Spring MVC controller where methods return ModelAndView objects or String (which are views).
- `@RestController` is a combination of `@Controller` and `@ResponseBody`.

It is used for RESTful services, where methods return data directly in the HTTP response body (typically JSON or XML).

Feature	<code>@Controller</code>	<code>@RestController</code>
Purpose	Traditional MVC controllers	RESTful web services
Response Type	Returns views (e.g., JSP)	Returns data directly (JSON, XML, etc.)
<code>@ResponseBody Annotation</code>	Needs <code>@ResponseBody</code> on methods for direct data return	Implicitly adds <code>@ResponseBody</code> to all methods
Usage Scenario	Web applications, view rendering	REST API development
Annotations Included	None beyond <code>@Component</code>	<code>@Controller + @ResponseBody</code>

---

## **238. You notice a delay when fetching millions of records through an endpoint in a Spring Boot application. How will you handle it?**

Answer:

Fetching and processing millions of records in a single request can lead to performance bottlenecks like high memory usage, increased latency, and potential timeouts.

Below are strategies to handle this scenario:

### **1. Pagination**

Instead of fetching all records at once, implement pagination to retrieve a subset of records in smaller chunks.

#### **Implementation:**

Use Pageable with Spring Data JPA:

```
public interface RecordRepository extends  
JpaRepository<Record, Long> {  
    Page<Record> findAll(Pageable pageable);  
}
```

#### **Controller Example:**

```
@GetMapping("/records")  
public Page<Record> getRecords(@RequestParam int page,  
@RequestParam int size) {  
    Pageable pageable = PageRequest.of(page, size);  
    return recordRepository.findAll(pageable);  
}
```

#### **Benefits:**

Reduces memory consumption.  
Improves response time for individual requests.

## 2. Asynchronous Processing

Perform the processing in the background and return the response to the client with a status update.

### Implementation:

Use @Async for asynchronous execution.

```
@Async  
public CompletableFuture<List<Record>> fetchRecordsAsync()  
{  
    List<Record> records = recordRepository.findAll();  
    return CompletableFuture.completedFuture(records);  
}
```

Inform the client about the processing status (e.g., via a unique request ID).

### Benefits:

Non-blocking endpoint.  
Enables parallel processing.

## 3. Streaming Data

Stream the records to the client instead of loading them all into memory.

### Implementation:

Use @ResponseBody and Stream:

```
@GetMapping(value = "/records", produces =  
MediaType.APPLICATION_JSON_VALUE)  
public ResponseEntity<StreamingResponseBody>
```

```
streamRecords() {  
    StreamingResponseBody stream = outputStream -> {  
        List<Record> records = recordRepository.findAll();  
        for (Record record : records) {  
            outputStream.write(record.toString().getBytes());  
            outputStream.flush();  
        }  
    };  
    return ResponseEntity.ok(stream);  
}
```

### Benefits:

Reduces memory usage.

Starts sending data to the client immediately.

## 4. Caching

If the records don't change frequently, cache the results using a caching mechanism like Redis or Ehcache.

### Implementation:

Annotate the method with @Cacheable:

```
@Cacheable("records")  
public List<Record> getRecords() {  
    return recordRepository.findAll();  
}
```

### Benefits:

Faster subsequent responses.

Reduces load on the database.

## 5. Optimize Database Query

Ensure proper indexing on frequently queried columns.

Use lazy loading for relationships.

Optimize SQL queries by fetching only required columns.

## 6. Data Partitioning and Sharding

Split large datasets across multiple databases or tables to improve query performance.

## 7. Batch Processing

If further processing of records is needed, process them in smaller batches using Spring Batch.

## 8. Returning Compressed Data

To reduce payload size, compress the response using Gzip:  
Configure compression in application.properties:

---

```
server.compression.enabled=true
server.compression.mime-types=application/json
server.compression.min-response-size=1024
```

---

## **239. Where is HandlerMapping stored in Spring?**

Answer:

HandlerMapping is a Spring MVC component that maps HTTP requests to handler methods. It is kept in the application context and instantiated during application startup.

Examples include RequestMappingHandlerMapping for @RequestMapping annotated methods.

## **240. What are atomic transactions, and how are they implemented in Spring?**

Answer:

Atomic transactions are operations that follow the “all or nothing” principle, meaning that a series of database operations (or any transactional operations) must either completely succeed or entirely fail.

If any part of the transaction fails, the entire transaction is rolled back to maintain data integrity.

This concept ensures:

Atomicity: All operations in the transaction are completed, or none are.

---

## **241. What is the DispatcherServlet in Spring Framework?**

Answer:

DispatcherServlet is the central front controller in Spring MVC that handles all incoming HTTP requests and delegates them to the appropriate components like controllers, view resolvers, and handlers.

Responsibilities:

1. Receives HTTP requests from the client.
2. Uses HandlerMapping to find the right controller.
3. Passes the request to the appropriate handler method.
4. Applies interceptors (if any).

5. Uses ViewResolver to resolve the final view.
6. Returns the rendered response to the client.

**Summary:**

- It's a Servlet defined in web.xml or auto-registered in Spring Boot.
  - Acts as the main entry point in the Spring MVC framework.
  - Implements the Front Controller pattern.
- 

**242. What is the primary difference between a Servlet Filter and an Interceptor in Java?**

**Answer:**

The primary difference lies in their scope and integration. Servlet Filters work at the web container level and apply to all incoming requests, acting on them before they reach servlets or JSPs.

Interceptors, however, operate at the Spring MVC handler level and are integrated with the Spring framework, allowing interaction with annotations, method parameters, and handler-specific logic.

Feature	Servlet Filter	Interceptor
Purpose	Pre-processes requests or post-processes responses for web resources.	Intercepts method calls or specific execution points in a Spring application.
Scope	Applies to all requests going through the servlet container (e.g., Tomcat).	Typically applies within the Spring Framework's context, intercepting MVC handler methods.
Configuration	Configured in <code>web.xml</code> or via annotations like <code>@WebFilter</code> .	Configured via Spring configuration, either XML or annotations like <code>@Component</code> or directly in configuration classes.
Execution Points	<ul style="list-style-type: none"> <li>- Can intercept requests before they reach the servlet or after response.</li> <li>- Operates at the HTTP request/response level.</li> </ul>	<ul style="list-style-type: none"> <li>- Before and after method execution.</li> <li>- Can intercept at different points like <code>preHandle</code>, <code>postHandle</code>, <code>afterCompletion</code> in Spring MVC.</li> </ul>
Lifecycle	Managed by the servlet container, part of the servlet lifecycle.	Managed by Spring's IoC container, lifecycle tied to Spring's component lifecycle.

<b>Applicability</b>	Broad, applies to all servlets or JSPs in the web application.	More specific, often used for intercepting controller methods or specific points in the request handling in Spring applications.
<b>Use Cases</b>	<ul style="list-style-type: none"> <li>- Authentication.</li> <li>- Logging.</li> <li>- Data compression.</li> <li>- URL rewriting.</li> </ul>	<ul style="list-style-type: none"> <li>- Logging method invocations.</li> <li>- Security checks.</li> <li>- Transaction management.</li> <li>- Performance monitoring.</li> </ul>
<b>Order of Execution</b>	Can be chained; order specified in configuration.	Order can be managed using <code>@Order</code> or <code>Ordered</code> interface in Spring.
<b>Access to Request/Response</b>	Direct access to <code>HttpServletRequest</code> and <code>HttpServletResponse</code> .	Typically accesses these through <code>HandlerInterceptor</code> interface methods in Spring.
<b>Framework Dependency</b>	Servlet API, works with any servlet container.	Spring Framework specific; dependent on Spring's handling mechanism.

## 243. How would you use an Interceptor in a Spring Boot application?

Answer:

### Step 1: Create the Interceptor Class

First, you need to define your Interceptor by implementing the `HandlerInterceptor` interface from Spring. This

interface provides methods for intercepting the request at different stages:

```
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;
import javax.servlet.http.*;

public class CustomInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
                             HttpServletResponse response,
                             Object handler) throws Exception {
        System.out.println("Pre Handle method is Calling");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request,
                           HttpServletResponse response,
                           Object handler,
                           ModelAndView modelAndView) throws Exception {
        System.out.println("Post Handle method is Calling");
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
                               HttpServletResponse response,
                               Object handler,
                               Exception ex) throws Exception {
        System.out.println("Request and Response is completed");
    }
}
```

## Step 2: Register the Interceptor

You need to register your custom interceptor with Spring's configuration. You can do this by creating a configuration class that extends WebMvcConfigurerAdapter or by using newer Spring methods.

## Step 3: Component Scanning (Optional)

If your Interceptor class is in a package that's not automatically scanned by Spring Boot, you might need to include it in component scanning

## **244. What is the difference between @Profile and @ConditionalOnXXX?**

Answer:

@Profile is used to activate beans based on environment-specific profiles (e.g., development or production).  
@ConditionalOnXXX (e.g., @ConditionalOnProperty, @ConditionalOnClass) allows for more granular control by conditionally creating beans based on specific conditions such as properties or class availability.

Feature	@Profile	@ConditionalOnXXX
Purpose	Environment-specific configuration	Conditional bean creation based on specific criteria
Scope	Broad (environment profiles)	Narrow (specific conditions)
Flexibility	Less flexible, tied to profiles	Highly flexible, condition-specific
Usage	For different deployment environments	For fine-tuning application behavior
Spring Boot	General Spring feature	Mostly Spring Boot specific

---

## **245. What are idempotent methods in REST, and why are they important?**

Answer:

Idempotent methods in REST are HTTP methods that can be called multiple times without different outcomes.

Examples are **GET**, **PUT**, and **DELETE**.

They are important because they ensure that even if a request is repeated due to network issues or retries, the result remains consistent.

Aspect	Description
<b>Definition</b>	An operation is idempotent if multiple identical requests have the same effect as a single request.
<b>Examples</b>	<ul style="list-style-type: none"><li>- <b>GET</b>: Fetching data doesn't change server state.</li><li>- <b>PUT</b>: Updating or creating a resource with the same data repeatedly results in the same state.</li><li>- <b>DELETE</b>: Deleting a resource is idempotent; multiple deletes do not change the outcome after the first successful delete.</li><li>- <b>PATCH</b> can be idempotent if designed carefully.</li></ul>
<b>Importance</b>	<ul style="list-style-type: none"><li>- <b>Reliability</b>: Safe for retries. If a request fails, it can be safely resent without risk of unintended effects.</li><li>- <b>Network Resilience</b>: Useful in environments with network issues where requests might time out or fail.</li><li>- <b>Client Simplicity</b>: Simplifies client logic; clients don't need to track whether a previous request succeeded or failed.</li><li>- <b>Server State Management</b>: Helps in designing APIs where the server can manage state transitions predictably.</li></ul>
<b>Safety vs Idempotence</b>	<ul style="list-style-type: none"><li>- <b>Safety</b>: A safe method (like GET) does not modify resources on the server. Idempotence includes safety but also covers methods that might change state.</li><li>- <b>POST</b> is neither safe nor idempotent by default.</li></ul>

---

## 246. Tell me some common HTTP status codes?

Answer:

200 OK: Successful request.

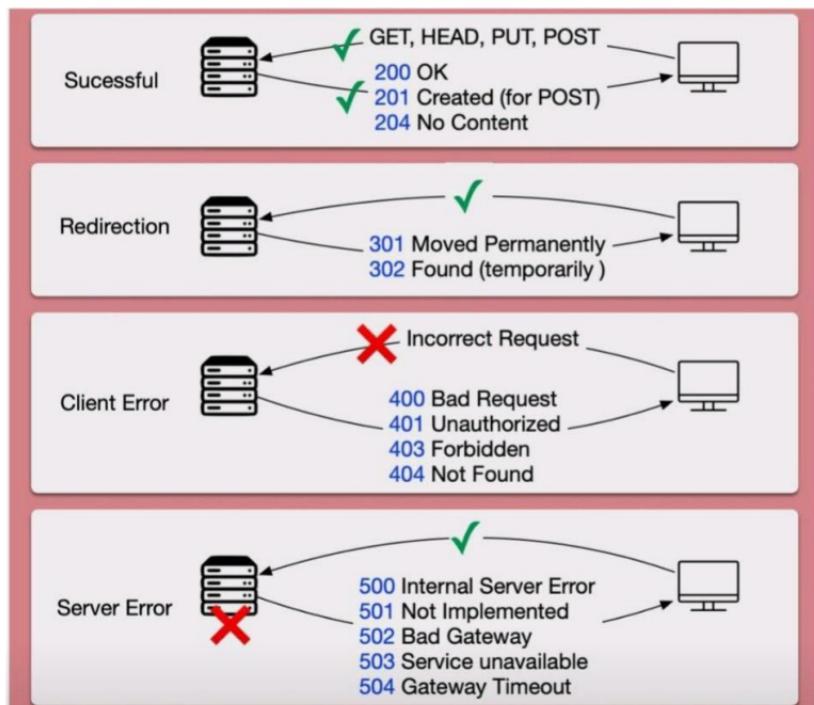
201 Created: A resource was successfully created.

204 No Content: Successful request with no body content.

400 Bad Request: Client-side error.

- 401 Unauthorized: Authentication required.
- 403 Forbidden: Authentication succeeded, but the user does not have permission.
- 404 Not Found: The requested resource does not exist.
- 500 Internal Server Error: Server-side error.

Status Code	Category	Meaning
200	Success	OK - The request has succeeded
201	Success	Created - New resource created
204	Success	No Content - Request succeeded, no content returned
400	Client Error	Bad Request - Invalid request
401	Client Error	Unauthorized - Authentication required
403	Client Error	Forbidden - Access to the resource is denied
404	Client Error	Not Found - Resource not found
429	Client Error	Too Many Requests - Rate limiting
500	Server Error	Internal Server Error - Unexpected condition
503	Server Error	Service Unavailable - Server temporarily down



## 247. What is the difference between PUT and PATCH?

Answer:

- PUT: Replaces the entire resource with the new data. If the resource doesn't exist, it can create it.
- PATCH: Updates only specific fields in the resource without replacing the entire entity.
- PUT: Idempotent—repeated requests have the same effect.
- PATCH: Ideally idempotent, but it depends on the

implementation.

- 
- PUT: Full update (e.g., updating an entire user profile).
  - PATCH: Partial update (e.g., updating just the email of a user profile).

---

## **248. What is ModelMapper in Java, and why is it used?**

Answer:

ModelMapper is a Java library used to map objects, particularly when converting between DTOs and entities. It simplifies the mapping process by automatically mapping fields with matching names and types.

It's particularly useful in large-scale applications where manual mapping would be tedious and error-prone.

---

## **249. What is the purpose of EntityManager in JPA?**

Answer:

EntityManager is the primary interface in JPA used to interact with the persistence context.

It provides methods for CRUD operations, queries, and transaction management, serving as the bridge between the application and the database.

---

## 250. How does EntityManager interact with the database in a Spring Boot application?

Answer:

In a Spring Boot application, the EntityManager is a key component of JPA (Java Persistence API) that interacts with the database.

Role of `EntityManager`:

- **CRUD Operations:** The `EntityManager` is responsible for basic CRUD (Create, Read, Update, Delete) operations on entities mapped to database tables.
- **Transaction Management:** It can manage transactions, either explicitly or through Spring's transaction management.
- **Query Execution:** Executes JPQL (Java Persistence Query Language) queries or native SQL queries.
- **Entity Lifecycle Management:** Manages the lifecycle of entities, including persisting, merging, removing, and refreshing entities.

In Spring Boot:

- **Auto-configuration:** Sets up `EntityManagerFactory`, `DataSource`, etc., based on your config.
- **Injection:** Use `@PersistenceContext` or `@Autowired` to inject `EntityManager`.

```
@Service
public class MyService {
    @PersistenceContext
    private EntityManager em;

    public void doSomething() {
        Entity e = em.find(Entity.class, 1L);
        // Use e
    }
}
```

- **Transaction Management:** Spring often handles transactions via `@Transactional`.
- 

## 251. What is Spring WebFlux, and how does it support reactive programming?

Answer:

**Spring WebFlux** is a part of the Spring Framework, introduced in Spring 5.0, specifically designed to support reactive programming for building web applications.

### Key Features:

- **Asynchronous:** Operations don't block threads, improving scalability.
- **Error Handling:** Sophisticated error management within data streams.
- **Reactive WebClient:** For making non-blocking HTTP requests.

Spring WebFlux is a framework for building reactive, non-blocking applications. It uses reactive streams and supports asynchronous processing, enabling applications to handle a large number of concurrent requests efficiently.

---

## 252. What are the benefits of using reactive

## **programming in a microservices architecture?**

Answer:

### **Reactive Programming Benefits for Microservices:**

- **Scalability:** High concurrency with minimal resource increase.
  - **Efficiency:** Optimal use of CPU and memory via non-blocking I/O.
  - **Resilience:** Robust error recovery, backpressure management.
  - **Performance:** Low latency, high throughput through asynchrony.
  - **Asynchrony:** Simplifies handling of service interactions.
  - **Real-time:** Efficient for real-time data streams.
  - **Decoupling:** Promotes service independence.
  - **Testability:** Easier to simulate and test complex behaviors.
  - **Integration:** Works well with modern reactive frameworks and libraries.
- 

### **253. What are the performance considerations when using ModelMapper in large-scale applications?**

Answer:

Reflection Overhead: ModelMapper relies on reflection, which can impact performance.

Complexity: Deep object graphs may slow down mappings.

Optimization: Use explicit mappings or

`setAmbiguityIgnored(true)` to avoid unnecessary computations. For large-scale use cases, alternatives like MapStruct are recommended.

---

## 254. How does Spring handle circular dependencies?

Answer:

Spring uses a three-phase approach to handle circular dependencies:

First Phase: Spring creates an instance of the bean but does not inject dependencies.

Second Phase: Spring injects dependencies, potentially circular ones, by setting the bean's references.

Third Phase: Spring calls the initialization methods like `@PostConstruct` or custom initialization methods.

- Circular dependencies involving singleton beans are resolved, but if you have a circular dependency in prototype-scoped beans, Spring will throw a `BeanCurrentlyInCreationException`.
- 

## 255. How does Spring Boot auto-configuration work?

Answer:

Spring Boot's auto-configuration automatically configures Spring beans based on the classpath settings, other beans, and various property settings.

The `@EnableAutoConfiguration` annotation triggers this process. Spring Boot uses `spring.factories` to define which configurations should be applied based on the available libraries and classes in the classpath.

---

## **256. How to change the packaging from JAR to WAR in a Spring Boot app?**

Answer:

Modify Build File: Change packaging in `pom.xml` to `war` or apply `war` plugin in `build.gradle`.

Extend `SpringBootServletInitializer`: Update your main class to extend this and override `configure`.

---

## **257. How to change the webserver to Jetty in a Spring Boot app?**

Answer:

Exclude Tomcat: From `spring-boot-starter-web` in your build file.

Add Jetty: Include `spring-boot-starter-jetty` dependency.

---

## **258. How to handle exceptions globally in a Spring Boot app?**

Answer:

### @ControllerAdvice:

Create a class annotated with @ControllerAdvice to catch exceptions across your application.

Use @ExceptionHandler to specify which exceptions to handle.

### @RestControllerAdvice:

Similar to @ControllerAdvice but for REST endpoints, automatically returning JSON responses.

### AOP (Aspect-Oriented Programming):

Use aspects to catch exceptions in a cross-cutting manner, useful for more complex handling.

### WebMvcConfigurer:

Configure global exception handling through Spring's MVC configuration.

Each method allows you to centralize your exception handling logic, improving maintainability and consistency in error responses across your application.

---

## **259. What is the difference between @Primary and @Qualifier?**

Answer:

### @Primary:

Used for: Setting a default bean when there are multiple choices.

Applied on: Bean definitions.

Effect: Chosen if no other criteria (like @Qualifier) are specified.

### @Qualifier:

Used for: Explicitly selecting a bean.

Applied on: Injection points.

Effect: Overrides @Primary for specific injection cases.

### **Key Points:**

@Qualifier takes precedence over @Primary.

@Primary helps set defaults; @Qualifier allows specific choices.

---

## **260. How can you integrate Spring Boot Actuator with external monitoring and alerting systems such as Grafana?**

Answer:

Steps for integrating Spring Boot Actuator with Grafana via Prometheus:

### **Spring Boot Actuator Setup:**

Add Dependencies: Include Actuator and Prometheus dependencies in your build tool.

Configure Actuator: Enable the Prometheus endpoint in your application properties.

### **Prometheus Setup:**

Install Prometheus: Use Docker or traditional installation.

Configure Prometheus: Set up prometheus.yml to scrape your app's /actuator/prometheus endpoint.

### **Grafana Setup:**

Install Grafana: Use Docker or install directly.

Add Prometheus as Data Source: Configure Grafana to connect to your Prometheus instance.

Dashboards: Import or create dashboards for your metrics.

### **Alerting:**

Create rules in Prometheus or directly in Grafana for notifications.

---

**261. Your production Java web server typically handles HTTP requests in <50ms, but now you see times raised to about 1 second. How would you investigate?**

Answer:

- 1. Performance Metrics:** Monitor performance metrics to identify any changes in response times or resource usage.
  - 2. Profiling:** Use profiling tools to identify slow methods or bottlenecks in the application.
  - 3. Check Logs:** Review logs for any errors, warnings, or unusual patterns around the time of increased response times.
  - 4. Load Testing:** Perform load testing
- 

**262. Can you explain different types of bean scopes?**

Answer:

Bean Scope defines the lifecycle and visibility of a bean within the Spring container. The common types are:

- **Singleton:** A single instance is created for the entire Spring container. The same instance is used throughout the application, making it the default scope.
- **Prototype:** A new instance is created each time the bean is requested. Suitable for beans with state or that need to be distinct.
- **Request:** Creates a new bean instance for each HTTP

request. Only applicable in web applications.

- **Session**: One instance per HTTP session. Useful for storing session-specific data in web applications.

- **Application**: One instance per ServletContext. Shares the bean across the entire web application.

Bean Scope	Description	Usage Example
<b>singleton</b>	Default scope. Only one shared instance of the bean will be managed by the IoC container for each Spring IoC container.	For stateless beans or services that do not need to maintain state between requests.
<b>prototype</b>	Creates a new bean instance every time the bean is requested. The container does not manage the lifecycle after creation.	For beans that need to maintain client-specific state, like beans created for each user request in a web application.
<b>request</b>	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request will have its own instance of a bean created off the same bean definition.	Useful in web applications where you want a new instance for each request, but it's managed by the container.
<b>session</b>	Scopes a single bean definition to the lifecycle of an HTTP <a href="#">Session</a> . Only valid in the context of a web-aware Spring <a href="#">ApplicationContext</a> .	For session data, like user preferences or session-specific data in web applications.
<b>application</b>	Scopes a single bean definition to the lifecycle of a <a href="#">ServletContext</a> . Only valid in the context of a web-aware Spring <a href="#">ApplicationContext</a> .	For beans that are shared across all users in a web application but should be bound to the lifecycle of an application.
<b>websocket</b>	Scopes a bean to the lifecycle of a WebSocket connection.	For beans that should exist for the duration of a WebSocket session.

---

## 263. Can we inject a prototype bean in a

## **singleton bean? If yes, what will happen if we inject a prototype bean in a singleton bean?**

Answer:

Yes, you can inject a prototype bean into a singleton bean. However, if you use direct injection (@Autowired), the prototype bean will be created once and reused. To get a new instance each time:

- Use @Lookup method injection or ObjectFactory to retrieve a new instance:

```
@Autowired  
private ObjectFactory<PrototypeBean>  
prototypeBeanFactory;  
  
public void someMethod() {  
    PrototypeBean bean = prototypeBeanFactory.getObject();  
}
```

---

## **264. How would you create a custom annotation in Spring to handle repetitive logic across multiple services or controllers?**

Answer:

Follow these steps:

- 1. Define Annotation:** Create an annotation named `Custom`.
- 2. Aspect Implementation:** Develop an aspect class that will execute whenever `Custom` is encountered.

3. Spring Configuration: Enable AspectJ in your Spring application to use aspects.

4. Usage: Apply `Custom` to methods or classes where you want the repetitive logic to run.

---

## 265. Where would you choose setter injection over constructor injection, and vice versa?

Answer:

- Constructor Injection: Use when dependencies are required and the object must be fully initialized upon creation. Ensures immutability and easier testing.
- Setter Injection: Use for optional dependencies or when you need to change dependencies after object creation. Provides flexibility.

Feature	Setter Injection	Constructor Injection
Immutability	Less immutable	Promotes immutability
Initialization	Post-construction	At construction
Optional/Mandatory	Easy for optional	Typically mandatory
Testing	Easier to mock	More complex for mocking
Readability	Less clear	More explicit
Best Practices	Less preferred	Generally recommended
Circular Dependency	More prone	Less likely

---

## **266. What happens when an @Transactional method calls another @Transactional method?**

Answer:

By default, the inner method runs within the same transaction.

Nested Transactions: Spring does not create separate nested transactions unless Propagation.REQUIRES\_NEW is used.

Handling: Use proper propagation settings like REQUIRES\_NEW for inner methods if a separate transaction is needed.

---

## **267. Can you provide an example of a real-world use case where @PostConstruct is particularly useful?**

Answer:

Here's a real-world use case for @PostConstruct without code:

### **Scenario:**

#### **Initializing a Resource Pool**

Imagine you're developing a system that handles video streaming. Before your application can serve videos, it needs to initialize a pool of video transcoders.

Why @PostConstruct: You want to ensure this pool is set up only after all necessary dependencies (like network configurations or resource limits) are available.

Application: After all Spring beans are created and dependencies injected, @PostConstruct on a method initializes the transcoder pool, setting up connections and configurations. This method might also validate these resources, ensuring they're ready for streaming requests.

This setup guarantees that the transcoding service is fully prepared before any video streaming request is processed, enhancing reliability and performance.

```
@PostConstruct  
public void init() {  
    // Initialize resources }
```

---

## 268. How can we dynamically load values in a Spring Boot application?

Answer:

By following ways:

- Using Profiles: Define environment-specific properties and activate profiles.
- Environment Variables: Configure values via system environment variables.
- Config Server: Use Spring Cloud Config Server for externalized configuration.

---

**269. Can you explain the key differences between YML and properties files, and in what scenarios you might prefer one format over the other?**

Answer:

**YAML:**

Pros: Human-readable, supports nested structures, inherent data types.

Use When: Dealing with complex configurations or needing readability.

yaml

server:

port: 8080

**Properties:**

Pros: Simple, widely supported, flat structure.

Use When: Configurations are simple, or compatibility is key.

Feature	YAML (YML) in Spring	Properties Files in Spring
Syntax	Hierarchical structure, uses indentation. Supports nested properties.	Flat structure, key=value pairs. No nesting.
Readability	More readable for complex configurations due to its structure.	Less readable for complex or nested configurations.
Handling of Lists	Lists are naturally represented (- <code>item1</code> , - <code>item2</code> ). Inline lists also possible.	Lists require repeating the key or using comma-separated values ( <code>my.list[0]=item1</code> ).
Type Safety	Direct support for various data types (strings, numbers, booleans, lists).	Everything is treated as a string; explicit type conversion might be needed in code.
Environment Variables	Supports <code>\$(VAR_NAME)</code> for variable substitution.	Also supports <code> \${...}</code> for variable substitution but less structured for complex scenarios.
Profiles	Can have multiple profiles in one file (- - separates profiles).	Separate files for each profile or use naming convention like <code>application-{profile}.properties</code> .

## 270. Have you worked with Spring Boot Actuator?

Answer:

Answer something like this :

"Yes, used it for health and metrics in production apps. Enabled /health, /metrics, added a custom HealthIndicator for Redis, and exposed Prometheus-compatible metrics. Also secured the endpoints using Spring Security and profiles."

---

## 271. Tell me some Key Endpoints of Spring Boot Actuator.

Answer:

1. /actuator/health: Provides the application's health status.

```
{  
  "status": "UP"  
}
```

2. /actuator/info: Displays metadata like build version.

```
{  
  "app": {  
    "name": "MyApp",  
    "version": "1.0.0"  
  }  
}
```

3. /actuator/env: Shows environment properties.

```
{  
  "propertySources": [  
    {  
      "name": "systemProperties",  
      "properties": {  
        "java.version": "1.8.0_181"  
      }  
    }  
  ]  
}
```

4. /actuator/metrics: Provides metrics such as memory usage.

```
{
  "jvm.memory.used": 1024,
  "jvm.gc.count": 5
}
```

## 5. /actuator/threaddump: Displays a thread dump.

```
{
  "threads": [
    {
      "name": "main",
      "state": "RUNNABLE"
    }
  ]
}
```

Endpoint	Description	Default Path	HTTP Method
<b>health</b>	Shows application health information.	<a href="#">/actuator/health</a>	GET
<b>info</b>	Displays arbitrary application info.	<a href="#">/actuator/info</a>	GET
<b>metrics</b>	Provides application metrics.	<a href="#">/actuator/metrics</a>	GET
<b>env</b>	Exposes properties from Spring's ConfigurableEnvironment.	<a href="#">/actuator/env</a>	GET
<b>loggers</b>	Shows and modifies the configuration of loggers in the application.	<a href="#">/actuator/loggers</a>	GET, POST
<b>mappings</b>	Shows all @RequestMapping paths.	<a href="#">/actuator/mappings</a>	GET
<b>threaddump</b>	Performs a thread dump.	<a href="#">/actuator/threaddump</a>	GET

## 272. Explain how actuator endpoints can be

## **customized and restricted based on roles.**

Answer:

Customizing and Securing Spring Boot Actuator Endpoints

### **1. Customize Actuator Endpoints**

Use management.endpoints.web.exposure.include and exclude in application.properties to control which endpoints are exposed.

You can also change their path using management.endpoints.web.base-path.

### **2. Restrict Endpoints by Role**

Use Spring Security to secure actuator endpoints.

Configure in application.properties:

```
management.endpoint.health.roles=ADMIN
```

Or restrict programmatically using HttpSecurity:

```
http
    .requestMatcher(EndpointRequest.to("health", "info"))
    .authorizeRequests()
    .anyRequest().hasRole("ADMIN");
```

### **3. Granular Access**

Some endpoints (e.g., health) allow partial access for all and full details only for authorized roles.

This lets you expose only what's needed and restrict sensitive info to specific user roles.

---

**273. If you need to authorize a request before calling a service method in Spring Boot, what will you do?**

**Answer:**

**Approach:**

Use Spring Security to handle authorization.

**Annotations:**

Apply @PreAuthorize or @Secured annotations on the service methods or controllers to enforce access control rules.

**Example:**

Annotate a method with

@PreAuthorize("hasRole('ROLE\_ADMIN')") to ensure only users with the ROLE\_ADMIN role can access it.

---

## **274. What will happen if setter-based and constructor-based injection are applied to the same class?**

**Answer:**

Constructor-based injection will handle required dependencies during object creation, ensuring essential components are set.

Setter-based injection can be used for optional or additional dependencies after the object is created, and it may override values set by the constructor.

---

## **275. What is the difference between Spring**

## **singleton and plain singleton?**

Answer:

### Spring Singleton:

- Managed by the Spring container.
- Only one instance per Spring container.
- The lifecycle and behavior can be influenced by Spring's configuration and lifecycle callbacks.

### Plain Singleton:

- Implemented using the Singleton design pattern in code.
  - One instance per JVM (or classloader).
  - Not managed by Spring; lacks Spring-specific lifecycle management features.
- 

## **276. Can we avoid this dependency ambiguity without using @Qualifier?**

Answer:

Yes, you can avoid dependency ambiguity by:

- Using @Primary: Marks a bean as the default choice when multiple beans are of the same type. Spring will prefer this bean when autowiring.

- Using Bean Names: Explicitly specify which bean to inject using the `@Resource(name="beanName")` annotation.

- Using @Inject with @Named: In addition to `@Qualifier`, `@Inject` and `@Named` (from JSR-330) can be used for

disambiguation.

---

## 277. Can we create a custom health indicator in Spring Boot?

Answer:

Yes, you can by implementing :

HealthIndicator: Define the health check in `health()`.

Example:

```
@Component
public class MyHealthIndicator implements HealthIndicator {
    public Health health() {
        return check() ? Health.up().build() :
Health.down().build();
    }

    private boolean check() {
        return true; // Your custom logic here
    }
}
```

---

## 278. How do you dynamically register beans at runtime in Spring Boot?

Answer:

In Spring Boot, dynamic bean registration allows you to create and register beans during application runtime instead of defining them upfront using annotations like

@Component or @Bean. This is useful when the beans you need depend on runtime conditions such as configuration files, database entries, or external inputs. There are two main ways to achieve this:

### 1. Using ConfigurableApplicationContext

You can register a fully constructed object as a singleton bean. This means Spring will manage the object just like any other bean. It's quick and ideal when the object is already available or constructed dynamically.

### 2. Using BeanDefinitionRegistry

Instead of passing an object, you define the class and let Spring handle its instantiation and dependency injection. This is a more flexible and Spring-native approach, especially when you want full control over bean lifecycle and wiring.

#### Why Use Dynamic Bean Registration?

- When bean creation is conditional or data-driven.
- When implementing plugin systems or multi-tenant configurations.
- To register services discovered at runtime (e.g., via classpath scanning or reflection).

It provides powerful flexibility but should be used with care to avoid introducing complexity or mismanaging the application context.

---

## 279. How to implement multi-tenancy in a Spring Boot application?

Answer:

Multi-tenancy lets one app serve multiple clients with isolated data. Common approaches:

1. **Separate database per tenant:** Strong isolation but complex management.
2. **Shared database, separate schemas:** Balanced isolation, switches schema per tenant.
3. **Shared database and schema:** Uses tenant ID in tables, simplest but less isolation.

In Spring Boot, use Hibernate's multi-tenancy features with a tenant resolver to identify the tenant and provide tenant-specific connections or filters. Choose the approach based on your isolation and complexity needs.

---

## 280. How to create a custom Spring Boot starter?

Answer:

A custom Spring Boot starter is a reusable module that packages dependencies and auto-configuration to simplify setup for specific functionality.

Steps:

1. **Create a new Maven/Gradle project** with a meaningful name like spring-boot-starter-xyz.
2. **Add dependencies** your starter will provide (e.g., libraries, other starters) in its pom.xml or build.gradle.
3. **Implement auto-configuration classes** annotated with @Configuration and @ConditionalOn... to load beans only when needed.

4. **Register auto-configuration** in META-INF/spring.factories (for Spring Boot <2.7) or META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports (for Spring Boot 2.7+), listing your config class.
5. **Publish the starter** to a Maven repository.
6. Users just add your starter dependency, and all config/beans load automatically.

This abstracts complex setup into a single dependency for easier reuse and consistency.

---

## **281. What happens internally during Spring Boot app startup?**

Answer:

Start your answer like this : "It bootstraps using [http://SpringApplication.run\(\)](http://SpringApplication.run()), triggers auto-configuration via spring.factories, loads beans via component scan, applies profiles, then starts the embedded server (like Tomcat)."

These 10 steps happen sequentially :

1. main() Method: Calls SpringApplication.run() to launch the app.
2. SpringApplication Setup: Detects app type (web, CLI), sets defaults.
3. Environment Loaded: Reads application.properties, env variables, command-line args.

4. ApplicationContext Created: Depending on app type, loads ApplicationContext.
  5. Component Scanning: Detects and registers beans annotated with @Component, @Service, etc.
  6. Auto-Configuration: Loads classes via @EnableAutoConfiguration, configures beans conditionally.
  7. Bean Creation: Instantiates beans, injects dependencies.
  8. Initialization Hooks: Runs @PostConstruct, CommandLineRunner, etc.
  9. Web Server Starts: Embedded Tomcat/Jetty starts if it's a web app.
  10. App Ready: Fires ApplicationReadyEvent. App is live and serving.
- 

## **282. How do you customize error responses in Spring Boot?**

Answer:

Start like this : "Created a global @RestControllerAdvice with @ExceptionHandler. Returned a standard error structure with timestamp, error code, message, and path. It keeps the API consistent."

Spring Boot allows you to customize error responses in several ways:

1. @ControllerAdvice + @ExceptionHandler  
Define global exception handlers to return custom messages and status codes.
2. Custom /error Endpoint  
Override the default error controller to change the structure of error responses.

**3. Custom ErrorAttributes**

Modify the default error details returned in JSON by customizing attributes like timestamp, message, and path.

**4. Custom Exception Classes**

Create your own exception and error response models for consistent API error structures.

These approaches help in returning meaningful, structured, and user-friendly error messages in REST APIs.

---

### **283. How do you reduce startup time in Spring Boot apps?**

**Answer:**

Start your answer like this : "Avoided unnecessary dependencies and lazy-loaded beans where applicable using @Lazy. Also analyzed startup logs with actuator/startup endpoint and tuned garbage collector flags in the JVM."

To optimize startup time in Spring Boot applications:

1. Use `spring.main.lazy-initialization=true`  
Delays bean creation until needed.
2. Avoid Classpath Scanning Overhead  
Limit `@ComponentScan` to specific packages.
3. Disable Unused Auto-configurations  
Use `@SpringBootApplication(exclude = {...})` or `spring.autoconfigure.exclude`.
4. Profile-specific Beans  
Load only what's required per environment using `@Profile`.
5. Remove Unused Starters

- Exclude heavy dependencies like Actuator or JPA if not used.
- 6. Enable Spring AOT + Native Image (GraalVM)  
Ahead-of-time compilation significantly reduces startup time.
- 7. Tune Logging Level  
Use minimal logging during startup (logging.level.root=WARN).

These optimizations help especially in microservices, CI pipelines, or serverless environments.

---

## 284. How do you test a Spring Boot application?

Answer:

Start with this : "Used @SpringBootTest for integration tests and @WebMvcTest for controller layer. Also used MockMvc and Mockito for unit testing services. Ensured proper use of profiles to avoid hitting real services in tests."

Spring Boot supports multiple testing layers:

### 1. Unit Tests

Test individual components (like services) using JUnit and Mockito without loading Spring context.

### 2. Slice Tests

Use annotations like @WebMvcTest, @DataJpaTest to test specific layers (e.g., controllers, repositories) with minimal context.

### 3. Integration Tests

Use @SpringBootTest to load the full application context and test components together.

---

#### 4. Mocking Dependencies

Replace real beans with mocks using @MockBean.

#### 5. Test REST APIs

Use TestRestTemplate or MockMvc to test controllers and endpoints.

#### 6. Use Profiles

Load test-specific configurations using  
@ActiveProfiles("test").

These layers help ensure correctness from unit-level logic to full end-to-end flow.

---

---

## **ADDITIONAL POINTS**

### **TOP 10 ANNOTATIONS THAT YOU SHOULD KNOW BEFORE GOING FOR AN INTERVIEW:**

#### 1. @SpringBootApplication

The @SpringBootApplication annotation is the cornerstone of any Spring Boot application. It is a composite annotation that combines @Configuration, @EnableAutoConfiguration, and @ComponentScan. This annotation marks the main class of a Spring Boot application and enables auto-configuration, component scanning, and configuration properties.

#### Key Points:

- Central entry point for Spring Boot applications.

- Automatically configures your application based on the dependencies on the classpath.
- Scans for components, configurations, and services in the application.

Interview Tip: Be prepared to explain the significance of each annotation combined within `@SpringBootApplication` and discuss scenarios where you might customize its behaviour.

## 2. `@EnableAutoConfiguration`

`@EnableAutoConfiguration` is one of the annotations integrated into `@SpringBootApplication`. This annotation instructs Spring Boot to automatically configure your application based on the dependencies present in your project.

### Key Points:

- Facilitates automatic configuration of the Spring application context.
- Can be customized by excluding certain configurations using `exclude` or `excludeName` attributes.

Interview Tip: You may be asked how to disable or customize specific auto-configurations in Spring Boot.

## 3. `@ContextConfiguration`

`@ContextConfiguration` is a Spring Test annotation that specifies how to load an `ApplicationContext` for test classes. It allows you to define the locations of the XML configuration files or annotated classes that will be used to configure the test context.

#### Key Points:

- Used in integration testing.
- Supports both XML and Java-based configuration.

Interview Tip: Be ready to discuss the differences between `@ContextConfiguration` and `@SpringBootTest`, and when to use each.

## 4. `@SpringApplicationConfiguration`

`@SpringApplicationConfiguration` was used in earlier versions of Spring Boot to specify the application configuration for integration tests. It has been deprecated in favour of `@SpringBootTest`.

#### Key Points:

- Deprecated in Spring Boot 1.4 and replaced by `@SpringBootTest`.
- Previously used to configure integration tests.

Interview Tip: Understand the reasons for its deprecation and the advantages provided by `@SpringBootTest`.

## 5. @ConditionalOnBean

@ConditionalOnBean is a conditional annotation that allows beans to be created only when certain other beans are present in the Spring context. It provides a powerful way to control bean creation based on the presence or absence of other beans.

### Key Points:

- Facilitates conditional bean creation.
- Useful for configuring beans based on the presence of other beans.

Interview Tip: You might be asked to explain how @ConditionalOnBean and @ConditionalOnMissingBean can be used together for more advanced configurations.

## 6. @Qualifier

@Qualifier is used in conjunction with @Autowired to resolve the ambiguity when multiple beans of the same type are available. It specifies which bean should be injected by name.

### Key Points:

- Helps in resolving dependency conflicts.
- Specifies which bean to inject when multiple beans of the same type exist.

Interview Tip: Be ready to discuss scenarios where @Qualifier is necessary and how it works in conjunction with @Primary.

## 7. @Async

@Async is an annotation that allows you to run methods asynchronously in a background thread pool. It enables parallel processing and improves the scalability of the application by freeing up the main thread for other tasks.

### Key Points:

- Facilitates asynchronous method execution.
- Improves application performance by executing time-consuming tasks in the background.

Interview Tip: Expect to be asked how to configure custom thread pools and handle exceptions in asynchronous methods.

## 8. @RestController vs @Controller

@RestController and @Controller are Spring annotations used to define web controllers. While @Controller is used in conjunction with @ResponseBody to return data in a web request, @RestController is a convenience annotation that combines @Controller and @ResponseBody, eliminating the need for the latter.

### Key Points:

- `@RestController` simplifies the development of RESTful web services.
- `@Controller` is more versatile, allowing for view resolution and template rendering.

Interview Tip: Be prepared to discuss scenarios where `@RestController` is more appropriate than `@Controller` and vice versa.

## 9. `@Conditional`

`@Conditional` is a versatile annotation that allows conditional bean registration based on a specific condition. It can be used to tailor the application context based on various factors like environment properties or the presence of other beans.

### Key Points:

- Enables conditional bean registration based on custom conditions.
- Often used in combination with custom Condition classes.

Interview Tip: You might be asked to create a custom condition using the `Condition` interface and explain how it integrates with the `@Conditional` annotation.

## 10. `@Transactional`

`@Transactional` is a crucial annotation in Spring that demarcates transaction boundaries. It ensures that the methods annotated with it are executed within a transactional context, with the ability to rollback in case of failures.

### Key Points:

- Manages transactions automatically.
- Can be applied at both the class and method level.
- Supports various propagation and isolation levels.

Interview Tip: Expect questions on different transaction propagation behaviours and how they impact the execution flow of methods in a Spring application.

---

## **SECURITY:**

Security is covered in interviews a lot these days. I would recommend to read about the below topics in detail.

**As a java RESTFUL APIs DEVELOPER , you should be familiar with the following key topics for creating secure applications:**

- OAuth 2.0 & OpenID Connect: For authorization and authentication flows.
- JWT: Token-based stateless authentication.
- RBAC: Assign permissions based on roles.
- Spring Security: Framework for securing applications.

- SSL/TLS, HTTPS: Secure communication over networks.
  - Key Management: Secure handling and rotation of cryptographic keys.
  - Injection & XSS Prevention: Sanitize inputs, use prepared statements, escape outputs.
  - CSRF Protection: Use tokens or SameSite cookies.
  - Input Validation: Use Hibernate Validator or similar for data validation.
  - API Gateway: Centralize security policies with tools like Spring Cloud Gateway.
  - Rate Limiting: Protect against abuse and DDoS.
  - CORS: Manage cross-origin requests safely.
  - HSTS: Enforce HTTPS connections.
  - mTLS: Two-way SSL for mutual authentication.
  - Logging & Monitoring: Implement audit logs and SIEM for security events.
  - Session Management: Secure session handling, expiration, and renewal.
  - RESTful Security: Proper use of HTTP methods, secure API design, and minimal data exposure.
  - API Versioning: Ensure security across API versions.
- 
- 

## **SPRINGBOOT PROJECTS RECOMMENDATION**

Over the years, I've received countless messages from

developers at all levels asking the same question: "How do I get good at Spring Boot?" Whether you're a junior developer just starting out or an experienced engineer looking to sharpen your skills, mastering Spring Boot can seem like a daunting task. Given its vast ecosystem and powerful capabilities, Spring Boot offers a steep learning curve, but with the right approach, you can gain proficiency in it.

The best way to truly understand Spring Boot is through hands-on experience. While theory and tutorials are important, there's no substitute for actually building projects. Many developers make the mistake of sticking only to tutorials and never diving into real-world applications, which leaves them with gaps in their knowledge when they face actual challenges in their projects or interviews.

I'll walk you through two real-world Spring Boot projects that I recommend you work on to accelerate your learning. These projects are designed to help you explore the most important aspects of the Spring ecosystem, including

database management, security, real-time messaging, and scalability. The practical experience you'll gain from these will not only help you understand Spring Boot better but also position you to succeed in interviews and on the job.

## **Project 1: Online Bookstore**

An Online Bookstore is a fantastic project to dive deep into the Java ecosystem while leveraging the full power of Spring Boot.

This project will familiarize you with the process of integrating different Spring Boot modules such as Spring Data JPA, Spring MVC, and Spring Security—skills that are highly sought after in both interviews and real-world application development.

### **Key Components**

1. User Registration and Management:
2. Book Catalog:

3. Shopping Cart and Orders:
4. Request Handling with Spring MVC:
5. Securing the Application:

### Why You Should Build It

- Full-Stack Mastery: You'll touch on frontend (with Spring MVC) and backend development (with Spring Data JPA, Spring Security).
- Real-World Relevance: E-commerce systems are in demand across industries, and the knowledge you gain will be applicable to many real-world applications.
- Interview-Ready: Many interviewers ask about e-commerce systems, and having this project in your portfolio demonstrates your ability to build production-level application

### Project 2: Real-Time Chat Application

A Real-Time Chat Application is an excellent project for

dive into WebSockets and real-time messaging, which are increasingly used in modern applications. Building such a project with Spring Boot will give you an understanding of how to handle real-time communication between multiple users in a scalable and efficient manner. In this project, you'll explore Spring's messaging capabilities, allowing you to implement bi-directional, low-latency communication.

In the process of building this application, you'll become proficient in setting up a WebSocket server and integrating it with authentication mechanisms using Spring Security. This project will also teach you how to handle concurrency and scalability—key concepts in building real-time systems.

### Key Components

1. WebSocket Integration:
2. User Authentication:
3. Message Broadcasting:
4. Private and Group Chat:
5. Concurrency and Scalability:

### Why You Should Build It

- Master WebSockets: You'll learn how to implement real-

time communication, which is a highly sought-after skill in today's development landscape.

- Advanced Spring Features: You'll go beyond the basics of Spring Boot and leverage its messaging and concurrency capabilities.
- Performance and Scalability: This project will challenge you to think about handling multiple users, session management, and ensuring that the chat system is scalable.

Both of these projects provide a deep dive into the Spring Boot ecosystem and cover essential topics that are not only relevant for learning but also crucial for interviews and real-world development. If you're looking to get good at Spring Boot, I highly recommend you tackle these projects.

These projects offer advanced challenges in areas like security, real-time messaging, and scalability, preparing you for complex, production-level scenarios.

---

---

# **Chapter 17: Hibernate**

Hibernate shouldn't be an issue , pretty simple, and limited interview questions .

If you know these 20 odd questions , Hibernate should be at the last of your worries list.

## **285. Explain the Hibernate architecture.**

Answer:

Hibernate Architecture includes:

Configuration: Sets up Hibernate properties.

SessionFactory: Creates Session instances.

Session: Provides CRUD operations and interacts with the database.

Transaction: Manages transactions.

Query: Executes HQL (Hibernate Query Language) queries

Entity: Represents database tables as Java objects.

---

## **286. How to set up Hibernate in Spring Boot?**

Answer:

1. Add Dependencies: Include spring-boot-starter-data-jpa and your database dependency (e.g., H2) in pom.xml.

2. Configure Application Properties: In application.properties or application.yml, set up database connection and Hibernate properties like:

```
spring.datasource.url=jdbc:h2:mem:testdb  
spring.jpa.hibernate.ddl-auto=update  
spring.jpa.show-sql=true
```

3. Create Entity Class: Annotate your class with @Entity.

4. Create Repository: Extend JpaRepository for database operations.

5. Run the Application: Spring Boot will auto-configure Hibernate.

---

## **287. What is the difference between @Entity and @Table annotations?**

Answer:

@Entity marks a class as a Hibernate entity.

`@Table` specifies the table name in the database. It is optional if the table name matches the entity name.

Feature	<code>@Entity</code>	<code>@Table</code>
Purpose	Marks class as JPA entity.	Specifies table details for entity mapping.
Usage	Essential for entity classes.	Optional for table customization.
Default	Entity name defaults to class name.	Table name defaults to entity name if not specified.

---

## 288. What is the difference between merge and update in Hibernate?

Answer:

### **merge():**

- Syncs detached entity state with session.
- Returns managed copy if no entity in session.
- Use when reattaching detached entities.

### **update():**

- Updates session-attached entity directly.
- Entity must be in session; otherwise, errors occur.
- Use when entity is already session-managed.

Operation	Purpose	Behavior	Use Case
merge()	Sync detached entity with session	Creates managed copy if no match in session; updates if match exists.	Reattaching detached entities
update()	Update session-attached entity	Directly updates existing session entity; entity must already be in session.	Entity already managed by session

---

## 289. What is HQL (Hibernate Query Language)?

Answer:

HQL is an object-oriented query language used to query entities in Hibernate.

It is similar to SQL but operates on entities rather than database tables.

### Key Features:

**Object-Oriented:** Queries are written using the names of classes and properties rather than table and column names.

**Portable:** HQL is database independent; a query written in HQL will work across different database systems without modification, assuming the mapping is consistent.

**Rich Query Capabilities:** Supports complex queries with joins, subqueries, aggregation, and more, similar to SQL but with an object model focus.

## Example:

```
Query query = session.createQuery("from Employee");
```

This query retrieves all instances of the **Employee** class.

---

## **290. What are @ManyToOne, @OneToMany, @OneToOne, and @ManyToMany associations in Hibernate?**

Answer:

@ManyToOne: Many entities can be associated with one entity.

@OneToMany: One entity can be associated with many entities

@OneToOne: One entity is associated with one other entity.

@ManyToMany: Many entities are associated with many entities.

Annotation	Relationship	Example	Mapping Details	Usage Notes
<code>@ManyToOne</code>	Many to One	Employee -> Department	Foreign key in "many" side (Employee).	Used on the "many" side.
<code>@OneToMany</code>	One to Many	Department -> Employees	Join column on "many" side or <code>mappedBy</code> on "one" side for bidirectional.	Used on the "one" side, <code>mappedBy</code> for bidirectional.
<code>@OneToOne</code>	One to One	Person -> Passport	Can share primary key or use foreign key; bidirectional with <code>mappedBy</code> .	One entity has <code>@OneToOne</code> , other might use <code>mappedBy</code> .
<code>@ManyToMany</code>	Many to Many	Student -> Courses	Requires join table ( <code>@JoinTable</code> ); <code>mappedBy</code> on one side for bidirectional.	Both sides can have annotation, one uses <code>mappedBy</code> .

## 291. What is the N+1 problem in Hibernate, and how is it resolved?

Answer:

**N+1 Problem:** Issuing 1 query for entities and N additional queries for each entity's associations.

Resolution:

1.JOIN FETCH: Use JOIN FETCH in your HQL to eagerly load the associated collection. Eg:

```
List<Department> departments =
```

```
session.createQuery("from Department d JOIN FETCH  
d.employees", Department.class).list();
```

2. Eager loading: Use fetch = FetchType.EAGER in the mapping to load associated entities in the same query:

```
@OneToOne(mappedBy="department", fetch=FetchType.EAGER)  
private List<Employee> employees;
```

3. @BatchSize(size = 10): Set fetch = FetchType.LAZY and use @BatchSize for lazy loading with reduced number of queries:

```
@OneToOne(mappedBy="department", fetch=FetchType.LAZY)  
@BatchSize(size=10)  
private List<Employee> employees;
```

4. Entity Graphs: Define fetch graphs with @EntityGraph.

---

## 292. What is a Hibernate Criteria API?

Answer:

The Criteria API provides a programmatic way to create and execute queries.

It is an alternative to HQL (Hibernate Query Language).

### Key Features:

**Type-Safe:** Uses classes and methods rather than strings for query construction, reducing errors from typos or incorrect field names.

**Dynamic Queries:** Allows for the dynamic creation of

queries based on runtime conditions, making it excellent for building queries in complex scenarios or for user-driven filtering.

**Criteria Objects:** Queries are built using Criteria objects, where you add restrictions, projections, ordering, and other query elements.

Usage:

```
Criteria criteria = session.createCriteria(Employee.class);
List<Employee> employees = criteria.list();
```

---

## 293. What are the different states of an entity in Hibernate?

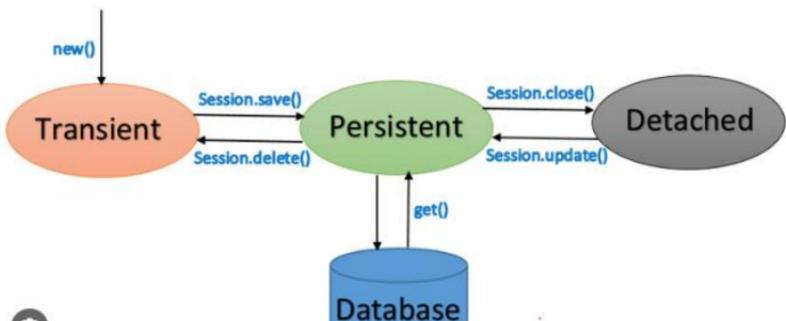
Answer:

Transient: The entity is not associated with a database session.

Persistent: The entity is associated with a session and mapped to the database.

Detached: The entity was associated with a session but is no longer attached.

Removed: The entity is marked for deletion.



State	Description	Characteristics	Example Action
<b>Transient</b>	Newly instantiated, not associated with session or database.	No ID, not in session, changes don't sync with DB.	<code>Employee employee = new Employee("John Doe");</code>
<b>Persistent</b>	Associated with a session, either newly saved or loaded from DB.	Has ID, changes tracked by session, syncs with DB on flush/commit.	<code>session.save(employee);</code> or <code>Employee emp = session.get(Employee.class, 1L);</code>
<b>Detached</b>	Was associated with a session but no longer is; retains identity and state.	Has ID, changes not tracked, can be reattached or merged.	<code>session.evict(employee);</code> or after <code>session.close();</code>

## 294. What is the difference between save, persist, and saveOrUpdate in Hibernate?

Answer:

1. save: Inserts a new entity and returns its identifier. It may not be synchronized immediately with the database.  
`session.save(entity);`

2. persist: Inserts a new entity without returning an identifier. Ensures synchronization with the database on transaction commit.

`session.persist(entity);`

3. saveOrUpdate: Inserts or updates an entity based on whether it has an identifier.

`session.saveOrUpdate(entity);`

Method	Purpose	Return Value	Behavior	Use Case
<code>save()</code>	Insert a new entity into database	<code>Serializable</code> (usually <code>Long</code> or <code>Integer</code> )	Inserts the entity into the database. If the entity is already associated with a session, behavior might be unexpected. Returns the ID.	When you need to insert and immediately know the new entity's ID.
<code>persist()</code>	Make entity persistent	<code>void</code>	Persists the entity to the session for later synchronization. Does not guarantee immediate database insertion. Does not return the ID.	When you want to manage the entity within a transaction but don't need the ID right away.
<code>saveOrUpdate()</code>	Insert or update based on entity's state	<code>void</code>	Determines if it should insert (save) or update the entity in the database based on whether the entity has an ID or not.	When unsure if the entity is new or existing; lets Hibernate decide whether to insert or update.

## 295. What is the difference between Session and SessionFactory?

Answer:

SessionFactory: A thread-safe object that creates Session instances. It is a long-lived object typically created once and used throughout the application.

Session: A single-threaded object that provides methods to perform CRUD operations. It is short-lived and should be opened, used, and closed within a transaction.

Aspect	SessionFactory	Session
Purpose	Factory for <code>Session</code> objects, manages configuration.	Manages persistence operations for a single unit of work.
Creation	Created once per application lifecycle.	Created per transaction or unit of work.
Thread Safety	Thread-safe, designed to be shared among threads.	Not thread-safe, should not be shared between threads.
Weight	Heavyweight; expensive to initialize.	Lightweight; cheap to create.
Mutability	Immutable once created; configuration is fixed.	Mutable; state changes during its lifecycle.
Cache	Manages second-level cache (if enabled).	Maintains a first-level cache for its lifecycle.
JDBC	Manages connection provider.	Uses connections from the <code>SessionFactory</code> .
Lifecycle	Long-lived; typically application-scoped.	Short-lived; created and closed for each transaction.
Usage	Used to create <code>Session</code> instances.	Used for CRUD operations, managing transactions.

## 296. How do you bring an entity from a detached to an attached state in Hibernate?

Answer:

- merge(): Reattaches a detached entity and returns a managed instance.

```
Entity managedEntity =  
    session.merge(detachedEntity);
```

- update(): Reattaches the entity, assuming it's in a persistent state.

```
session.update(detachedEntity);
```

---

## 297. What is hibernate.dialect?

Answer:

hibernate.dialect is a Hibernate configuration property that specifies the particular SQL dialect to be used for a database. This dialect setting tells Hibernate how to generate and interpret SQL statements for the specific database you're working with, ensuring compatibility and optimal performance.

**Common Dialects:** Here are some examples of dialects for popular databases:

- MySQL: `org.hibernate.dialect.MySQLDialect` or `org.hibernate.dialect.MySQL8Dialect` for MySQL 8.0+
- PostgreSQL: `org.hibernate.dialect.PostgreSQLDialect`
- Oracle: `org.hibernate.dialect.Oracle10gDialect` (versions vary)
- SQL Server: `org.hibernate.dialect.SQLServerDialect` or `org.hibernate.dialect.SQLServer2012Dialect` for SQL Server 2012+
- H2: `org.hibernate.dialect.H2Dialect`

## **298. What is @GeneratedValue in Hibernate?**

Answer:

@GeneratedValue specifies the strategy for generating primary key values, such as

AUTO, IDENTITY, SEQUENCE, or TABLE.

---

## **299. What is the difference between get and load in Hibernate?**

Answer:

get():

- Function: Retrieves an entity by its identifier. Returns null if the entity does not exist.
- Behavior: Always hits the database and returns a fully initialized object.

```
MyEntity entity = session.get(MyEntity.class, id);
```

load():

- Function: Retrieves an entity by its identifier. Throws ObjectNotFoundException if the entity does not exist.
- Behavior: Returns a proxy that is initialized on access. May not hit the database immediately.

```
MyEntity entity = session.load(MyEntity.class, id);
```

In summary, get() returns null if the entity is not found and

always hits the database,

while load() throws an exception if the entity is not found and may return a proxy.

	get()	load()
1	Return value null is possible	Never returns null
2	Fast if record exists	Slow if record exists
3	Used to retrieve object (record)	Used for delete etc. operations
4	Eager fetching	Lazy fetching
5	Always hits the database	Not always hits
6	Does not return proxy object	Always returns a proxy object
7	Performance-wise is slow as it may have to make number of rounds to database to get data	Better performance. If already in the cache, the record is available to much difference
8	As it returns null if no record exist, the execution continues	It throws ObjectNotFoundException, if record not found. Execution terminates if not handled successfully

---

### 300. What is the difference between first-level and second-level cache in Hibernate?

Answer:

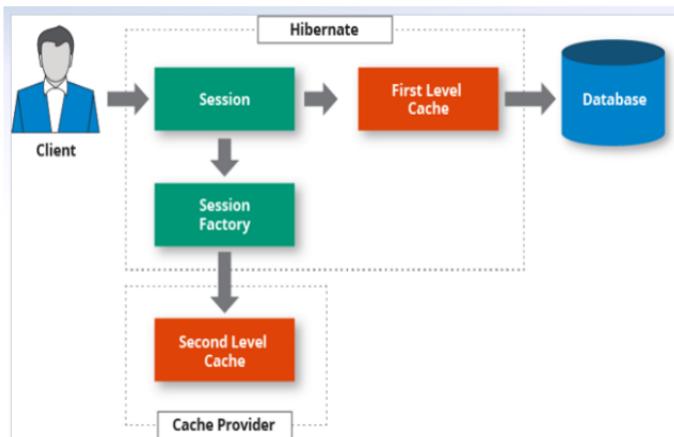
First-Level Cache: Also known as the session cache, it is mandatory and exists within the scope of a Hibernate session.

It ensures that the same object is not loaded multiple times within a session.

Second-Level Cache: Optional and shared among sessions. It is managed at the SessionFactory level and can be configured to use different caching providers (e.g.,

(EHCache, Infinispan).

It reduces database load by caching entities, collections, and query results.



---

### 301. What are the strategies for entity inheritance in Hibernate, and how do they affect performance?

Answer:

Single Table Inheritance: All classes in the hierarchy are mapped to a single table. This approach can be efficient for querying but can lead to sparse tables with many null values.

Table per Class Hierarchy: Each class in the hierarchy has its own table. This approach avoids null columns but can lead to complex joins and performance overhead.

Table per Concrete Class: Each concrete class has its own table. This approach avoids joins but can lead to redundant data and is less flexible for querying across subclasses.

---

### **302. What are Interceptors in Hibernate, and how can it be used for custom operations?**

Answer:

Interceptors: Allow custom logic to be executed during entity lifecycle events.

Implement Interceptor interface to define custom operations, such as logging changes or auditing.

Interceptors are configured via Hibernate settings or programmatically.

---

### **303. What are cascade and inverse types in Hibernate?**

Answer:

Cascade:

- Definition:

Specifies which operations (e.g., save, delete) should be applied to associated entities.

- Types:

`CascadeType.ALL`, `CascadeType.PERSIST`,  
`CascadeType.MERGE`, `CascadeType.REMOVE`, etc.

Example:

```
@OneToMany(cascade = CascadeType.ALL)  
private List<Book> books;
```

### Inverse:

#### - Definition:

Indicates the non-owning side of a bidirectional relationship, managed by the owning side.

#### - Usage:

Defined with mappedBy in @OneToMany or @ManyToMany.

### Example:

```
@OneToMany(mappedBy = "author")  
private List<Book> books;
```

---

## **304. What is hbm2ddl.auto in Hibernate?**

Answer:

hbm2ddl.auto is a Hibernate configuration property that controls how the database schema is managed at startup.

It has the following options:

- validate: Checks if the schema matches the entities, without making changes.
- update: Updates the schema without dropping existing data.

- create: Drops and recreates the schema, deleting existing data.
- create-drop: Drops the schema after session ends, used for testing.
- none: Disables automatic schema management.

It's critical to use the right setting for the appropriate environment (e.g., validate for production).

---

### 305. What is the @EntityListeners annotation in Hibernate?

Answer:

@EntityListeners allows defining callback methods to handle entity lifecycle events like @PrePersist, @PostUpdate, etc. It separates event handling from the entity logic.

```
@Entity  
@EntityListeners(AuditListener.class)  
public class User {  
    @Id  
    private Long id;  
    private String name;  
}  
public class AuditListener {  
    @PrePersist  
    public void prePersist(User user) {  
        // Logic before entity is persisted }}}
```

---

# Chapter 18: Java Messaging

## 306. What are the key differences between point-to-point (P2P) and publish/subscribe (Pub/Sub) models in JMS?

Answer:

Java Message Service (JMS) provides two primary messaging models: Point-to-Point (P2P) and Publish/Subscribe (Pub/Sub).

Here are the key differences between them:

Aspect	Point-to-Point (P2P)	Publish/Subscribe (Pub/Sub)
Destination	Queue	Topic
Consumers	One message per consumer	Multiple subscribers per message
Order	FIFO	No guaranteed order
Durability	Messages persist	Messages lost unless durable
Use Case	Single processing task	Broadcasting information
Acknowledgment	Required	For durable subscriptions only

### Point-to-Point (P2P):

Messages are sent to a queue.

A single consumer receives each message.

Suitable for task-based messaging where each message is processed once.

### Publish/Subscribe (Pub/Sub):

Messages are sent to a topic.

Multiple consumers can receive each message.

Suitable for event-driven architectures and broadcasting updates.

### Example Usage:

```
Queue queue = session.createQueue("TaskQueue");
Topic topic = session.createTopic("EventTopic");
```

---

## **307. How do you ensure message delivery reliability in JMS?**

**Answer:**

Ensuring message delivery reliability in Java Message Service (JMS) involves several strategies and mechanisms.

Here's how you can achieve reliability:

### 1. Acknowledgment Modes:

**AUTO\_ACKNOWLEDGE:** Messages are automatically acknowledged by the session once they have been processed. This is simple but less reliable if the application crashes after processing but before the acknowledgment.

**CLIENT\_ACKNOWLEDGE:** The consumer must explicitly acknowledge the message after processing. This gives more control over when a message is considered processed but requires more code management.

**DUPS\_OK\_ACKNOWLEDGE:** Allows for potential duplicates but acknowledges messages in batches, which is more efficient but might lead to some messages being processed twice.

2. Transactions: Messages can be sent or received within a transaction. If the transaction commits, all messages in the transaction are acknowledged or sent. If it rolls back, none are. This ensures atomicity in message processing.

### 3. Message Persistence:

Persistent Messages: Messages are stored to stable storage (like disk) before being acknowledged or consumed, ensuring they aren't lost if the JMS provider crashes.

Non-Persistent Messages: Faster, but messages can be lost if the JMS server crashes before the message is delivered.

4. Durable Subscriptions: For the Pub/Sub model, durable subscriptions allow subscribers to receive messages even if they were not active at the time of publication. The JMS provider stores messages until they are delivered to the subscriber.

5. Redelivery: If a consumer fails to acknowledge a message (due to an error in processing), the message can be redelivered after a certain timeout. This can be configured to retry a finite number of times or indefinitely.

---

## **308. What are the differences between synchronous and asynchronous message consumption in JMS?**

Answer:

### Synchronous:

Consumer explicitly calls `receive()` to fetch messages.

Blocks the thread until a message is available.

Suitable for low-frequency or on-demand processing.

### Asynchronous:

Use a MessageListener to handle messages automatically.

Non-blocking, allowing high-throughput systems.

---

## **309. How do you handle poison messages in JMS?**

### Answer:

A message that cannot be processed successfully, causing repeated failures.

### Handling Strategies:

1. Dead Letter Queue (DLQ): Move poison messages to a dedicated queue for inspection.
2. Retry Mechanism: Limit retries using redelivery policies.

### Example with Redelivery Policy:

```
connection.setExceptionListener(new ExceptionListener() {  
    @Override  
    public void onException(JMSEException e) {  
        // Handle poison message  
    }  
});
```

---

## **310. What is the difference between durable and non-durable subscriptions in JMS?**

### Answer:

### Durable Subscription:

Retains messages when the subscriber is offline.  
Use for critical systems requiring guaranteed delivery.

#### Non-Durable Subscription:

Messages are delivered only if the subscriber is active.  
Suitable for real-time, less critical applications.

#### Code Example for Durable Subscriber:

```
TopicSubscriber subscriber =  
    session.createDurableSubscriber(topic, "SubscriberName");
```

---

## **311. What KAFKA?**

Answer:

If asked in an interview about Kafka , give this definition  
Apache Kafka is a distributed event streaming platform for  
high-throughput, low-latency, scalable, durable, fault  
tolerant, real-time data processing.

**"Apache Kafka is an open-source, distributed event streaming platform.**

#### Core Components:

**Topics:** Data is categorized into topics, similar to categories or feeds.

**Producers:** These are systems or applications that send data to Kafka topics.

**Consumers:** Applications that read this data from topics for further processing.

#### Key Features:

**Scalability:** Kafka scales horizontally, capable of handling

millions of messages per second.

**Durability:** Offers fault tolerance through data replication across multiple servers.

**Low Latency:** Facilitates real-time data processing with very low latency.

**Replayability:** Allows consumers to read data from any point in time, which is useful for reprocessing or debugging.

#### **Use Cases:**

It's widely used for log aggregation, stream processing, real-time analytics, and as a backbone for microservices architectures in event-driven systems.

In essence, Kafka is about managing data streams efficiently, ensuring data integrity and availability, while providing the flexibility to scale with demand."

---

## **312. How does Apache Kafka achieve high throughput and low latency?**

**Answer:**

Apache Kafka achieves high throughput and low latency through several key design principles and optimizations:

### **1. Sequential Disk I/O:**

**Batching:** Kafka stores messages in a log format, which means messages are appended sequentially to the end of the file. Sequential I/O operations are much faster than random I/O, especially on magnetic disks but also beneficial for SSDs.

**Page Cache:** Kafka relies heavily on the operating

system's page cache, which minimizes the need for disk I/O by buffering data in memory. This also helps with read operations since the data might still be in the page cache.

## 2. Partitioning:

Scalability: Topics in Kafka are divided into partitions. Each partition can be handled by different brokers, allowing messages to be distributed across multiple machines. This parallelism in reading and writing helps in scaling out the throughput.

Order Preservation: Within a partition, the order of messages is guaranteed, but across partitions, order is not maintained. This design allows for high throughput without the overhead of maintaining global order.

## 3. Batch Processing:

Batch Writes: Messages are batched together before being written to disk, reducing the number of I/O operations per message. This is crucial for both throughput and latency.

Batch Fetches: Consumers can fetch messages in batches, which reduces network round trips.

## 4. Zero-Copy Optimization:

Sendfile: Kafka uses the sendfile system call, which allows the OS to transfer data from the kernel's page cache directly to the network without extra copying, significantly reducing CPU overhead.

## 5. Message Retention:

Log Compaction: Kafka can keep messages around for a configurable period (or based on size), allowing consumers to read at their pace, which helps in scenarios where consumers might be slow but does not impact the throughput of message production.

## 6. Replication:

**Leader-Follower:** Each partition has a leader and multiple followers. The leader handles all read and write requests, while followers replicate data for fault tolerance. This setup ensures data availability but is optimized so replication does not significantly impact latency or throughput.

---

### **313. What is an idempotent producer in Kafka, and why is it important?**

**Answer:**

An **idempotent producer** in Apache Kafka is a feature introduced in Kafka version 0.11 that ensures that messages are delivered exactly once to a specific partition of a topic, preventing duplication even if there are network errors or retries.

An idempotent producer is configured to send messages in such a way that, even if the producer retries sending a message due to failures (like network issues or broker unavailability), the message will not be duplicated in the Kafka topic partition.

Each producer is assigned a unique producer ID (PID). Messages are tagged with a monotonically increasing sequence number.

When the producer sends a message, the broker checks the sequence number against what it has already committed. If the sequence number is not the next expected number, the broker will ignore the message, thereby preventing duplicates.

The use of idempotent producers is particularly beneficial in scenarios where data accuracy is paramount, like financial transactions, audit logging, or any system where duplicate data could lead to errors or inefficiencies. By enabling idempotence, Kafka provides stronger delivery

semantics while maintaining performance, making it an essential feature for modern data streaming applications.

---

### **314. What is the difference between ActiveMQ, RabbitMQ and Kafka, and when would you use each?**

Answer:

**ActiveMQ:** Best for scenarios where JMS compliance and a variety of protocol support are crucial, with a focus on enterprise integration and reliability.

**Kafka:** Ideal for handling large data streams, providing high throughput, and when data persistence and replayability are needed.

**RabbitMQ:** Excellent for microservices, where you need flexible routing, support for multiple messaging patterns, and when you need a balance between performance and feature richness.

Feature/Aspect	ActiveMQ	Kafka	RabbitMQ
Type	Message Broker (JMS)	Distributed Streaming Platform	Message Broker
Messaging Model	Point-to-Point, Publish/Subscribe	Publish/Subscribe (with log-based storage)	Point-to-Point, Publish/Subscribe, RPC, etc.
Scalability	Good for enterprise applications	Highly scalable for big data scenarios	Good, with clustering for high availability
Performance	Moderate throughput, suited for complex routing	High throughput, low latency	High performance for message delivery
Persistence	Configurable (file or database)	Log-based, configurable retention	Message persistence supported
Data Model	Messages in memory or storage	Immutable logs (topics, partitions)	Messages in memory or on disk
Protocols	JMS, AMQP, MQTT, STOMP, etc.	Kafka's proprietary protocol	AMQP, MQTT, STOMP, HTTP, etc.
Use Cases	<ul style="list-style-type: none"> <li>- Enterprise integration</li> <li>- Transactional messaging</li> <li>- Multi-protocol support</li> </ul>	<ul style="list-style-type: none"> <li>- Real-time data pipelines</li> <li>- Log aggregation</li> <li>- Event sourcing</li> <li>- Stream processing</li> </ul>	<ul style="list-style-type: none"> <li>- Microservices communication</li> <li>- Task queues</li> <li>- Work distribution</li> <li>- Complex routing</li> </ul>

<b>Ordering</b>	Strict message order in queues	Order within a partition, not across	Order can be maintained within a queue
<b>Consumer Complexity</b>	Simple, broker handles much logic	More complex, consumers manage offsets	Moderate, with options like message acknowledgment
<b>Delivery Guarantees</b>	At-least-once, at-most-once, exactly-once	At-least-once by default, exactly-once with idempotency	At-least-once by default, exactly-once with confirmations
<b>When to Use</b>	<ul style="list-style-type: none"> <li>- When traditional messaging patterns are needed</li> <li>- When dealing with moderate message volumes</li> <li>- For integration across different protocols</li> </ul>	<ul style="list-style-type: none"> <li>- For high-throughput, real-time data processing</li> <li>- For scalable, fault-tolerant systems</li> <li>- When data needs to be retained and replayed</li> </ul>	<ul style="list-style-type: none"> <li>- For distributed systems needing robust message routing</li> <li>- When you need multiple protocol support</li> <li>- For microservices architecture needing queues</li> </ul>

Each of these systems has its strengths, and the choice often depends on the specific requirements of the application, including the volume of messages, the complexity of message interactions, and the ecosystem of tools and languages you're working with.

---

### 315. Explain the role of Zookeeper in Kafka and alternatives if needed.

Answer:

**Zookeeper** plays a crucial role in Apache Kafka for

managing and coordinating distributed systems:

**Cluster Membership:** Zookeeper keeps track of which brokers are part of the Kafka cluster, including their status (e.g., active, failed).

**Controller Election:** Kafka uses Zookeeper for electing a controller broker which is responsible for managing partitions and replicas across the cluster.

**Topic Configuration:** It stores metadata about topics, like their partitions' distribution across brokers.

**ACL Management:** Zookeeper manages Access Control Lists (ACLs) for topics, ensuring security and permissions are correctly enforced.

**Leader Election for Partitions:** It facilitates the election of leaders for partition replicas, ensuring data replication is managed efficiently.

**Synchronization:** Helps in synchronizing the state across the cluster, especially during broker failures or network partitions.

### **Alternatives to Zookeeper in Kafka**

Kafka has been moving towards reducing its dependency on Zookeeper for several reasons, including simplifying the architecture and reducing operational complexity. Here are some alternatives or modifications:

**Kafka Raft (KRaft):** With Kafka 2.8.0, Kafka introduced KRaft, which stands for Kafka Raft Consensus protocol. This is an effort to make Kafka self-sufficient by integrating the coordination logic that Zookeeper provides directly into Kafka.

---

## **316. How would you debug a scenario where Kafka consumers are lagging?**

**Answer:**

You can do following :

**1. Check Lag:**

Use `kafka-consumer-groups.sh --describe` to see current lag.

**2. Consumer Configs:**

Adjust `fetch.min.bytes`, `fetch.max.wait.ms`, `max.poll.records`.

**8. System Metrics:**

Monitor CPU, memory, network, disk I/O.

**9. Consumer Code:**

Profile for bottlenecks, optimize processing.

**10. Broker & Network:**

Check broker health, network latency.

**11. Rebalancing:**

Look at logs for frequent rebalances, adjust `max.poll.interval.ms`, `session.timeout.ms`.

**12. Topic Setup:**

Verify partition count, replication factor.

**13. Message Volume:**

Consider message size, production rate.

**14. Logs:**

Check for exceptions in consumer logs.

---

### **317. When would you choose Kafka over RabbitMQ?**

**Answer:**

When working with high-throughput systems and log-based streaming.

**Features:**

Distributed and fault-tolerant.

Supports massive data ingestion.

Highly suitable for event-driven architectures and

replaying messages.

---

### **318. When would you choose RabbitMQ over Kafka?**

**Answer:**

When low-latency messaging or complex message routing is required.

**Features:**

Supports Advanced Message Queuing Protocol (AMQP).  
Built-in support for message acknowledgment and retry.  
Easier to implement priority queues and direct message delivery.

---

---

# Chapter 19: Java Microservices

Some topics mentioned here require further exploration on your part, as it was not feasible to cover these extensive subjects in full detail within the scope of this interview book.

## 319. When is Microservices preferred over monolithic application?

Answer:

Below table highlights why microservices might be preferable in scenarios where flexibility, scalability, and fault isolation are crucial.

Aspect	Microservices	Monolithic
Scalability	Individual services can be scaled independently.	Scaling affects the entire application.
Technology Stack	Each service can use its own tech stack.	One unified stack for the entire application.
Maintenance	Services can be updated or fixed independently without impacting the whole app.	Any update or fix requires handling the entire codebase.
Development	Smaller, focused codebases; teams can work autonomously.	Larger codebase; changes might affect many parts of the application.
Fault Tolerance	Failures are isolated; other services continue running.	A failure can potentially bring down the entire application.
Business Alignment	Services can align directly with business capabilities.	Alignment might be less direct or harder to manage with growing complexity.
Integration	API-first design simplifies integration with other services or systems.	Integration points within the app can be complex if not well-architected.
Resource Use	Better resource allocation tailored to each service's needs.	Resources might be under or over-allocated for different parts of the app.
Testing	Easier to test individual services in isolation.	Testing can be more complex due to interdependencies within the app.

<b>Deployment</b>	Allows for continuous deployment of individual services.	Deployment involves the entire application, potentially slower updates.
-------------------	--	---

---

## 320. How Do You Handle Inter-Service Communication in Microservices?

Answer:

Communication between microservices can be handled using RESTful APIs, gRPC, message brokers (e.g., RabbitMQ, Kafka), or direct HTTP/HTTPS calls.

The choice depends on factors like latency requirements, data format, and whether synchronous or asynchronous communication is preferred.

---

## 321. What is the difference between Microservices, Event-Driven, and Distributed Architecture?

Answer:

Seen many candidates getting confused between Microservices, Event-Driven, and Distributed Architecture.

Because all three are related but not the same:

Distributed Architecture: The umbrella. Any system split across multiple machines.

Microservices: A way of doing distributed systems where each service handles one business capability.

Event-Driven Architecture: A way services communicate using async events (Kafka, RabbitMQ), not direct calls.

The confusion comes because:

Every microservices system is distributed.

Many microservices systems are also event-driven.

Event-driven patterns can be used in both monoliths and microservices.

Remember:

Distributed = where it runs

Microservices = how you split

Event-driven = how they talk

---

## 322. What Is Service Discovery and How Does It Work?

Answer:

Service Discovery in the context of Java, particularly within microservices architectures, is a mechanism that allows services to locate and communicate with each other dynamically.

Service Discovery is the process by which services in a distributed system (like microservices) find and interact with each other without hardcoding IP addresses or ports.

Implementation:

Eureka by Netflix is a popular service registry that Spring Cloud integrates with easily.

- Use `@EnableEurekaServer` on a class to create a registry server.
- Use `@EnableEurekaClient` on service classes to register them as clients.
- Configuration for Eureka is typically managed via `application.yml` or `application.properties`.

```
// Example of a Eureka Server
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}

// Example of a Service Client
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication
@EnableEurekaClient
public class MyServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyServiceApplication.class, args);
    }
}
```

## Benefits:

Dynamic Scaling: Services can be scaled up or down without service consumers needing to know about these changes.

Resilience: If a service instance fails, others can be discovered to handle requests.

Simplified Management: Centralized management of

service instances reduces complexity in service communication.

### Challenges:

Single Point of Failure: If the registry goes down, service discovery could fail, though this is mitigated with high availability configurations.

Network Overhead: Additional network calls for discovery might introduce some latency.

Service discovery in Java, especially with frameworks like Spring Cloud, simplifies the management of microservices by automating the process of how services find each other, making the architecture more robust and scalable.

---

## **323. What is Feign Client?**

Answer:

Before we get there, let's understand the journey step by step and get to know related terms first :

### **1. HttpClient**

The most basic way to make HTTP calls in Java.  
Comes with the JDK (Java 11 introduced a modern HttpClient).

Supports GET, POST, PUT, DELETE.  
Low-level and requires verbose code.  
Mostly used in standalone Java applications.

Before Java 11, developers mostly used two options to

make HTTP calls:

HttpURLConnection (built into JDK, very low-level and verbose).

Apache HttpClient (popular external library, easier and feature-rich).

## 2. RestTemplate

Spring introduced RestTemplate to simplify HTTP calls.  
Easier to use compared to raw HttpClient.

Example: `restTemplate.getForObject(url, String.class)`.

But Blocking in nature - the thread waits until the response is received.

Now in maintenance mode; not recommended for new projects.

## 3. WebClient

The modern alternative to RestTemplate, introduced in Spring 5.

Non-blocking and reactive.

Can handle a large number of requests with fewer threads.

Preferred in performance-intensive microservices.

Works well with reactive programming models.

## 4. Feign Client

A declarative HTTP client provided by Spring Cloud.  
Instead of writing code to make HTTP calls, you just declare an interface.

---

Spring generates the implementation automatically at runtime.

Integrates seamlessly with service discovery (like Eureka) and load balancing.

Can be combined with resilience libraries (like Resilience4j) for retries and circuit breakers.

### Example:

```
@FeignClient(name = "order-service")
public interface OrderClient {
    @GetMapping("/orders/{id}")
    OrderDTO getOrder(@PathVariable("id") Long id);
}
```

Now, calling another service is as simple as:

```
orderClient.getOrder(1L);
```

---

## **324. What is the difference between Forward Proxy and Reverse Proxy?**

Answer:

Let's understand these terms :

### 1. Proxy (in general)

A proxy is a middleman between two systems. It takes a request from one side, sends it to the other, and returns the response.

### 2. Forward Proxy

---

Sits between the client and the internet  
Hides the client's identity from servers  
Used for anonymity, bypassing restrictions, caching

Examples: Squid Proxy, VPN services

Client → Forward Proxy → Internet

### 3. Reverse Proxy

Sits between the internet and backend servers  
Hides servers from clients  
Used for load balancing, SSL termination, caching, security

Examples: NGINX, HAProxy, AWS ALB, Cloudflare

Client → Reverse Proxy → Backend Servers

### 4. Some other related terms that you might have heard of :

Load Balancer - A load balancer is a type of reverse proxy, but with a narrower goal: distribute incoming traffic across multiple servers.

Its focus is purely on performance and availability.  
It ensures that no single server is overloaded and provides failover if one server goes down.

Examples: HAProxy, AWS ELB, F5.

### Load Balancer vs Reverse Proxy :

A reverse proxy can do many things: security, caching, routing.

---

A load balancer is mainly concerned with splitting traffic across servers.

In short, all load balancers are reverse proxies, but not all reverse proxies are load balancers.

API Gateway – Reverse proxy with API management features (Kong, AWS API Gateway)

CDN (Content Delivery Network) – Reverse proxy that caches and delivers content closer to users (Cloudflare, Akamai)

In short:

Forward Proxy protects the client

Reverse Proxy protects the server

---

## 325. What is BFF?

Answer:

For developers like us(welcome to Java dev life 😊)

BFF != Best Friends Forever

BFF = Backend-for-Frontend.

It's a pattern where instead of one backend serving all clients, you create a dedicated backend per frontend (web, mobile, smart devices).

Why?

Different frontends need different data.

Mobile wants lightweight responses, web can handle detailed ones.

It hides microservices complexity from UI teams.

Real-life example:

Netflix – web app shows rich details, mobile just needs quick play info, TV app optimized for slower devices. Each gets its own BFF tailored to its needs.

---

## **326. What is API Gateway and Why Is It Used?**

Answer:

An API Gateway is a single entry point for client requests, handling routing, load balancing, security, and request aggregation.

Functions:

- Routing: Directs requests to the appropriate microservice.
- Security: Manages authentication and authorization.
- Aggregation: Combines responses from multiple services.

Common Tools:

- Spring Cloud Gateway: For routing and filtering in Spring Boot.
- Zuul: Netflix's gateway for routing and filtering.
- Kong: Open-source API Gateway for load balancing and security.
- Nginx: Can act as an API Gateway with reverse proxy capabilities.

Example (Spring Cloud Gateway):

@Configuration

```
public class GatewayConfig {  
    @Bean  
    public RouteLocator  
    customRouteLocator(RouteLocatorBuilder builder) {  
        return builder.routes()  
            .route("example_route", r -> r.path("/example/")  
                .uri("lb://example-service"))  
            .build();  
    }  
}
```

#### Benefits:

- Centralized Management: Simplifies client interactions.
  - Enhanced Security: Single point for security policies.
- 

### **327. What are the Challenges of Testing Microservices?**

#### **Answer:**

Challenges include testing the interactions between services, managing test environments, and ensuring that changes in one service do not affect others. To address these, use integration tests, contract tests (e.g., Pact), and employ mocking frameworks. Implementing a CI/CD pipeline can also streamline the testing process.

---

### **328. How do you ensure Security in a Microservices Architecture?**

#### **Answer:**

Ensure security by implementing OAuth2 or JWT for

authentication and authorization, using API gateways for centralized security policies, and employing encryption for data in transit and at rest.

Regular security assessments and adhering to best practices for securing APIs are also essential.

**OAuth 2.0** is solely about authorization, allowing applications to access user resources without sharing credentials.

**OpenID Connect (OIDC)** builds on OAuth 2.0 to add authentication capabilities, providing a way to verify user identity across multiple services with a standardized set of claims in the ID Token.

**JWT** is a token format that can be used in both OAuth 2.0 (as Access Tokens) and OIDC (as ID Tokens or Access Tokens) for carrying information securely. It's versatile but doesn't specify the protocol for obtaining or using the token.

---

### **329. Can you explain the Command Query Responsibility Segregation (CQRS) pattern and its application in a microservices architecture? What are its benefits and challenges?**

Answer:

CQRS separates read operations (queries) from write operations (commands).

#### Application in Microservices:

- Commands: Handle state changes (e.g., updating orders).

- Queries: Retrieve data (e.g., fetching order history).

#### Benefits:

- Scalability: Independent scaling of read and write services.
- Performance: Optimizes data access and modification.
- Flexibility: Different data models for reads and writes.

#### Challenges:

- Complexity: Increased system complexity and data synchronization.
- Eventual Consistency: Delays in data consistency.

#### Example:

In an e-commerce system, separate services handle order creation (commands) and order queries, allowing optimized performance for each.

---

## **330. What patterns do you use to ensure fault tolerance in microservices?**

Answer:

You can discuss below 4 patterns;

Circuit Breaker (Resilience4j, Hystrix) — Prevents cascading failures when a service is down.

Retry Pattern — Automatically retries a failed request before failing.

Bulkhead Pattern — Isolates failures to prevent them from affecting the entire system.

Failover Mechanisms — If one service fails, route traffic to another instance.

---

### **331. What is the difference between an API Gateway and a Load Balancer?**

Answer:

Cover these points:

#### **1. What is a Load Balancer?**

Imagine you have 3 copies of the same app running.  
A Load Balancer sits in front and distributes user traffic evenly to all 3.

It makes sure no single app is overloaded.  
If one app crashes, it skips that and sends traffic to the others.  
Think of it like a traffic cop that balances traffic on a highway.

Purpose: Distribute load across multiple same servers  
Common in monolithic or replicated service setups

#### **2. What is an API Gateway?**

Now imagine you have different microservices:

/login → Login Service  
/orders → Order Service  
/cart → Cart Service

An API Gateway acts as a single entry point.

It checks the path and routes to the right service.

It also does authentication, logging, rate limits, etc.  
Think of it like a receptionist who sends you to the right department.

Purpose: Route traffic to different services, add features  
Common in microservice architecture

### 3. Can they be used together?

Yes!

You first hit the API Gateway

Then inside, it may use a Load Balancer to talk to multiple instances of one microservice.

---

## **332. Can you explain the Circuit Breaker pattern and how it can be implemented in a Java microservices architecture? What are the benefits and potential challenges of using this pattern?**

Answer:

The Circuit Breaker pattern helps manage failures in a distributed system by preventing requests to a failing service and allowing it to recover. It transitions between three states: Closed, Open, and Half-Open.

### Implementation in Java Microservices:

- Closed State: Normal operation; requests are passed through.
- Open State: Requests are blocked when failures exceed a threshold; fallback responses are used.

- Half-Open State: Allows a limited number of requests to check if the service has recovered.

#### Java Libraries:

- Resilience4j: Provides a `CircuitBreaker` class for managing state transitions and configuring failure thresholds.
- Hystrix (Deprecated): Offers similar functionality for circuit breaking and fallback mechanisms.

#### Benefits:

- Fault Tolerance: Prevents cascading failures by isolating faults.
- Resilience: Enhances system stability by handling failures gracefully.
- Recovery: Allows services to recover and resume normal operation after a failure.

#### Challenges:

- Complexity: Adds overhead in managing circuit breaker states and integrating with services.
- Configuration: Requires careful tuning of thresholds and timeout settings.
- Fallback Management: Needs implementation of fallback logic to handle failures gracefully.

#### Example:

In a Java microservice, use Resilience4j to configure a circuit breaker for an external payment service. If the payment service fails repeatedly, the circuit breaker opens and prevents further requests until the service recovers, thus protecting the system from overloading the failing service.

---

### **333. Explain Orchestration vs Choreography in Microservices?**

Answer:

#### **1. Orchestration = One Central Brain**

There's a main service that tells everyone else what to do and when.

##### **Example:**

Let's say a customer places an order.

OrderService does this:

1. Calls PaymentService -> "Take payment"
2. Then calls InventoryService -> "Reduce stock"
3. Then calls ShippingService -> "Ship the item"
4. Finally calls NotificationService -> "Send confirmation"

The whole workflow is controlled in one place.

Like A project manager giving tasks to the team.

##### **Pros:**

Easy to understand

Easier to debug and retry failures

##### **Cons:**

Becomes a bottleneck

Tight coupling – if services change, the orchestrator must update too

## 2. Choreography = No Central Control

There's no boss. Each service just does its part when the time comes.

Example:

1. OrderService says: "OrderPlaced" (event)
2. PaymentService listens and says: "PaymentCompleted"
3. InventoryService hears that and says: "StockReduced"
4. ShippingService hears and ships
5. NotificationService hears and emails the user

Everyone is reacting to events like dancers in a performance.

Like each person knows their move when the music plays.

Pros:

Very loosely coupled

Easy to add new steps without changing others.

Cons:

Harder to trace the full process

Debugging can get tricky when things go wrong

Which to Choose?

Use Orchestration if you want:

Full control

Simpler debugging

Centralized logic

Use Choreography if you want:

---

Scalability  
Loose coupling  
Event-driven flexibility

---

### **334. Can you explain the Saga pattern and its implementation in a Java microservices architecture? What are its benefits and challenges?**

**Answer:**

The Saga pattern manages distributed transactions by breaking them into smaller, compensatable steps. Each step has a corresponding compensating action in case of failure.

#### Implementation in Java:

- Choreography: Services communicate directly and manage their own transactions.
- Orchestration: A central coordinator manages the sequence and compensations.

#### Benefits:

- Consistency: Ensures eventual consistency.
- Resilience: Handles failures with compensating actions.

#### Challenges:

- Complexity: Increased complexity in managing and coordinating transactions.
- Error Handling: Requires robust error and compensation handling.

#### Example:

In an e-commerce system, a saga coordinates order

creation, payment processing, and shipping, with compensations for any failures.

---

### **335. What is the 12-Factor App methodology, and how can it be applied to Java microservices? What are the benefits and challenges?**

**Answer:**

The 12-Factor App methodology outlines best practices for building scalable and maintainable applications.

#### **Application to Java Microservices:**

1. Codebase: Use a single codebase per service.
2. Dependencies: Manage with Maven or Gradle.
3. Config: Store in environment variables.
4. Backing Services: Treat as replaceable resources.
5. Build, Release, Run: Separate stages using CI/CD.
6. Processes: Design services to be stateless.
7. Port Binding: Expose services on specific ports.
8. Concurrency: Scale by running multiple instances.
9. Disposability: Ensure quick start and stop.
10. Dev/Prod Parity: Use Docker to maintain environment consistency.
11. Logs: Handle as event streams.
12. Admin Processes: Manage as separate tasks.

#### **Benefits:**

- Scalability: Facilitates scaling and deployment.
- Consistency: Ensures environment consistency.

- Resilience: Supports quick recovery.

**Challenges:**

- Complexity: Managing dependencies and configurations.
  - Consistency: Keeping dev and prod environments similar.
  - Logging: Requires robust solutions for log management.
- 

**336. You have a microservices architecture where multiple services rely on synchronized state information, but you're seeing discrepancies in data across services. How would you ensure consistent state synchronization?**

Answer:

Implement an event-driven architecture with a message broker to ensure all services receive consistent updates. Use techniques like eventual consistency or the Saga pattern to handle distributed state changes. Consider using distributed caches or a shared data store to maintain consistency.

---

**337. A sudden and unexpected load spike is causing some of your microservices to fail under pressure. How would you address this problem while ensuring minimal disruption?**

Answer:

Use autoscaling to dynamically adjust resources in

response to load spikes.

Implement rate limiting and request throttling to manage traffic.

Review and optimize the service's performance to handle increased load more efficiently.

Ensure proper load balancing across instances.

---

**338. You need to deploy a new version of a microservice without disrupting the existing users. How would you manage service versioning and ensure a smooth transition?**

Answer:

Use semantic versioning and implement versioned APIs to handle multiple versions simultaneously.

Consider deploying the new version alongside the old one (blue-green deployment) and use feature flags or canary releases to gradually switch traffic.

Ensure backward compatibility where possible and communicate changes to clients.

---

**339. You need to deploy a new version of a microservice without disrupting the existing users. How would you manage service versioning and ensure a smooth transition?**

Answer:

Use this approach for deploying a new microservice

version:

### Versioning Strategy:

Use Semantic Versioning (SemVer) to indicate compatibility levels.

### Deployment Techniques:

API Gateway: Route traffic to different versions based on rules.

#### Blue-Green Deployment:

Setup: Two identical environments.

- Deploy: New version to one, keep old active.
- Switch: Redirect traffic when ready, easy rollback if needed.

#### Canary Releases:

- Start with a small percentage of traffic to new version, gradually increase.

#### Feature Flags:

- Enable/disable new features or versions without redeployment.

### Service Registration:

Service Registry: Use tools like Eureka or Consul for dynamic service discovery.

### Database Changes:

Ensure schema changes are backward compatible or use dual writes temporarily.

### Monitoring:

Real-time Insights: Use tools like Prometheus for monitoring.

**Alerting:** Set thresholds for alerts on new version performance.

**Rollback Plan:**

Always have a plan to revert to the previous stable version.

**Workflow:**

1. New Version Deployment: Deploy to a staging environment.
2. Validation: Test in production-like conditions.
3. Traffic Routing: Start routing minimal traffic or enable via feature flags.
4. Monitor: Observe performance and stability.
5. Transition: Gradually increase traffic or fully switch after validation.

This approach ensures a controlled rollout with minimal disruption, allowing for quick adjustments or rollbacks if issues arise.

---

**340. You have a Java application that performs a transaction across two different databases (DB1 and DB2). Explain how you would ensure data consistency in 2PC?**

**Answer:**

Use 2Phase Commit, the transaction coordinator performs two phases:

Phase 1 (Prepare): The coordinator asks both DB1 and DB2 to prepare for a commit. Both databases lock the necessary resources but do not commit yet.

Phase 2 (Commit): If all databases respond with a "ready" status, the coordinator sends a commit command. Otherwise, it sends a rollback command.

---

**341. Your microservices architecture implements OAuth 2.0 for authentication, with each service handling its own security. However, you're facing challenges when propagating the OAuth token through internal service-to-service calls, leading to unauthorized requests. How would you redesign the security architecture to properly propagate authentication tokens across internal services?**

Answer:

Take this approach to redesigning your microservices security for better token propagation:

**1. API Gateway Token Validation**

Action: Implement an API Gateway to validate OAuth tokens once at the entry point.

Benefit: Reduces token validation load on individual services.

**2. JWT for User Identity**

Action: Use JWT for user authentication. Services decode and verify JWTs independently.

Benefit: Stateless, reduces network calls for token validation.

### 3. Service Tokens

Action: Each service gets its own OAuth token using Client Credentials Grant for service-to-service communication.

Benefit: Keeps user tokens separate from service communication tokens.

### 4. Token Forwarding

Action: Forward user tokens in headers or body for traceability across services.

Benefit: Maintains user context throughout service calls.

### 5. Use of mTLS

Action: Implement mutual TLS for service-to-service calls instead of token forwarding.

Benefit: Adds another layer of security, reducing token handling.

Combine these strategies based on your specific needs. For instance, use the API Gateway for initial token validation, JWT for user identity, and implement service tokens or mTLS for internal communications. This hybrid approach enhances security and simplifies token management across services.

---

**342. In a microservices system where services may have different levels of criticality (e.g., critical payment processing vs. some analytics), how would you implement a resiliency strategy that provides different**

## **levels of fault tolerance and recovery times based on the importance of the service?**

Answer:

To implement a resiliency strategy that accounts for varying service criticality, I would adopt differentiated resiliency patterns:

Service Categorization: First, categorize services by criticality: high-criticality (e.g., payment services), medium-criticality (e.g., user authentication), and low-criticality (e.g., analytics).

SLA & SLO: Define specific Service Level Agreements (SLAs) and Service Level Objectives (SLOs) for each service category. For critical services like payment processing, the SLA should require high availability and low latency, while analytics can tolerate longer downtimes.

Redundancy & Replication: Critical services need data replication across regions and active-active setups to ensure zero data loss, while low-critical services can opt for simpler active-passive or cold standby strategies.

Circuit Breaker Configuration: Utilize tools like Resilience4j to define circuit breakers with different thresholds. For critical services, the thresholds (e.g., failure rate) would be stricter, allowing for quicker fallback, while less critical services can have lenient settings.

Recovery Times: Implement graceful degradation for non-critical services (like switching to cached data) and ensure instant recovery mechanisms (e.g., auto-scaling, warm

standby) for critical services.

Chaos Engineering: Perform chaos testing (using tools like Chaos Monkey) in production to simulate failures and ensure that your resiliency strategy for each service works as expected.

---

**343. In a microservices architecture with hundreds of services, how would you approach service orchestration versus service choreography? What are the pros and cons of each approach, and in what scenarios would you prefer one over the other?**

Answer:

**Service Orchestration:** In orchestration, a central orchestrator (usually a service or controller) manages the interactions between different microservices. It is responsible for calling each service in a defined sequence to achieve a business outcome.

Pros:

Centralized control over the workflow.

Easier to manage and monitor workflows.

Simplifies error handling and rollback mechanisms.

Cons:

Can become a single point of failure.

Difficult to scale as the number of services grows.

Tight coupling between the orchestrator and services.

**Service Choreography:** In choreography, each service is responsible for its own actions and communicates with other services via events. No central authority is

responsible for orchestrating the process.

Pros:

Decentralized, allowing services to operate more autonomously.

Highly scalable as each service operates independently.

Greater flexibility and loose coupling between services.

Cons:

Complex to manage as it relies heavily on distributed event-based communication.

Debugging issues is harder as there's no central point of control.

Requires careful design to avoid cyclic dependencies between services.

**Scenario Preferences:**

I would prefer orchestration in workflows that are well-defined and require strict control, such as payment transactions or order management, where the sequence of operations is critical.

Choreography is better suited for systems that demand high scalability and flexibility, such as event-driven architectures where services evolve independently. For example, user notifications or analytics pipelines can leverage choreography.

---

**344. When implementing distributed tracing in a microservices system, how would you handle the challenge of tracing requests that span multiple services and potentially involve long-running operations? What tools and patterns would you use to collect, aggregate, and analyze trace data for end-to-end visibility across the system?**

Answer:

Handling distributed tracing for multiple services, especially long-running operations, requires a combination of tracing patterns and tools:

Unique Trace IDs: For each request, generate a unique trace ID that travels with the request as it flows across different services. This ensures that each service can log and report its portion of the request, which can later be stitched together to form an end-to-end trace.

Span and Segment Tracking: Use tools like OpenTelemetry or Jaeger to track spans (a unit of work) within each service. Spans are related to the trace ID, which allows us to visualize the journey of the request across services.

Eventual Consistency: In long-running operations, especially in asynchronous processes, traces may be incomplete for some time. By enabling eventual consistency within the tracing system, the full trace will eventually show the entire request's path once all operations complete.

Instrumentation Libraries: Use libraries provided by Zipkin, Jaeger, or OpenTelemetry for automatically collecting trace data. These tools integrate well with most microservice platforms like Spring Boot or Node.js.

Centralized Aggregation: Aggregating traces from multiple services into a centralized system such as Elastic Stack

(ELK), Datadog, or Jaeger ensures that all trace data is collected and analyzed in one place. This allows for identifying bottlenecks, latency issues, and failures across the entire system.

**Pattern: Correlation Context:** Use correlation context patterns to capture additional metadata like request headers, user session IDs, or transaction IDs, which help in diagnosing issues related to specific user requests.

---

**345. In a microservices-based architecture, you have implemented a circuit breaker using Hystrix or Resilience4j. During a failure in the downstream service, the circuit breaker does not open, resulting in cascading failures across services. How would you debug and resolve this issue?**

Answer:

To debug and resolve a circuit breaker that doesn't open:

**Check Configuration:** Ensure that the failure thresholds (e.g., failure rate, timeout, or error count) are correctly configured. If thresholds are too high, the circuit breaker won't open in time to prevent cascading failures. Adjust the thresholds based on service latency and error rates.

**Failure Detection:** Use logs and monitoring tools (e.g., Prometheus, Grafana) to check if errors in the downstream service are being recognized by the circuit breaker. It's possible that some failure modes (e.g., timeouts) aren't being captured correctly.

Fallback Implementation: Ensure that a proper fallback method is implemented and activated when the circuit breaker opens. In some cases, fallback mechanisms are not defined or aren't properly linked to the breaker's open state.

Timeouts and Retries: Often, timeouts are misconfigured, allowing downstream services to fail after a prolonged time, which may prevent the breaker from tripping. Reduce the timeout values and limit the number of retries in downstream calls.

Logging and Alerts: Enable detailed logging for the circuit breaker and set up alerts in monitoring tools to detect when a service is about to fail. These logs will help in identifying the cause of the breaker not opening.

---

### **346. In a scenario where Microservice A calls B calls C, how do you ensure atomicity?**

Answer:

Ensuring atomicity in a scenario where Microservice A calls B, which in turn calls C, is challenging because traditional database transactions don't span across multiple services in a distributed system. Here's how you can achieve atomicity or a state close to it:

#### **1. Saga Pattern**

The Saga pattern is used for long-running transactions that span multiple services. There are two primary approaches:

#### **Choreography:**

Each service produces and listens to events. When a service completes its part, it publishes an event for the next

service in the chain to act upon.

If any step fails, each service must implement a compensating transaction to undo its actions. This ensures atomicity through event-driven rollback.

**Example:**

- A calls B (sends event)
- B processes, then calls C (sends event)
- C processes. If successful, it's done. If C fails, C sends a compensating event, B listens, undoes its action, then sends another compensating event back to A, which can then roll back its initial action.

**Orchestration:**

An orchestrator service manages the entire transaction, calling each service in sequence and handling rollbacks if any part fails.

The orchestrator keeps track of the transaction state, deciding when to commit or rollback.

**Example:**

- The orchestrator calls A, then B, then C.
- If C fails, the orchestrator instructs B to reverse its changes, then A to do the same.

## 2. Two-Phase Commit (2PC)

While less common due to its complexity and potential for blocking, 2PC can be used:

**Prepare Phase:** Each service prepares for the transaction, locking necessary resources but not committing changes.

**Commit Phase:** If all services are ready, the coordinator sends a commit message; otherwise, it sends a rollback message.

However, 2PC in microservices can lead to performance issues and is not as scalable or fault-tolerant as other methods.

## 3. Distributed Transactions with XA

If your microservices share a database or can use XA transactions (which are supported by some databases), you can leverage XA for distributed transactions.

XA transactions ensure that all participating resources commit or rollback together, but this approach often comes with significant performance overhead and might not be suitable for all scenarios.

#### **4. Event Sourcing**

Each service records all state changes as events in an event store.

If a failure occurs, you can rebuild the state by replaying events up to the point of failure and then apply compensating events or commands to reverse the transaction.

#### **5. Idempotency and Retry Mechanisms**

Ensure operations are idempotent so that retrying doesn't cause unintended side effects.

Combined with retry logic, this can help recover from transient failures, though it doesn't directly ensure atomicity.

#### **Implementation Considerations:**

**Database:** Use databases that support in-memory transactions for each service to maintain local consistency.

**Eventual Consistency:** In microservices, you often deal with eventual consistency rather than immediate consistency. The key is to manage this process to mimic atomicity from the user's perspective.

**Logging and Monitoring:** To manage sagas or compensations, robust logging and monitoring are crucial for tracking transaction states and debugging issues.

**Timeout and Dead Letter Queues:** Implement timeouts for transaction steps and use dead letter queues for handling messages that couldn't be processed, allowing for

manual or automatic retry.

While achieving true atomicity across microservices is complex, these patterns and practices help simulate atomic behavior, ensuring data integrity and consistency across distributed systems.

---

**347. How would you track how many times a specific endpoint has been accessed in a microservice?**

Answer:

To track endpoint access counts, I would implement the following solutions:

Using Middleware/Interceptor: In frameworks like Spring Boot, I would create an HTTP filter or interceptor that logs each request made to the specific endpoint and increments a counter. This can be stored in in-memory counters like Redis, or simply logged for later aggregation.

Prometheus Metrics: Using Prometheus, I would expose a custom metric (e.g., http\_requests\_total) for each endpoint. These metrics can then be scraped by Prometheus and displayed in Grafana dashboards for real-time monitoring.

Distributed Log Analysis: For more advanced tracking, tools like ELK stack (Elastic, Logstash, Kibana) or AWS CloudWatch Logs can be used to capture request logs and aggregate the access count by filtering based on the endpoint.

Database Logging: For persistent tracking, I would store

each access log (with timestamp and user details) in a relational or NoSQL database. This allows for historical analysis and reporting.

By employing these strategies, you can maintain real-time insights into how often specific endpoints are accessed, allowing for deeper analytics and resource optimization.

---

**348. How would you manage schema evolution in a microservices system where each service owns a different part of the data model, ensuring that changes to one service's schema do not break other services? What strategies would you use to handle backward compatibility during schema updates, such as versioning or contract testing?**

**Answer:**

Schema evolution is a critical concern in microservices because each service typically owns its own data model, and changes to one service's schema should not affect other services. Here are strategies to manage schema evolution:

**Versioning:** Introduce versioned APIs and schemas. Each schema update introduces a new version (e.g., /v1/users and /v2/users). The older version remains operational until all dependent services migrate to the new version. This ensures backward compatibility.

**Backward Compatibility:** Maintain backward-compatible changes whenever possible. This means making non-breaking changes like adding optional fields rather than removing or modifying existing ones. Deprecated fields

should be retained for a set period to allow services time to adjust.

**Consumer-Driven Contracts (CDC):** Use contract testing frameworks like Pact to ensure that changes to a service's schema do not break its consumers. CDC allows consumers of a service to define expectations, which the provider can validate before deploying schema changes.

**Schema Registry:** In systems using event-based communication (e.g., Kafka), leverage a schema registry (e.g., Confluent Schema Registry) to validate schema changes. The schema registry ensures that new messages comply with existing consumers' schemas, preventing compatibility issues.

**Database Migrations:** For database changes, use tools like Liquibase or Flyway for database migrations. Ensure the database schema can support multiple versions during transitions by using strategies like expand-contract, where new columns are added and old ones are removed only after all services no longer rely on them.

---

**349. How would you ensure secure communication between microservices in a zero-trust network, where no service is inherently trusted? What methods or protocols would you use to authenticate and authorize communication between services? How would you handle key management, certificate rotation, and the encryption of data in transit?**

**Answer:**

In a zero-trust environment, securing communication between microservices is paramount. Here's how I would

approach this:

**Mutual TLS (mTLS):** Implement mutual TLS to ensure that both the client and the server authenticate each other. This requires the services to have certificates issued by a trusted Certificate Authority (CA). With mTLS, both the communication channel and the identities of the services are secured.

**Service Mesh:** Use a service mesh like Istio or Linkerd to enforce mTLS, manage authentication, authorization, and encryption between services without changing application code. A service mesh abstracts communication security into a separate layer, making it easier to manage at scale.

**OAuth2 & JWT:** For authorization, I would implement OAuth2 with JSON Web Tokens (JWT) to issue access tokens that services validate before communicating with other services. The JWT would carry claims (e.g., service roles) that are verified by downstream services to ensure authorized communication.

**Key Management:** Use a centralized key management system such as AWS KMS, Azure Key Vault, or HashiCorp Vault to store and manage keys, including TLS certificates. This ensures proper key rotation and auditing.

**Certificate Rotation:** Automate certificate rotation using tools like Certbot (for Let's Encrypt) or HashiCorp Vault's PKI secrets engine. This can be orchestrated using a service mesh or dedicated CI/CD pipelines to ensure certificates are updated without downtime.

**Data Encryption:** All data in transit between services should be encrypted using TLS to prevent eavesdropping and man-in-the-middle attacks. Additionally, ensure sensitive data is encrypted at rest using symmetric encryption (AES) managed by a key management service.

---

### **350. How do you handle shared state between microservices that cannot be stateless, like session management?**

#### **Answer:**

Handling shared state, particularly session management, between microservices requires careful design to avoid coupling and ensure scalability:

**Externalized Session Store:** Store session data in an external distributed cache like Redis, Memcached, or Hazelcast. This makes the session data accessible to all microservices without tying the session state to a particular service instance, ensuring scalability and fault tolerance.

**Stateless Tokens (JWT):** Instead of managing session state centrally, issue stateless tokens like JWT (JSON Web Tokens) for user authentication and session management. These tokens carry user claims and session data, allowing services to validate and process requests without querying a session store.

**Sticky Sessions:** In cases where session data cannot be externalized, implement sticky sessions via load balancers. This ensures that all requests from a user are routed to the

same instance of a service. However, this approach can limit scalability and resilience, as the failure of a service instance could result in lost sessions.

**CQRS for Shared State:** For complex shared states, employ the CQRS (Command Query Responsibility Segregation) pattern, where one service handles commands (modifying the state), while other services handle queries (reading the state). This can be combined with event sourcing to ensure that all services have a consistent view of the shared state.

---

**351. In microservices, deployments happen frequently, and it's common for a new deployment to introduce issues such as bugs, performance degradation, or integration problems. When this happens, rolling back to a previous stable version is critical.**

**How would you design a rollback strategy for microservices in case a deployment fails?**

**Answer:**

Designing a rollback strategy for microservices requires robust deployment practices to minimize downtime and reduce the risk of failed deployments:

**Canary Deployments:** Implement canary releases, where new versions of a service are deployed to a small subset of instances or users. Monitor the performance and behavior of the canary before rolling out the update to the entire system. If issues arise, quickly revert the canary to the previous stable version.

**Blue-Green Deployment:** Use blue-green deployments, where two identical production environments are maintained (blue and green). The new version is deployed to the blue environment while the green continues serving traffic. If the new version fails, traffic is quickly switched back to the green environment.

**Immutable Infrastructure:** Use immutable deployments where the new version of a service is deployed as a completely new instance (or container) while the old version remains intact. If the new deployment fails, simply stop the new instances and route traffic back to the old instances.

**Database Rollbacks:** Database migrations should support rolling back changes in a non-disruptive manner. Use tools like Flyway or Liquibase to apply reversible migrations. For destructive changes (e.g., dropping columns), ensure data snapshots or backups are in place to restore the database to its previous state if needed.

**Feature Toggles:** Use feature toggles (also known as feature flags) to control new functionality. If a new deployment introduces a bug, you can simply toggle off the new features without rolling back the entire service.

---

**352. In a microservices architecture where services communicate asynchronously through a message broker like Kafka, handling idempotency is critical to prevent issues like duplicate message processing, which can occur due to retries, network failures, or**

**replays. What strategies would you implement to ensure idempotency in microservices that consume messages from a message broker like Kafka? How would you design your services to handle scenarios where the same message might be processed multiple times due to retries or failures?**

**Answer:**

To ensure idempotency and prevent duplicate message processing in microservices consuming messages from a message broker like Kafka, I would adopt the following strategies:

**Idempotent Consumers:** Design services to be idempotent, meaning processing the same message multiple times will not lead to different outcomes. For example, ensure that if a message results in updating a record in the database, the update is only applied if there is a state change.

**Message Deduplication:** Implement message deduplication by attaching a unique message ID or event ID to each message. Store the ID in a database or cache (e.g., Redis) and check for its existence before processing the message. If the message ID is already processed, skip further processing.

**Exactly-Once Semantics:** Kafka provides exactly-once semantics (EOS), which guarantees that a message will be processed only once even in the event of retries or failures. Ensure that producers and consumers are configured for idempotent writes and use Kafka transactions to avoid reprocessing messages after a failure.

**Event Sourcing:** For critical operations, use event sourcing where every state change is captured as an event and stored in an event store. This allows for reprocessing and auditing of events, while ensuring that each event is applied only once, based on its unique ID.

**Retry and Dead Letter Queues:** Implement retry mechanisms with exponential backoff to handle transient failures and avoid overwhelming the system. If a message cannot be processed after multiple retries, move it to a Dead Letter Queue (DLQ). This isolates problematic messages for manual inspection and prevents them from causing further issues in the system.

**Database Atomicity with Kafka:** For operations involving both Kafka and a database, use a transactional outbox pattern. In this pattern, events are first written to the database as part of the same transaction that modifies the application's state. A separate process then reads these events and publishes them to Kafka, ensuring atomicity between the database and Kafka operations. By combining idempotency, message deduplication, and exactly-once semantics, you can ensure that.

---

### 353. What are Refresh Tokens?

**Answer:**

When you log in with OAuth2/OIDC, you usually get:

Access Token -> short-lived (e.g. 5–15 mins), used to call APIs.

Refresh Token -> long-lived (e.g. hours or days), used to get a new access token without asking the user to log in again.

### Why not just make Access Tokens long-lived?

Security -> If a token leaks, a short expiry limits the damage.

### How Refresh Token works :

1. You sign in -> Auth server gives you Access Token (15m) + Refresh Token (1h).
2. Your app calls an API with the Access Token.
3. When Access Token expires, your app silently sends the Refresh Token to the /token endpoint of Auth server.
4. AUTH server validates it and issues a new Access Token (and sometimes a new Refresh Token).
5. User stays logged in without retyping credentials.

### Security concerns:

Refresh Token should never go to the frontend/browser if you can avoid it. Store it safely (server-side or secure storage).

If a Refresh Token is leaked, attacker can mint new Access Tokens.

Use Refresh Token Rotation -> every time you use it, the Auth Server issues a brand-new one and invalidates the old.

### When to use them:

Mobile apps and SPAs that can't force the user to log in frequently.

Long-lived sessions (some apps, enterprise dashboards).  
The answer is Rate Limiting

---

### **354. You said you have retry logic. What if the service is down , will it keep retrying forever?**

#### **Answer:**

This question checks if you understand the risks of retry logic because retries can easily become DDOS-on-yourself if done wrong.

#### Best practices for retries:

1. Retry only on transient failures  
E.g., network timeout, 5xx errors  
Not for 4xx errors (like 404, 401)

2. Use exponential backoff

Increase delay between retries: 1s -> 2s -> 4s...

3. Set max retries & timeout

Avoid infinite retries or retry storms.

---

### **355. You said your microservice is stateless. What does that mean, and why is it important?**

#### **Answer:**

This is a common trap question in backend interviews. Many candidates just say: "It doesn't store session" but don't explain why that matters.

#### Let's break it down:

What does "stateless" actually mean?

A stateless service doesn't store any info about previous requests.

Every request is treated like it's coming from scratch , no memory of "what happened before."

#### Why is that useful?

1. Easier to scale

You can add more instances without worrying about where a user's session is stored.

2. Any instance can serve any request

No need to "stick" users to the same server.

3. Better fault tolerance

If one server crashes, no session data is lost , another can take over instantly.

#### Example:

Imagine you're building a login system.

#### Stateful service:

Stores session data (like userId) in memory..

### Stateless service:

After login, you send a JWT token to the client.  
On every future request, the client sends this token.  
No session data is stored on the server.  
Any server can verify the token and serve the request.

### So a Good answer could be something like this:

“Stateless means the server doesn’t store data between requests. Each request has everything needed to process it, like a token. This helps us scale easily, avoid sticky sessions.”

---

## **356. You mentioned your service is fast , Do you mean it has high throughput or low latency?**

### **Answer:**

This is a classic backend interview trap.  
Many developers use the terms interchangeably, but they mean very different things.

#### 1. Latency

The time it takes to get a response after making a request.  
Think: "How fast is one request?"

#### Example:

You click a button -> the server responds in 200ms -  
>that's the latency.

#### 2. Throughput

How many requests the system can handle in a given time.  
Think: "How many requests per second?"

#### Example:

---

If your system handles 1000 requests every second, that's your throughput.

Here's how to think about it:

A system can have low latency but low throughput , maybe it responds quickly but can't handle many requests at once.

Or it can have high throughput but high latency, it handles a large number of requests, but each takes a while to complete.

So, a strong answer for the question could be something like this :

“We reduced latency by optimizing DB queries and caching frequently accessed data. Then, to increase throughput, we horizontally scaled our service and added rate limiting to avoid overload.”

---

---

### **357. How do we prevent a single client from overwhelming our system with too many requests?**

**Answer:**

The answer is Rate Limiting.

Rate limiting is a technique used to control the number of requests a client can make to a system within a specific time window.

Example: "A user can only make 100 API calls per

minute."

### It helps with:

Protecting against abuse / DDoS attacks

Ensuring fair usage among clients

Maintaining system stability

## Common Algorithms

### 1. Fixed Window

Simple: Count requests in a fixed time window.

Issue: Traffic spikes at boundary edges.

### 2. Sliding Window

Uses moving time intervals.

Smoother than fixed window.

### 3. Token Bucket

Tokens added at a fixed rate.

A request needs a token to proceed.

Allows bursts but enforces overall rate.

### 4. Leaky Bucket

Requests leak out at a constant rate.

Smooths traffic, avoids bursty load.

## Where It's Used :

APIs : Prevent abuse (e.g., GitHub, Twitter APIs).

Login Systems: Prevent brute force attacks.

Payment Systems: Avoid double charges.

Microservices : Protect downstream services.

---

---

# **Chapter 20: Java design patterns & principles**

If you are getting started with Design Patterns, you should probably start with the important ones.

Six design patterns that are most likely to be coded and discussed during interviews:

## **1. Factory Pattern**

Purpose: The Factory Pattern provides an interface for creating objects but allows subclasses to alter the type of objects that will be created.

This pattern is particularly useful when the exact type of object to create isn't known until runtime.

### **Common Use Cases:**

- Creating objects that share a common interface but have different implementations.
- Simplifying object creation when multiple constructors or complex initialization is required.

### **Example Scenario:**

Creating different types of notifications (e.g., Email, SMS) using a factory method to instantiate the appropriate class based on user input.

### **Interview Focus:**

- Understanding how to implement the factory

method.

- Differentiating between Factory Method and Abstract Factory patterns.
- 

## 358. Explain Factory Pattern with a n example .

Answer:

Ever seen code full of new SomeClass() everywhere?  
That's tightly coupled, hard to test, and painful to maintain.  
The Factory Pattern solves this.

### Bad Code (No Factory Pattern):

```
class OrderService {  
    void placeOrder() {  
        Tea tea = new Tea(); // Directly creating Tea instance  
        tea.prepare(); // Calling its method  
    }  
}
```

Now imagine **5+ classes** doing this:

OrderService, BillingService, DeliveryService,  
KitchenService, ManagerDashboard —

each one writing new Tea() or new Coffee().

This leads to **tight coupling**, making the code hard to change, test, or extend.

### Why Is This Bad?

-Want to change Tea to GreenTea?  
You must update all classes manually.

---

- Want to add logging on object creation?  
You must add logging code in every class.
- Want to test with mock drinks?  
You cannot easily inject mocks or stubs.

### What is the Factory Pattern?

The Factory Pattern is a **creational design pattern** that **abstracts the object creation logic**. Instead of clients calling new directly, they request objects from a factory. This decouples object creation from usage.

#### Example:

```
interface Beverage {  
    void prepare();  
}  
  
class Tea implements Beverage {  
    public void prepare() {  
        System.out.println("Preparing Tea");  
    }  
}  
  
class Coffee implements Beverage {  
    public void prepare() {  
        System.out.println("Preparing Coffee");  
    }  
}  
  
class BeverageFactory {  
    public Beverage getBeverage(String type) {  
        if ("tea".equalsIgnoreCase(type)) return new Tea();  
        if ("coffee".equalsIgnoreCase(type)) return new Coffee();  
        throw new IllegalArgumentException("Unknown beverage  
type");  
    }  
}
```

---

}

### Client Code:

```
BeverageFactory factory = new BeverageFactory();
Beverage drink = factory.getBeverage("tea");
drink.prepare();
```

### Advantages of Using the Factory Pattern:

1. Centralized Object Creation All new logic is maintained in one place, simplifying changes.
  2. Loose Coupling Clients depend on interfaces or abstract classes, not concrete implementations.
  3. Open for Extension, Closed for Modification (OCP) You can add new beverage types without modifying client code.
  4. Easier Testing You can inject mocks or stubs by substituting factory implementations.
  5. Encapsulates Complex Logic The factory can handle creation conditions, configurations, or caching internally
- 

## **359. When is Abstract Factory preferred over Factory pattern?**

Answer:

Prefer Abstract Factory over Factory Method in Java when:

Product Families: You need to create entire sets of related objects that must be consistent, like different GUI toolkits for various platforms.

Interoperability: Objects within a family need to work

together or share specific traits.

Flexibility: Anticipating future additions or changes to product lines without altering existing client code.

Complex Hierarchies: Handling complex object relationships where maintaining consistency across creation is crucial.

In essence, use Abstract Factory for complexity involving multiple related object types and Factory Method for simpler, singular object creation.

---

## **2. Singleton Pattern**

The most favourite design pattern of Interviewers!

Purpose: The Singleton Pattern ensures that a class has only one instance and provides a global point of access to that instance. This pattern is used to control access to a single instance of a class.

Common Use Cases:

- Managing resources such as database connections or thread pools.
- Implementing a configuration manager with a single set of configurations.

Example Scenario:

Designing a logging service where only one instance should handle all log entries.

Interview Focus:

- Implementing the Singleton pattern with lazy

initialization and thread safety.

- Discussing potential issues with Singleton, such as difficulties in unit testing.

Singleton is one of the most favourite topics of the interviewers , be prepared with these questions:

---

### **360. What is double-check locking in Singleton?**

**Answer:**

Double-check locking is a technique to ensure thread-safe, lazy initialization of a Singleton instance while reducing synchronization overhead.

**Steps:**

1. First Check: If the instance is `null` , proceed to the synchronized block.
2. Synchronized Block: Inside, check again if the instance is still `null` .
3. Instance Creation: If `null` , create the instance.

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

---

## **361. How to resolve issues with reflection and cloning in Singleton?**

Answer:

### Prevent Reflection:

Throw an exception in the constructor if the instance already exists.

```
private Singleton() {  
    if (instance != null) {  
        throw new IllegalStateException("Instance already  
created!");  
    }  
}
```

### Prevent Cloning:

Override the `clone()` method to throw `CloneNotSupportedException`:

```
@Override  
protected Object clone() throws  
CloneNotSupportedException {  
    throw new CloneNotSupportedException();  
}
```

---

## **362. Which is the best way to create a Singleton: Enum or Bill Pugh Singleton?**

Answer:

### Enum Singleton (Better for simplicity and safety)

- **Serialization and Reflection Safe:** Enum handles serialization and prevents reflection-based attacks automatically.
- **Thread-Safety:** Enum ensures thread safety inherently.
- **Simplicity:** Requires minimal code and no extra handling for edge cases.

### Bill Pugh Singleton (Better for flexibility)

- **Lazy Initialization:** Singleton is created only when requested, which Enum doesn't offer.
- **Custom Initialization:** You can customize instance creation with more control compared to Enum.

#### So you should:

- Use Enum if you need a simple, secure Singleton without lazy initialization.
  - Use Bill Pugh if you need lazy initialization or customization.
- 

## **3. Observer Pattern**

Purpose: The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

This pattern is useful for creating a subscription mechanism to allow multiple observers to listen to changes in a subject.

### Common Use Cases:

- Implementing event handling systems.
- Notifying multiple components of state changes in an application.

### Example Scenario:

Creating a user interface where multiple views need to be updated when a data model changes.

### Interview Focus:

- Implementing the Observer pattern with decoupled subject and observer classes.
  - Handling scenarios with multiple observers and potential performance considerations.
- 

## **4. Builder Pattern**

Purpose: The Builder Pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

This pattern is used to construct an object step by step and provides control over the object's construction process.

### Common Use Cases:

- Creating complex objects with many optional components.
- Constructing objects where the construction process is independent of the parts that make up the object.

### Example Scenario:

Building a complex configuration object for a web application, where various settings and options can be specified.

### Interview Focus:

- Implementing the Builder pattern to provide a fluent API for object creation.
  - Discussing immutability and how to handle optional parameters.
- 

## **363. Can you explain the difference between the Builder Pattern and the Factory Pattern?**

Answer:

The Builder Pattern is for constructing complex objects step by step, especially when there are multiple optional parameters. It avoids the problem of constructors with many arguments.

### Example:

StringBuilder, where you append data and call `toString()` to get the final result.

The Factory Pattern creates objects in one step, hiding the creation logic. It's commonly used when you need to decide which object to create at runtime, based on certain conditions.

### Example:

`Calendar.getInstance()`, which can return different calendar types based on the locale.

### Key Differences:

**Builder Pattern:** Used for complex object construction. It builds the object incrementally and offers flexibility with optional parameters. It allows chaining methods and ends with a `build()` method.

**Factory Pattern:** Used for simple object creation where the type of object may vary. It creates the object in one step and returns it directly.

---

## **5. Strategy Pattern**

Purpose: The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern allows the algorithm to vary independently from the clients that use it.

### Common Use Cases:

- Implementing different sorting algorithms.
- Providing different behaviors for an algorithm based on runtime conditions.

### Example Scenario:

Designing a payment processing system where different payment strategies (e.g., credit card, PayPal) can be chosen dynamically.

### Interview Focus:

- Implementing the Strategy pattern with interchangeable algorithms.
  - Discussing how to use the pattern to avoid conditionals and improve flexibility.
- 

### **364. Give an example where Strategy pattern is used, could be JDK or your own?**

Answer:

Yes, one of the most common examples of the Strategy Pattern in the JDK is the use of the Comparator interface.

#### Example: Comparator in Java

In Java, the Comparator interface allows you to define multiple strategies for comparing objects. The sorting algorithm remains the same (for example, Collections.sort() or Arrays.sort()), but the comparison strategy can vary based on the Comparator implementation.

```
class Employee {  
    String name;  
    int age;  
    Employee(String name, int age) { this.name = name;  
        this.age = age; }  
    public String toString() { return name + ": " + age; }  
}  
  
class NameComparator implements  
    Comparator<Employee> {  
    public int compare(Employee e1, Employee e2) {  
        return e1.name.compareTo(e2.name);  
    }  
}
```

```

        }
    }

class AgeComparator implements
Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        return Integer.compare(e1.age, e2.age);
    }
}

public class StrategyExample {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(new
Employee("Alice", 30), new Employee("Bob", 25));

        Collections.sort(employees, new
NameComparator());
        System.out.println("Sorted by name: " +
employees);

        Collections.sort(employees, new
AgeComparator());
        System.out.println("Sorted by age: " + employees);
    }
}

```

Here, NameComparator and AgeComparator are strategies used by Collections.sort()

## **6. Decorator Pattern**

Purpose: The Decorator Pattern attaches additional responsibilities to an object dynamically. This pattern provides a flexible alternative to subclassing for extending

functionality.

#### Common Use Cases:

- Adding responsibilities to objects at runtime.
- Enhancing or modifying the behavior of objects in a scalable way.

#### Example Scenario:

Designing a user interface with various layers of decorations (e.g., borders, scroll bars) applied to components.

#### Interview Focus:

- Implementing the Decorator pattern with abstract components and concrete decorators.
  - Discussing how the pattern can be used to adhere to the Open/Closed Principle.
- 

## **365. How would you implement logging in Java using Decorators?**

Answer:

Here's an implementation of logging using the Decorator Pattern in Java:

Interface:

```
interface Processor {  
    void process(String data);  
}
```

Concrete Component:

```
class SimpleProcessor implements Processor {  
    @Override public void process(String data) {
```

```
        System.out.println("Processing: " + data);
    }
}
```

## Decorator:

```
class LoggingDecorator implements Processor {
    private final Processor wrapped;

    public LoggingDecorator(Processor p) {
        this.wrapped = p;
    }

    @Override
    public void process(String data) {
        System.out.println("Logging before processing: " +
data);
        wrapped.process(data);
        System.out.println("Logging after processing: " + data);
    }
}
```

## Usage:

```
Processor simple = new SimpleProcessor();
Processor logged = new LoggingDecorator(new
SimpleProcessor());

simple.process("data"); // No logging
logged.process("data"); // With logging
```

This setup allows adding logging to any Processor without changing its code, showcasing the Decorator Pattern's flexibility in extending behavior at runtime.

---

## 366. Name few design patterns used in JDK?

Answer:

Here are a few design patterns used in JDK:

1. Singleton Pattern - Used in Runtime, Desktop, System, Logger classes.
  2. Factory Method Pattern - Used in classes like Calendar, NumberFormat, and ResourceBundle.
  3. Builder Pattern - Used in classes like StringBuilder, StringBuffer, and java.lang.ProcessBuilder.
  4. Observer Pattern - Used in java.util.Observer and java.util.Observable (though deprecated, still represents the pattern).
  5. Decorator Pattern - Used in [java.io](#) package with classes like BufferedReader, BufferedWriter, and InputStreamReader.
  6. Proxy Pattern - Used in java.lang.reflect.Proxy to create dynamic proxy classes.
- 

### 367. What design principle(s) is below code breaking ?

```
interface Trader {  
    void executeTrade();  
    void auditTrade();  
    void manageRisk();  
}
```

```
class FrontOfficeTrader implements Trader {
```

```
public void executeTrade() {  
    // does trade  
}  
  
public void auditTrade() {  
    throw new UnsupportedOperationException();  
}  
public void manageRisk() {  
    throw new UnsupportedOperationException();  
}  
}
```

**Answer:**

The code breaks the **Interface Segregation Principle (ISP)** from the SOLID design principles.

**Explanation:**

Interface Segregation Principle (ISP) states that clients should not be forced to depend on methods they do not use.

In this example, FrontOfficeTrader implements the Trader interface, which requires implementing executeTrade(), auditTrade(), and manageRisk().

However, FrontOfficeTrader only supports executeTrade(), and for the other two methods it throws UnsupportedOperationException.

This indicates that the interface is too broad and forces the class to implement irrelevant methods, violating ISP.

**Summary:**

**Broken Principle:** Interface Segregation Principle (ISP)

**Problem:** Forcing implementation of unused methods leads

to poor design and potential runtime errors.

---

## 368. What is Dependency Inversion Principle?

Answer:

**Dependency Inversion Principle (DIP)** means high-level modules shouldn't depend on low-level modules. Both should depend on abstractions (interfaces).

Ignoring DIP leads to tight coupling and fragile code.  
It promotes flexible, maintainable, and testable software.

### Example: Notification System

Violates DIP — NotificationManager tightly coupled to SMSService

```
class SMSService {  
    void sendSMS(String message) {  
        System.out.println("Sending SMS: " + message);  
    }  
}  
  
class NotificationManager {  
    private SMSService smsService = new SMSService();  
  
    void notifyUser(String message) {  
        smsService.sendSMS(message);  
    }  
}
```

### Why this is bad:

- NotificationManager depends on the Notifier interface, not concrete classes.

- Easily add new notification methods without changing existing code.
- Code is cleaner, more flexible, and easier to test.
- DIP enables scalable and maintainable systems by depending on abstractions, not concrete details.

### Better design following DIP:

```
interface Notifier {
```

```
    void send(String message);
```

```
}
```

```
class SMSService implements Notifier {
```

```
    public void send(String message) {
```

```
        System.out.println("Sending SMS: " + message);
```

```
}
```

```
}
```

```
class EmailService implements Notifier {
```

```
    public void send(String message) {
```

```
        System.out.println("Sending Email: " + message);
```

```
}
```

```
}
```

```
class NotificationManager {
```

```
    private Notifier notifier;
```

```
    public NotificationManager(Notifier notifier) {
```

```
        this.notifier = notifier;
```

```
}
```

```
    void notifyUser(String message) {
```

```
        notifier.send(message);
```

```
}
```

```
}
```

### Benefits:

---

NotificationManager depends on the Notifier interface, not on concrete implementations.

New notification methods can be added without modifying existing code, adhering to the Open/Closed Principle.

The codebase becomes cleaner, more flexible, and easier to test.

DIP promotes building scalable and maintainable systems by relying on abstractions rather than concrete details.

---

### 369. What design principle(s) below class is breaking?

```
class FeeCalculator {  
    double calculateFee(Object instrument) {  
        if (instrument instanceof Stock) return 1.5;  
        else if (instrument instanceof Bond) return 2.0;  
        else return 0;  
    }  
}
```

Answer:

The class FeeCalculator is breaking the following design principles:

#### Open/Closed Principle (OCP)

The class is **not open for extension** because adding a new instrument type requires modifying the existing calculateFee method.

It violates OCP because you must change existing code to support new instrument types, rather than extending behavior.

---

## Single Responsibility Principle (SRP) (to some extent)

The method mixes logic for fee calculation for different instrument types, which could be separated into their own classes.

---

## **370. Explain Liskov's substitution principle in SOLID?**

Answer:

Liskov Substitution Principle (LSP) ensures that subclasses can be used in place of their parent classes without altering the correctness of the program.

Violation leads to:

Unexpected runtime issues

Fragile and misleading designs

Example of Violation:

```
class BankAccount {  
    void withdraw(double amount) {  
        // logic to withdraw money  
    }  
}
```

```
class FixedDepositAccount extends BankAccount {  
    @Override  
    void withdraw(double amount) {  
        throw new UnsupportedOperationException("Withdrawals  
not allowed from Fixed Deposit");  
    }  
}
```

## Why this violates Liskov's Substitution Principle:

If client code expects any BankAccount to support withdraw(), replacing it with FixedDepositAccount will break the logic.

This violates LSP, because the subclass (FixedDepositAccount) changes the expected behavior of the base class (BankAccount).

In banking, not all accounts support withdrawal. For example, FixedDepositAccount shouldn't override a withdraw() method just to throw an exception. Doing so violates LSP. A better design is to separate WithdrawableAccount and ensure only the appropriate subclasses extend it.

## Better Design Using Separation:

```
class BankAccount {  
    // common methods like deposit(), viewBalance()  
}
```

```
class WithdrawableAccount extends BankAccount {  
    void withdraw(double amount) {  
        // allowed to withdraw  
    }  
}
```

```
class SavingsAccount extends WithdrawableAccount {}  
class CurrentAccount extends WithdrawableAccount {}
```

```
class FixedDepositAccount extends BankAccount {  
    // no withdraw method  
}
```

Don't force subclasses to implement behaviors they can't support. Design hierarchies to preserve behavior consistency and support robust polymorphism.

---

## 371. What is Eventual Consistency?

Answer:

It means:

"The system isn't consistent right now but will be... eventually."

In distributed systems, when data is stored on multiple nodes or replicas, they don't sync instantly. But given time (and no new writes), all of them will eventually agree.

Example:

You update your profile picture:

You see it instantly

Your friend sees the old one for 2 more seconds

Eventually, everyone sees the new one.

That's eventual consistency ( temporary delay, permanent correctness.)

Why does it happen?

To stay highly available and fast, systems allow some lag in syncing data across nodes.

Instead of blocking the system for full consistency, it returns fast and syncs in the background.

---

### Where is it used?

NoSQL DBs like DynamoDB, Cassandra, Riak  
Distributed systems that prioritize availability over strict consistency (CAP Theorem)

Eventual Consistency is perfect for apps where speed matters more than instant consistency like chats, analytics, or social feeds.

---

## **372. What is AOP (Aspect-Oriented Programming)?**

Answer:

AOP is a programming paradigm that helps you separate cross-cutting concerns , logic that is spread across multiple parts of an application but is not the core business logic.

### Why it exists :

In large applications, some functionalities (like logging, security checks, transaction management, performance monitoring) get repeated in many classes. Without AOP, this leads to scattered, duplicated code and makes maintenance harder.

### How AOP works:

Instead of writing that logic everywhere, AOP lets you define it once in a separate unit called an Aspect, and then “weave” it into your application at specific points during execution.

---

# **Chapter 21: Some common Database Questions**

Database is a big topic in itself , but in this book I will share the Most asked DB interview questions to Java developers .

If you experience in NoSQL , be prepared with a detailed answer for **SQL vs NoSQL** Question.

## **373. What is the Difference Between Clustered and Unclustered Indexes?**

Answer:

Very commonly asked interview Question.

A clustered index determines the physical order of data in a table, meaning the rows are stored on disk in the same order as the index.

An unclustered (or non-clustered) index, on the other hand, does not affect the physical order of data. Instead, it creates a separate structure that points to the location of the rows in the table.

---

## **374. What is the difference between SQL and NoSQL Databases?**

Answer:

Feature	SQL (Relational Databases)	NoSQL Databases
<b>Data Model</b>	<b>Structured</b> , table-based with rows and columns. Data is normalized to reduce redundancy.	<b>Unstructured or Semi-structured</b> , can be document-based, key-value, column-family, or graph.
<b>Schema</b>	<b>Fixed Schema</b> : Requires a predefined schema that applies to all data within a table.	<b>Flexible or Schema-less</b> : Allows for dynamic schemas; documents in a collection can vary in structure.
<b>Scalability</b>	<b>Vertical Scaling</b> (scale up with more powerful hardware). Harder to scale horizontally.	<b>Horizontal Scaling</b> (scale out across commodity servers). Designed for easier horizontal scaling.
<b>ACID Compliance</b>	<b>Strong ACID Compliance</b> : Ensures Atomicity, Consistency, Isolation, Durability.	<b>Varies</b> : Some NoSQL databases offer ACID guarantees for certain operations, others prioritize availability and partition tolerance (CAP theorem).
<b>Query Language</b>	<b>SQL</b> (Structured Query Language) for complex queries, joins, and transactions.	<b>Varies</b> : Often use custom query languages, APIs, or even SQL-like syntax in some systems; less support for complex queries like joins across datasets.
<b>Use Cases</b>	<b>Transactional Systems</b> , complex queries, where data integrity is paramount (e.g., banking).	<b>Big Data</b> , real-time web applications, content management systems where flexibility and speed are needed over strict consistency.

<b>Examples</b>	MySQL, PostgreSQL, Oracle, SQL Server.	MongoDB (Document), Cassandra (Column-family), Redis (Key-value), Neo4j (Graph).
<b>Data Consistency</b>	<b>High Consistency:</b> Transactions ensure data consistency.	<b>Eventual Consistency</b> or configurable consistency levels; some support strong consistency.
<b>Setup and Maintenance</b>	<b>More Complex:</b> Requires DBA for schema design, normalization, and maintenance.	<b>Often Simpler:</b> Less need for schema management, but can require more application-level logic for data integrity.
<b>Cost</b>	Can be <b>Expensive</b> due to licensing (for proprietary databases) and hardware for scaling.	<b>Potentially Cheaper:</b> Many are open-source, and commodity hardware can be used for scaling.
<b>Performance</b>	Good for <b>read-heavy operations</b> with complex queries. Write operations can be slower due to locking mechanisms.	<b>High performance for read/write operations,</b> especially for large volumes of simple data operations.

#### Key Points:

- **SQL databases** are ideal for applications where data consistency, structure, and complex querying are crucial.
- **NoSQL databases** shine in scenarios where data can be denormalized, where you have varying data models, or where you need to handle massive scale with less emphasis on immediate consistency.

## 375. What are different types of Statements in JDBC?

**Answer:**

Here are the main types of JDBC Statements:

Statement:

Basic SQL execution without parameters. Good for simple queries.

PreparedStatement:

Precompiled SQL statements, ideal for performance with repeated executions and parameter usage.

CallableStatement:

For executing stored procedures, handling more complex database interactions.

---

## **376. What is a primary key and a foreign key? How do they differ?**

**Answer:**

Primary Key: A unique identifier for each record in a table. It must contain unique values and cannot be NULL.

Foreign Key: A field (or group of fields) in one table that refers to the primary key in another table, establishing a relationship between the two tables. A foreign key can accept duplicate values and can be NULL.

---

## **377. What is ACID in Database?**

**Answer:**

Atomicity: Transactions are all-or-nothing; if one part fails, the entire transaction fails.

Consistency: Transactions bring the database from one valid state to another, maintaining all predefined rules.

Isolation: Transactions occur independently, ensuring that concurrent transactions do not affect each other.

Durability: Once a transaction is committed, it remains so, even in the event of a system failure.

---

## 378. What are materialized views?

Answer:

Materialized views are database objects that store the results of a query as a physical table.

Materialized views are used when you need to precompute and store the results of complex queries to improve performance, especially for frequently run, resource-intensive queries.

Use Case:

Data Warehousing and Reporting Running complex aggregation queries over large datasets for reporting is slow.

Solution: Use materialized views to precompute and store aggregate data.

Example:

Create a materialized view to precompute total sales per day

```
CREATE MATERIALIZED VIEW daily_sales AS SELECT  
order_date, SUM(total_amount) AS total_sales FROM orders  
GROUP BY order_date;
```

Benefit:

Querying daily\_sales is faster than recalculating each time.

---

## **379. What is the difference between a procedure and a function in a database?**

Answer:

**IMPORTANT QUESTION .**

Don't miss out on this one.

### **Procedure:**

Purpose: Executes operations without returning a value directly.

Use: For actions like updating data or performing complex sequences.

### **Function:**

Purpose: Computes and returns a single value or table.

Use: For calculations or returning results used in queries.

### **Differences:**

- Return Value: Functions must return something; procedures don't.
- Usage: Functions can be used in SQL expressions; procedures are called explicitly.
- Transaction: Functions generally shouldn't alter database state directly in most databases.

**Choose procedures for tasks, functions for computations.**

Feature	Procedure	Function
<b>Return Value</b>	Can return 0 or more values using output parameters or through <code>OUT</code> or <code>INOUT</code> parameters.	Must return a single value or table (in some SQL dialects like SQL Server).
<b>Usage in SQL Statements</b>	Cannot be used directly in SQL statements like <code>SELECT</code> , <code>WHERE</code> , <code>HAVING</code> .	Can be used in SQL statements ( <code>SELECT</code> , <code>WHERE</code> , <code>HAVING</code> , etc.) because they return a value.
<b>Purpose</b>	Often used for executing a series of SQL statements, performing operations, and managing transactions.	Primarily used for computations, returning values based on some logic or data manipulation.
<b>Call Syntax</b>	Invoked using <code>CALL</code> or <code>EXECUTE</code> statement (e.g., <code>CALL procedure_name(params);</code> ).	Can be called like any scalar or table-valued function in SQL statements ( <code>function_name(params)</code> ).
<b>Parameters</b>	Supports <code>IN</code> , <code>OUT</code> , and <code>INOUT</code> parameters for more complex data flow.	Generally supports only <code>IN</code> parameters, although some databases support <code>OUT</code> in specific contexts.
<b>Transaction Management</b>	Can manage transactions, commit or rollback changes.	Typically does not manage transactions; changes are part of the calling transaction.
<b>Result Set</b>	Can return multiple result sets or no result set at all.	Should return only one result set or a single value.

---

### 380. How would you find the employee with the Nth highest salary in a database?

**Answer:**

Don't miss out on this one as well.

Most interviewers just ask this query to just your SQL skills.

You can use a SQL query with subqueries to find the Nth highest salary. For example,

in SQL:

```
SELECT * FROM employees
WHERE salary = (
    SELECT DISTINCT salary
    FROM employees
    ORDER BY salary DESC
    LIMIT 1 OFFSET N-1
);
```

Replace `N` with the desired rank.

This query selects the employee with the Nth highest salary by ordering salaries in descending order and skipping the top N-1 salaries.

---

## **381. What is LEFT OUTER JOIN**

**Answer:**

A LEFT OUTER JOIN returns all rows from the left table and matching rows from the right table. If there's no match, the result will have NULL for columns from the right table.

Example:

```
SELECT Employees.name, Departments.dept_id  
FROM Employees  
LEFT OUTER JOIN Departments  
ON Employees.emp_id = Departments.emp_id;
```

---

## 382. What is the difference between **ROW\_NUMBER**, **RANK** and **DENSE\_RANK**?

Answer:

**ROW\_NUMBER()**: Assigns a unique sequential integer to each row, with no ties. E.g., 1, 2, 3.

**RANK()**: Gives the same rank to tied rows, but skips ranks. E.g., 1, 1, 3 if two rows are tied for first.

**DENSE\_RANK()**: Gives the same rank to tied rows, without skipping. E.g., 1, 1, 2 if two rows are tied for first.

Function	Description	Behavior	Example
<b>ROW_NUMBER()</b>	Assigns a unique number to each row within a result set, regardless of ties.	Numbers rows sequentially, starting from 1, without gaps.	If three rows tie for first place, they'll get 1, 2, 3.
<b>RANK()</b>	Assigns a rank to each row, where rows with equal values get the same rank.	Leaves gaps in ranking for tied values. For example, if two rows tie for first, the next rank would be 3, not 2.	If three rows tie for first, they all get 1, the next rank is 4.
<b>DENSE_RANK()</b>	Similar to <b>RANK()</b> , but doesn't leave gaps in the ranking sequence.	Rows with equal values get the same rank, but the next rank is consecutive.	If three rows tie for first, they all get 1, the next rank is 2.

## Example:

```
SELECT
    score,
    ROW_NUMBER() OVER (ORDER BY score DESC) AS row_num,
    RANK() OVER (ORDER BY score DESC) AS rank,
    DENSE_RANK() OVER (ORDER BY score DESC) AS dense_rank
FROM scores;
```

Result for scores [10, 10, 8, 8, 7, 7, 7]:

score	row_num	rank	dense_rank
10	1	1	1
10	2	1	1
8	3	3	2
8	4	3	2
7	5	5	3
7	6	5	3
7	7	5	3

In this example:

- `ROW_NUMBER()` gives each row a unique number.
- `RANK()` shows gaps for tied ranks.
- `DENSE_RANK()` does not skip ranks for ties.

---

## 383. Describe the CAP theorem and how it applies to distributed databases

Answer:

The CAP theorem states that a distributed system can achieve at most two of the following three properties:

- 1. Consistency**: All nodes see the same data at the same time.
- 2. Availability**: Every request gets a response, even if some nodes fail.
- 3. Partition Tolerance**: The system continues to operate despite network partitions.

#### **Application:**

CP Systems: Ensure consistency and partition tolerance but may sacrifice availability.

AP Systems: Ensure availability and partition tolerance but may sacrifice consistency.

---

## **384. What is connection pooling?**

Answer:

**Connection Pooling** refers to the practice of maintaining a pool of database connections that can be reused. This reduces the overhead of creating new connections for each database operation, which is particularly beneficial for applications that need to frequently access databases.

#### **Concept:**

- When an application starts or when the first connection is needed, a pool of connections is created.
- Connections are borrowed from this pool, used, and then returned rather than closed.
- This approach minimizes the time spent on

establishing connections, which can be costly in terms of performance.

---

### **385. How can you analyze and improve the performance of a slow SQL query?**

**Answer:**

Approach to improving a slow SQL query:

Identify:

Execution Plan: Use EXPLAIN to understand how the query runs.

Optimize:

Indexes: Ensure appropriate indexing on key columns.

Query Structure:

- Use joins over subqueries where possible.
- Avoid SELECT \*, limit columns.
- Add LIMIT if not all rows are needed.

Schema:

Partitioning: Split large tables if applicable.

Normalization: Adjust based on query needs.

Test & Iterate:

Performance Check: Compare before and after optimization.

Continuous Monitoring: Performance can change with data growth.

This methodical approach helps pinpoint and resolve performance bottlenecks in SQL queries.

---

### **386. Explain the differences between vertical**

## **and horizontal scaling in databases. When would you choose one over the other?**

Answer:

Vertical Scaling (Scaling Up): Involves adding more resources (CPU, RAM, storage) to a single machine. It's easier to implement but has limitations based on the hardware capabilities. It is often chosen for transactional databases where consistency and ACID properties are crucial.

Horizontal Scaling (Scaling Out): Involves adding more machines or nodes to distribute the load. This approach is more complex but provides better fault tolerance and can handle a higher volume of transactions. It's typically used in distributed databases or NoSQL systems to achieve scalability and availability.

---

### **387. How do you write a SQL query to find duplicate records in a table?**

Answer:

To find duplicate records, you can use the GROUP BY clause combined with the HAVING clause. For example, to find duplicates in the Employee table based on the email column:

```
SELECT email, COUNT(*) AS Count
FROM Employee
GROUP BY email
HAVING COUNT(*) > 1;
```

This query groups records by email and counts them, returning only those with a count greater than one.

---

## **388. What is a Common Table Expression (CTE), and how is it used?**

**Answer:**

A Common Table Expression (CTE) is a temporary result set defined within the execution of a single SELECT, INSERT, UPDATE, or DELETE statement. It can be referenced within the query. For example:

```
WITH EmployeeCTE AS (
    SELECT department, AVG(salary) AS AvgSalary
    FROM Employee
    GROUP BY department
)
SELECT department
FROM EmployeeCTE
WHERE AvgSalary > 60000;
```

This query first calculates the average salary for each department and then retrieves departments with an average salary greater than 60,000.

---

## **389. Explain the concept of window functions and provide an example.**

**Answer:**

Window functions perform calculations across a set of table rows related to the current row. Unlike aggregate functions, they do not reduce the number of rows returned. For example, using the ROW\_NUMBER() function to assign a unique sequential integer to each row within a

partition:

```
SELECT name, salary,  
       ROW_NUMBER() OVER (ORDER BY salary DESC)  
AS Rank  
FROM Employee;
```

This query assigns a rank to each employee based on their salary in descending order.

---

### **390. What are subqueries, and how do they differ from joins?**

Answer:

Subqueries are nested queries used to retrieve data that will be used in the main query. They can return a single value, a row, or a table. Unlike joins, which combine rows from two or more tables based on related columns, subqueries can be used to filter data or calculate aggregates that are then used in the main query. For example:

```
SELECT name  
FROM Employee  
WHERE department_id = (SELECT id FROM Department  
WHERE name = 'Sales');
```

This subquery retrieves employees from the Sales department by first finding the department's ID.

---

### **391. What is the difference between UNION and JOIN?**

Answer:

UNION: Combines the results of two or more SELECT statements into a single result set, removing duplicate rows. Each SELECT must have the same number of columns in the result sets.

JOIN: Combines rows from two or more tables based on a related column, returning rows that match the specified condition. For example:

```
SELECT E.name, D.department_name  
FROM Employee E  
JOIN Department D ON E.department_id = D.id;
```

This query retrieves employee names along with their respective department names.

---

## **392. How do you handle NULL values in SQL queries?**

Answer:

You can handle NULL values using the IS NULL and IS NOT NULL conditions. Additionally, functions like COALESCE() and NULLIF() are used to replace or compare NULL values. For example:

```
SELECT name, COALESCE(salary, 0) AS Salary  
FROM Employee;
```

This query replaces NULL salaries with 0.

---

## **393. What are aggregate functions, and can**

## **you provide examples of their use?**

Answer:

Aggregate functions perform calculations on a set of values and return a single value. Common aggregate functions include:

COUNT(): Counts the number of rows.

SUM(): Calculates the total of a numeric column.

AVG(): Computes the average value.

MIN(): Returns the smallest value.

MAX(): Returns the largest value.

For example, to find the total salary of all employees:

```
SELECT SUM(salary) AS TotalSalary  
FROM Employee;
```

---

## **394. What is database sharding and its benefits?**

Answer:

Sharding is a database architecture pattern where large datasets are divided into smaller, more manageable pieces called shards, each hosted on a separate database server. Benefits include:

Improved performance by distributing read/write loads.

Increased storage capacity as data can grow across multiple servers.

Enhanced availability and fault tolerance, as individual shards can fail without taking down the entire system.

---

### **395. Discuss the trade-offs between eventual consistency and strong consistency in distributed databases**

**Answer:**

Eventual Consistency: Guarantees that, given enough time without new updates, all replicas will converge to the same value. It allows for higher availability and performance, but it may lead to temporary inconsistencies, which can be problematic for certain applications (e.g., financial transactions).

Strong Consistency: Ensures that all reads return the most recent write. While it provides a straightforward programming model, it can limit availability and increase latency, especially in distributed systems where nodes may experience network delays.

---

### **396. How do you implement database versioning in microservices architecture?**

**Answer:**

Database versioning in microservices can be achieved through:

Schema Migration Tools: Tools like Flyway or Liquibase enable version control of database schemas, allowing each microservice to manage its own database schema and migrations independently.

API Versioning: Versioning APIs can help manage changes in the database schema while maintaining backward compatibility for consumers.

Backward-compatible Changes: Design schema changes (e.g., adding new columns or tables) to be backward-compatible so that old versions of the microservice can still operate with the updated database schema.

---

### **397. What is the concept of a distributed transaction and how it differs from a local transaction.**

Answer:

Local Transaction: Involves operations on a single database. It follows the ACID properties and is managed by a single transaction manager.

Distributed Transaction: Involves operations across multiple databases or services. It is managed using protocols like the Two-Phase Commit (2PC) or Atomic Commit, which coordinate the commit or rollback of transactions across distributed systems, ensuring consistency.

---

### **398. What strategies would you use to handle database schema changes in a production environment?**

Answer:

Strategies include:

Blue-Green Deployments: Maintain two identical environments, allowing for seamless transitions during schema updates.

Rolling Updates: Gradually deploy schema changes to avoid downtime, ensuring older versions can still interact with the database.

Feature Toggles: Use toggles to enable or disable new features that depend on schema changes, allowing for safe rollbacks if issues arise.

Backward Compatibility: Always design changes to be backward-compatible to avoid breaking existing functionality.

---

### **399. What is database denormalization, and when should it be applied?**

Answer:

Denormalization is the process of intentionally introducing redundancy into a database to improve read performance. It involves combining tables or adding redundant data.

It should be applied when:

The application requires complex read queries that would benefit from fewer joins.

Performance is more critical than storage efficiency.

The system is read-heavy, and the cost of maintaining data consistency is outweighed by the performance gains.

---

## **400. Explain the difference between sharding and replication in databases? When would you use each?**

Answer:

Both sharding and replication deal with scaling databases, but they solve different problems:

### **1. Sharding = Horizontal Partitioning**

Think of sharding as splitting data across multiple databases based on a key (e.g., user ID, region).

Each shard holds a subset of the data.

No shard has the full dataset.

Improves write scalability.

Reduces load on each DB node.

Use case:

When your data size exceeds what a single server can handle or when write throughput becomes a bottleneck.

Example:

A social media app shards user data by  $\text{user\_id} \bmod(4)$  across 4 databases.

### **2. Replication = Copying Data**

Replication means copying the same data to multiple servers.

One primary (master) for writes.

One or more replicas (slaves) for reads or backups.

Improves read scalability and provides fault tolerance.

### Use case:

When you have many read operations, or want high availability in case the primary DB fails.

### Example:

An e-commerce site uses one primary DB and 3 read replicas to serve product pages.

### NOTE:

Sharding is not the same as replication. In sharding, nodes have different data. In replication, nodes have the same data.

### You can combine both:

Sharded data, and  
Each shard is replicated  
for both scalability and availability.

---

## **401. What is a View and its limitations?**

### Answer:

A View in a database is a virtual table based on the result set of a SQL query.

It does not store data physically but provides a way to look at data from one or more tables.

Simplifies complex queries, provides data abstraction, enhances security by restricting access to specific columns or rows.

### Limitations of Views:

#### 1. No Physical Storage:

Views do not store data physically (except materialized views), so every time you query a view, the underlying query runs, which can impact performance.

#### 2. Performance Issues:

Complex views with joins and aggregations can be slow, especially if nested views are involved.

#### 3. Updatability Restrictions:

Not all views are updatable. Views based on multiple tables, aggregation functions, GROUP BY, DISTINCT, or UNION may not support INSERT, UPDATE, or DELETE operations directly.

#### 4. Dependency on Base Tables:

If underlying tables change (e.g., columns dropped), the view might become invalid or cause errors.

#### 5. Limited Use of Indexes:

Since views don't store data, indexes on views (except materialized views) are not possible, which can affect query optimization.

---

---

# **Chapter 22 : Common Java Stream Coding Problems**

**BEFORE GOING STRAIGHT INTO CODING PROBLEMS  
LET'S UNDERSTAND FEW STREAMS BASIC OPERATIONS:**

## **1. Filter Even Numbers**

**Problem:** Filter out even numbers from a list.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
List<Integer> evenNumbers = numbers.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());
```

**Explanation:** The filter operation applies a predicate to each element, only keeping those where the predicate returns true. Here,  $n \rightarrow n \% 2 == 0$  checks if a number is even. The result is collected into a new list.

## **2. Square Each Number**

**Problem:** Square each number in the list.

```
List<Integer> squaredNumbers = numbers.stream()
    .map(n -> n * n)
    .collect(Collectors.toList());
```

**Explanation:** map transforms each element of the stream by applying a function to each. Here, each number  $n$  is squared ( $n * n$ ).

## **3. Sum of All Numbers**

**Problem:** Calculate the sum of all numbers in the list.

```
int sum = numbers.stream()
    .reduce(0, (a, b) -> a + b);
```

**Explanation:** reduce combines all elements into a single result. Here, it starts with an identity value of 0 and adds each number, summing them up.

## 4. Count Even Numbers

**Problem:** Count how many even numbers are in the list.

```
long count = numbers.stream()
    .filter(n -> n % 2 == 0)
    .count();
```

**Explanation:** Similar to problem 1, but instead of collecting, we use count() to get the number of elements that pass the filter.

## 5. Check if All Numbers Are Positive

**Problem:** Check if every number in the list is positive.

```
boolean allPositive = numbers.stream()
    .allMatch(n -> n > 0);
```

**Explanation:** allMatch checks if all elements match the predicate. Here, it checks if all numbers are greater than zero.

## 6. Find Any Number Greater than 5

**Problem:** Find any number in the list that's greater than 5.

```
Optional<Integer> firstGreater Than Five = numbers.stream()
    .filter(n -> n > 5)
    .findAny();
```

**Explanation:** filter narrows down to numbers greater than 5, and findAny returns an Optional containing one of these numbers (or empty if none exist).

## 7. Group Numbers by Parity

**Problem:** Group numbers into even and odd groups.

```
Map<Boolean, List<Integer>> groupedByParity =  
    numbers.stream()  
        .collect(Collectors.groupingBy(n -> n % 2 == 0));
```

**Explanation:** groupingBy collects elements into a Map. The key is the result of the function  $n \rightarrow n \% 2 == 0$ , which groups numbers by whether they're even (true) or odd (false).

### LET'S JUMP INTO SOME CODING PROBLEMS NOW:

## 402. Using Java Streams *Find Even Numbers from a List*

Answer:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);  
  
List<Integer> evenNumbers = numbers.stream()  
    .filter(n -> n % 2 == 0)  
    .collect(Collectors.toList());  
  
System.out.println(evenNumbers); // [2, 4, 6]
```

## **403. Using Java Streams Merge Two Lists Using Streams and Without Streams**

Answer:

### Java Streams :

```
List<String> list1 = Arrays.asList("A", "B");
List<String> list2 = Arrays.asList("C", "D");

List<String> merged = Stream.concat(list1.stream(), list2.stream())
                           .collect(Collectors.toList());

System.out.println(merged); // [A, B, C, D]
```

### Explanation:

- Stream.concat() combines the streams of both lists.
- collect(Collectors.toList()) then gathers all elements into a new list.

### Without Using Java Streams:

```
List<String> list1 = new ArrayList<>(Arrays.asList("A", "B"));
List<String> list2 = Arrays.asList("C", "D");

list1.addAll(list2);

System.out.println(list1); // [A, B, C, D]
```

### Explanation:

We create a new ArrayList from list1 to avoid modifying the original list.

addAll(list2) appends all elements from list2 to

mergedListWithoutStreams.

### Key Points:

Stream Approach: More declarative, potentially more readable for complex stream operations, but might have slight performance overhead for simple concatenations.

Non-Stream Approach: More imperative, direct, and could be more performant for straightforward operations due to avoiding the overhead of stream creation and processing.

---

## **404. Using Java Streams *Find the First Non-Repeating Character in a String***

Answer:

```
String input = "swiss";  
  
Character result = input.chars()  
    .mapToObj(c -> (char) c)  
    .collect(Collectors.groupingBy(c -> c, LinkedHashMap::new, Collectors.counting()))  
    .entrySet().stream()  
    .filter(e -> e.getValue() == 1)  
    .map(Map.Entry::getKey)  
    .findFirst()  
    .orElse(null);  
  
System.out.println(result); // Output: w
```

### Explanation:

Character Counting: We use str.chars() to get a stream of integers representing the characters in the string. We then map these to char, count each character's occurrence using

groupingBy, and store the results in a LinkedHashMap to preserve the order of appearance.

Finding Non-Repeating: We then stream through the entries of our charCount map, filtering for those with a count of 1, which signifies a non-repeating character.

First Match: findFirst() is used to get the first such character. If no character matches (i.e., all characters repeat), orElse(null) returns null.

Main Method: Demonstrates how to use this method with a sample string.

---

#### **405. Using Java Streams *Sort a HashMap by Keys and Values***

Answer:

SORT BY KEY

```
public class SortHashMap {  
    public static void main(String[] args) {  
        // Create a sample HashMap  
        Map<String, Integer> unsortedMap = new HashMap<>();  
        unsortedMap.put("banana", 2);  
        unsortedMap.put("apple", 5);  
        unsortedMap.put("cherry", 1);  
  
        // Sort by keys  
        Map<String, Integer> sortedByKey = unsortedMap.entrySet()  
            .stream()  
            .sorted(Map.Entry.comparingByKey())  
            .collect(Collectors.toMap(  
                Map.Entry::getKey,  
                Map.Entry::getValue,  
                (e1, e2) -> e1,  
                LinkedHashMap::new  
            ));  
  
        // Print the sorted map by keys  
        System.out.println("Sorted by keys: " + sortedByKey);  
    }  
}
```

## SORT BY VALUE

```

public class SortHashMap {
    public static void main(String[] args) {
        // Create a sample HashMap
        Map<String, Integer> unsortedMap = new HashMap<>();
        unsortedMap.put("banana", 2);
        unsortedMap.put("apple", 5);
        unsortedMap.put("cherry", 1);

        // Sort by values
        Map<String, Integer> sortedByValue = unsortedMap.entrySet()
            .stream()
            .sorted(Map.Entry.comparingByValue())
            .collect(Collectors.toMap(
                Map.Entry::getKey,
                Map.Entry::getValue,
                (e1, e2) -> e1,
                LinkedHashMap::new
            ));

        // Print the sorted map by values
        System.out.println("Sorted by values: " + sortedByValue);
    }
}

```

### Explanation:

Stream Creation: entrySet().stream() converts the map entries into a stream.

### Sorting:

sorted(Map.Entry.comparingByKey()) sorts by keys.  
 sorted(Map.Entry.comparingByValue()) sorts by values.

Collection: collect(Collectors.toMap(...)) collects the sorted entries back into a Map. Here, we use LinkedHashMap to maintain the insertion order which

reflects the sorted order.

Map.Entry::getKey and Map.Entry::getValue are used to map the key and value back into the new map. $(e1, e2) \rightarrow e1$  is a merge function to handle duplicate keys, although in this case, there won't be any duplicates since we're dealing with unique keys from the original map.

LinkedHashMap::new ensures the map maintains the order of insertion, which is now sorted.

---

## 406. Using Java Streams *Group Words by Their First Character*

Answer:

```
public class GroupByFirstChar {  
    public static void main(String[] args) {  
        List<String> words = Arrays.asList("apple", "banana", "avocado", "blueberry", "cherry");  
  
        Map<Character, List<String>> grouped = words.stream()  
            .collect(Collectors.groupingBy(word -> word.charAt(0))); // group by first char  
  
        System.out.println(grouped);  
    }  
}
```

Explanation:

### Stream Creation:

We start with words.stream() to create a stream from the list of words.

### Grouping Operation:

Collectors.groupingBy() is used to group the elements by a classifier function. Here, the classifier function is a lambda expression word  $\rightarrow$  word.charAt(0) which returns the first

character of each word.

Result:

The result is a Map<Character, List<String>> where the keys are the first characters of the words, and the values are lists of words starting with those characters.

Output:

We then print each entry in the map, where key is the character and value is the list of words.

---

## 407. Using Java Streams *Find Top N Frequent Words in a Paragraph*

Answer:

```
public static void main(String[] args) {
    String paragraph = "the quick brown fox jumps over the lazy dog the fox was quick";
    int N = 3;

    Map<String, Long> wordFreq = Arrays.stream(paragraph.split("\s+"))
        .map(String::toLowerCase)
        .collect(Collectors.groupingBy(w -> w, Collectors.counting()));

    List<Map.Entry<String, Long>> topN = wordFreq.entrySet().stream()
        .sorted(Map.Entry.<String, Long>comparingByValue(Comparator.reverseOrder()))
        .limit(N)
        .collect(Collectors.toList());

    System.out.println("Top " + N + " words: " + topN);
```

Output:

java: 4  
spring: 2  
boot: 1

### Explanation:

Text Processing: Instead of using `\W+`, I've used `\s+` to split on whitespace since there's no punctuation in this paragraph. This splits the string into words based on spaces.

Lowercase Conversion: All words are converted to lowercase to ensure [case-insensitive counting](#).

Filtering: Removes any potential empty strings, though in this example, there won't be any due to the nature of the text.

Counting and Sorting: The rest of the process (grouping by word, counting occurrences, sorting by count in descending order, and limiting to the top 3) remains the same as in the previous example.

---

## 408. Using Java Streams *Find All Palindromic Strings in a List*

Answer:

```
public static void main(String[] args) {
    List<String> words = Arrays.asList("madam", "hello", "noon");

    List<String> palindromes = words.stream()
        .filter(w -> w.equalsIgnoreCase(new StringBuilder(w).reverse().toString()))
        .collect(Collectors.toList());

    System.out.println("Palindromes: " + palindromes);
}
```

# **Chapter 23 : Fifteen Methods that interviewers love**

Understanding key Java methods can help you showcase your expertise during interviews. Below are a few important methods commonly discussed:

## **1. Class.forName()**

Class.forName() loads a class dynamically at runtime using its fully qualified name. It is often used in scenarios where the class to be loaded is not known at compile time.

### Example:

```
Class<?> clazz = Class.forName("com.example.MyClass");
```

## **2. Thread.yield()**

### Explanation:

Thread.yield() is a static method that suggests the current thread to pause and allow other threads of the same priority to execute. It's a hint to the thread scheduler but does not guarantee that the current thread will pause.

### Example:

```
Thread.yield(); // Hint to the thread scheduler to give other threads a chance
```

## **3. String("").intern()**

### Explanation:

String.intern() returns a canonical representation of the

string object. Strings with the same content are stored in a common pool, reducing memory usage.

### Example:

```
String str1 = new String("example").intern();
String str2 = "example";
System.out.println(str1 == str2); // true, as both refer to the
                                same interned string
```

## 4. **map.entrySet()**

### Explanation:

Map.entrySet() returns a set view of the mappings contained in the map. It is useful for iterating over the key-value pairs in the map.

### Example:

```
Map<String, Integer> map = new HashMap<>();
map.put("a", 1);
map.put("b", 2);

for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println(entry.getKey() + ": " +
entry.getValue());
}
```

## 5. **object.wait()**

### Explanation:

Object.wait() causes the current thread to wait until another thread invokes notify() or notifyAll() on the same object. It is used in thread synchronization.

### Example:

```
synchronized (lockObject) {
```

```
        lockObject.wait(); // Thread waits until notified  
    }
```

## 6. **Thread.join()**

### Explanation:

Thread.join() allows one thread to wait for the completion of another thread. It ensures that the calling thread waits until the specified thread has finished execution.

### Example:

```
Thread thread = new Thread(() ->  
    System.out.println("Running"));  
thread.start();  
thread.join(); // Main thread waits for this thread to finish
```

## 7. **stream().flatMap()**

### Explanation:

stream().flatMap() is used to flatten a stream of collections into a single stream. It is useful for processing nested data structures.

### Example:

```
List<List<String>> listOfLists =  
    Arrays.asList(Arrays.asList("a", "b"), Arrays.asList("c",  
        "d"));  
List<String> flattenedList = listOfLists.stream()  
    .flatMap(List::stream)  
    .collect(Collectors.toList());
```

## 8. **Optional.ofNullable()**

### Explanation:

`Optional.ofNullable()` creates an `Optional` instance that may or may not contain a non-null value. It is used to avoid `NullPointerException` and handle optional values more gracefully.

Example:

```
Optional<String> optional = Optional.ofNullable("value");
optional.ifPresent(System.out::println); // Prints "value"
```

## 9. **Collections.synchronizedList()**

Explanation:

`Collections.synchronizedList()` returns a synchronized (thread-safe) list backed by the specified list. It helps ensure thread safety in multi-threaded environments.

Example:

```
List<String> syncList = Collections.synchronizedList(new
ArrayList<>());
syncList.add("item");
```

## 10. **Collections.unmodifiableList()**

Explanation:

`Collections.unmodifiableList()` returns an unmodifiable view of the specified list. It prevents modifications to the list and is used to create read-only lists.

Example:

```
List<String> list = Arrays.asList("a", "b", "c");
List<String> unmodifiableList =
Collections.unmodifiableList(list);
```

## **11. Stream.reduce()**

### Explanation:

Stream.reduce() performs a reduction on the elements of the stream using an associative accumulation function and returns an Optional describing the reduced value.

### Example:

```
int sum = Arrays.asList(1, 2, 3, 4).stream()  
    .reduce(0, Integer::sum);  
System.out.println(sum); // Prints "10"
```

## **12. AtomicInteger.incrementAndGet()**

### Explanation:

AtomicInteger.incrementAndGet() atomically increments the current value by one and returns the updated value. It is part of the java.util.concurrent.atomic package and ensures thread-safe operations.

### Example:

```
AtomicInteger atomicInt = new AtomicInteger(0);  
int incrementedValue = atomicInt.incrementAndGet();  
System.out.println(incrementedValue); // Prints "1"
```

## **13. Enum.valueOf()**

### Explanation:

Enum.valueOf() returns the enum constant of the specified enum type with the specified name. It's used to convert a string into an enum constant.

### Example:

```
Day day = Enum.valueOf(Day.class, "MONDAY");  
System.out.println(day); // Prints "MONDAY"
```

## **14. Object.clone()**

**Explanation:**

Object.clone() creates and returns a copy of the object. It requires the class to implement Cloneable and override the clone() method properly.

**Example:**

```
public class MyClass implements Cloneable {  
    @Override  
    protected Object clone() throws  
    CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

## **15. Thread.sleep()**

**Explanation:**

Thread.sleep() pauses the current thread for a specified number of milliseconds. It is useful for introducing delays or simulating time-consuming tasks.

**Example:**

```
try {  
    Thread.sleep(1000); // Sleeps for 1 second  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

---

---

# **Chapter 23: RESUME TIPS**

In a competitive job market, it's essential to not only possess the right technical skills but also to effectively showcase them in your CV.

This is how you should present in your CV.

## **TECHNICAL SKILLS**

### **1. JVM Analysis**

Proficiency in understanding and optimizing JVM performance is vital for any Java developer. This includes managing garbage collection, optimizing heap memory, and understanding thread management.

- Tools: VisualVM, JConsole, JProfiler.
- Key Areas: Garbage collection tuning, memory management, thread dumps analysis.

**CV Tip:** Highlight specific instances where you improved application performance by optimizing JVM settings.  
Example: “Reduced application latency by 20% through JVM garbage collection tuning.”

## 2. Concurrency

Expertise in multithreading and concurrent programming is essential for building high-performance Java applications. Mastering thread synchronization, managing thread pools, and utilizing modern concurrency utilities like CompletableFuture and ForkJoinPool are crucial.

- Key Concepts: Deadlock avoidance, thread safety, concurrency utilities (java.util.concurrent package).

CV Tip: Include examples of how you resolved concurrency issues or improved performance. Example: “Enhanced application throughput by 35% by optimizing multithreading using ForkJoinPool.”

## 3. Test-Driven Development (TDD)

A strong foundation in Test-Driven Development (TDD) using tools like JUnit and Mockito is critical. TDD ensures your code is reliable, maintainable, and free of bugs.

- Key Concepts: Unit testing, mocking, stubbing, behaviour-driven development (BDD) with Cucumber.

CV Tip: Demonstrate your TDD expertise by citing specific projects. Example: “Increased code coverage to 95% by implementing TDD practices with JUnit and Mockito.”

## 4. Spring Framework

A comprehensive understanding of the Spring Framework, particularly Spring Boot and Microservices, is indispensable. This includes developing RESTful services, leveraging Spring Data, and managing dependency injection.

- Spring Security/OAuth: Implement robust security measures using Spring Security, OAuth2, and JWT tokens.

CV Tip: Detail your experience in developing and securing Spring-based applications. Example: “Developed microservices with Spring Boot, securing APIs using OAuth2, leading to a 50% reduction in security vulnerabilities.”

## 5. Messaging Systems

Experience with messaging systems like Kafka and RabbitMQ is crucial for building scalable, event-driven architectures. Understanding how to configure producers, consumers, and manage message brokers is key.

- Key Concepts: Event-driven architecture, message serialization (Avro, Protobuf), idempotency.

CV Tip: Highlight your experience with messaging systems, especially in high-throughput environments. Example: “Implemented a Kafka-based messaging system,

processing over 10 million messages daily with near-zero latency.”

## 6. Distributed Caching

Proficiency in distributed caching solutions like Redis or Memcached helps optimize application performance by reducing load on the database and improving response times.

- Key Concepts: Cache-aside pattern, write-through and write-back caches, TTL (Time-To-Live) settings.

CV Tip: Showcase how you’ve utilized caching to improve application efficiency. Example: “Reduced database load by 40% through effective use of Redis for distributed caching.”

## 7. Continuous Integration/Continuous Deployment (CI/CD)

Hands-on experience with CI/CD pipelines using tools like Jenkins, Docker, and Kubernetes is vital for automating build processes, containerizing applications, and deploying to cloud environments.

- Key Concepts: Pipeline scripting, container orchestration, deployment automation.

**CV Tip:** Describe your role in setting up CI/CD pipelines and the impact on deployment speed and reliability.

Example: “Automated deployment process using Jenkins and Docker, reducing release cycle time by 50%.”

## 8. Alerts & Monitoring

Knowledge of monitoring and alerting tools such as Splunk, Grafana, and the ELK stack is crucial for maintaining the health and performance of your applications. This includes setting up dashboards, creating alerts, and analyzing logs.

- **Key Concepts:** Metrics collection, log aggregation, performance monitoring.

**CV Tip:** Include details on how you’ve used these tools to prevent or resolve incidents. Example: “Configured ELK stack for real-time monitoring, reducing incident response time by 30%.”

## OTHER THINGS TO ADD

### 1. Quantify Achievements

- Use metrics to highlight your impact. For example, “Improved application response time by 30% through JVM tuning and optimized garbage collection strategies.”
- Include details on the scale of systems you’ve worked on, such as “Managed a Kafka-based messaging system processing over 10 million messages daily.”

## 2. Performance Optimization

- Emphasize experience in performance tuning, such as “Optimized SQL queries, reducing database load by 25% and improving API response times by 40%.”
- Mention specific techniques or tools used, like “Utilized Redis for caching, reducing database read operations by 50%.”

## 3. Highlight Scalability

- Discuss your contributions to system scalability, e.g., “Architected microservices that scaled to handle 5x traffic during peak loads.”
- Detail your experience with distributed systems, load balancing, and cloud-based deployment.

## 4. Project-Specific Achievements

- Showcase your contributions to major projects, e.g., “Led the migration of a monolithic application to microservices, resulting in a 60% reduction in deployment time.”
- Mention your role in large-scale integrations, such as “Integrated Kafka with microservices to achieve real-time data processing with sub-second latency.”

## 5. Soft Skills and Collaboration

- Don’t forget to include collaboration and leadership skills, especially if you’ve worked in Agile teams, e.g., “Facilitated sprint planning and retrospectives, improving team productivity by 20%.”
- Mention any mentoring or training roles, like “Mentored junior developers on best practices in TDD and Spring Security.”

## 6. Certifications and Continuous Learning

- List relevant certifications, such as “Certified Spring Professional Developer” or “Oracle Certified Professional: Java SE Developer.”
- Mention ongoing education, e.g., “Currently pursuing advanced courses in Kubernetes and Cloud Native Development.”

Focus on quantifiable achievements, detailed technical skills, and contributions to significant projects to stand out in a competitive job market.

Remember, your CV is not just a summary of your experience but a reflection of your technical expertise and professional accomplishments.

P.S. No one cares about the ATS Score

---

# Chapter 24: REAL INTERVIEW REPORTS

## JAVA DEVELOPER 4 YEARS EXPERIENCED REAL INTERVIEW REPORT

### For Mandatory Skills

**Core Java** --> Good

**Knows:** Internals of Hashmap, Why Collection.synchronize(HashMap) vs ConcurrentHashMap() in terms of performance, Why to choose Comparable and Comparator, HashMap (String with it's occurrence problem) (Provided In\_Efficient solution with n\*n Time\_Complexity)

**Don't know:** --

**Java 8 (+ Features)** --> Good

**Knows:** Stream API (clear on how/ when to use intermediate, terminal operations etc.), map() vs flatMap(), Functional Interface (Basics)

**Don't know:** --

**SpringBoot/ Spring/ JPA** --> Above Average

**Knows:** How to change jar to war, @SpringBootApplication, Reading values from Application.properties, Different layers of SpringBoot and annotations, Basics of JPA/ Hibernate, How to write repository layer and Queries using JPA

**Don't know:** How to change Tomcat (not clear), Transaction management (Basics ONLY, In\_Depth missing), Exceptional handling (@ControllerAdvice), How to fix circular dependency

**REST API** --> Good

**Knows:** POST, PUT vs PATCH

**Don't know:** --

**Object Oriented Programming** --> Above Average

**Knows:** 4 pillars of OOP

**Don't know:** SOLID principle

**Design Patterns** --> Weak

**Knows:** Singleton criteria

**Don't know:** How to break Singleton, any other creational/ behavioural/ structural Design Patterns

### For Good To Have Skills

Data Structures: Above Average

Knows: Collection API

Don't know: Efficient approach is missing

Agile Scrum --> No Experience (Candidate confirmed)

### For Soft Skills

Communication --> Above Average

Confidence --> Above Average

Clarity of Thoughts --> Above Average

### Coding Feedback

Problem solving/ Coding --> Above Average

1. HashMap (String with it's occurrence) --> Correctly Answered (Verbally)

2. Stream API based problem --> Correctly Answered

3. SQL (Employees who are Managers) --> Correctly Answered (Approx.)

4. DSA --> Partially Answered

## **Questions asked**

- Could you please briefly introduce yourself?
- What's the domain of your current project?
- What type of Architects are using in your project?
- What is SOLID principle?
- Explain the difference between ConcurrentHashMap and synchronized HashMap in terms of performance.
- What is Collision in the context of a HashMap?
- How can you create a Spring Boot application from scratch?
- How can you change the packaging from JAR to WAR in a Spring Boot application?
- How can you switch from Tomcat server to Jetty server in a Spring Boot application?
- How can you handle exceptions globally in a Spring Boot application?
- How can you utilize transaction management in Spring Boot?
- What are isolation levels in transaction management and why are they important?
- How can you fix circular dependency in Spring Boot?
- How can you write a repository layer in JPA?
- How can you write a native query in JPA for selecting all records from an employee table?
- Why do we use @Qualifier annotation in Spring?
- Explain the differences between POST, PUT, and PATCH HTTP methods.
- How can you efficiently check if one string is a circular rotation of another string?
- Write a program to find all manager names from a single employee table based on the manager ID relationship.- Is P n equal to 100?
- How many loops are needed to solve this program efficiently?
- What approach would you use to solve this question?
- Are you familiar with agile methodology? If so, which methodologies have you worked on in your career?
- Have you worked with design patterns, specifically the Singleton pattern?

## **Summary of answers**

\*\*Topics:\*\*

### **1. \*\*Introduction and Current Project:\*\***

- Aditya Soundade introduced himself as working in the United industry and looking for better opportunities to work on new technologies.
- He explained his current project involving Global Link service, dealing with transaction messages, decryption, validation, and framework usage.

### **2. \*\*Technical Knowledge - Java and Spring Boot:\*\***

- Discussed SOLID principles related to class design and modification.
- Compared synchronized hashmap and concurrent hashmap performance in a multi-threaded environment.
- Explained collision concept in hashmaps.
- Shared insights on creating a Spring Boot application, changing packaging from JAR to WAR, and switching servers from Tomcat to Jetty.
- Mentioned using exception handlers in Spring Boot for global exception handling.
- Discussed transaction management in Spring Boot and isolation levels for payment systems.
- Addressed circular dependency issue and worked on JPA repository layer queries.
- Explained the usage of @Qualifier annotation and HTTP methods like POST, PUT, and PATCH.

### **3. \*\*Coding Problems:\*\***

- Developed pseudocode solutions for string manipulation problems and database query scenarios.
- Solved a problem involving checking if one string is a circular rotation of another.

### **4. \*\*Final Questions and Ratings:\*\***

- Answered questions related to managing managers in an employee table and determining circular rotations in strings.
- Rated himself 6 out of 10 in Spring Boot knowledge.\*\*Topics: Factorial Calculation\*\*

- Candidate discussed the calculation of factorial for different numbers, such as 5, 10, and 100.
- Mentioned about trailing zeros in factorial calculations.
- Discussed the approach to solve the program efficiently without using stream API.
- Questioned about the number of loops required to solve the program and its impact on time complexity.

**\*\*Topics: Agile Methodology\*\***

- Candidate was asked about their experience with agile methodology in the workplace.
- Confirmed that they are aware of agile methodology but did not provide detailed information.
- Mentioned working in organizations where daily status meetings were held to discuss tasks performed.
- Clarified that the current project with State Bank of India does not follow a specific methodology.

**\*\*Topics: Design Patterns\*\***

- Candidate mentioned Singleton design pattern.
- Asked about breaking the Singleton pattern and criteria for creating a Singleton class.
- Candidate provided some criteria for creating a Singleton class, such as private constructor, public factory method, and public static variable.
- Candidate expressed familiarity with Factory method but could not recall details about other creational design patterns.

#### **Coding Questions**

Question: Core Java --> Find string with it's occurrence.

I/P:

```
String[] names = {"Java", "Angular", "React", "NodeJS", "Azure", "NodeJS", "Java", "Angular", "NodeJS", "Angular"}
```

O/P:

Java	2
Angular	3
React	1
NodeJS	3
Azure	1

=====

Question: Java 8 (Stream API)

I/P:

```
String[] names = {"Java", "Angular", "React", "NodeJS", "Azure", "NodeJS", "Java", "Angular", "NodeJS", "Angular"}
```

Operation 1: Need to process only those names which starts with "A"

Operation 2: In front of every string write --> "I'll learn"

Operation 3: Accumulate the processed names and return it.

=====

Question: SQL: Employee Table:

EMPLOYEE_ID	NAME	PHONE_NUMBER	MANAGER_ID
1	"Sumit"	1345656654	3
2	"Vivek"	3453453454	4
3	"Arun"	4564654656	4
4	"Guru"	5658675676	3

=====

```
str1 = "ABACD"
str2 = "CDABA"
```

```
boolean areCircularRotation(string str1, string str2) {
    // logic
}
```

# JAVA DEVELOPER 5 YEARS EXPERIENCED REAL INTERVIEW REPORT

## **For Mandatory Skills**

Java 8: Answered for Optional class, Streams API vs Collections API, Difference between Findfirst and Findany

Not answered for Metaspace

Micro Services: Answered for making service Scalable, Circuit breaker design pattern, Service Discovery and Service Registry

Rest API: Answered for Put vs Post, http 401 vs 403, http 200

Design Patterns: The candidate answered for design patterns like Singleton, Factory patterns

## **For Good To Have Skills**

The candidate is above average in good to have skills

## **For Soft Skills**

The candidate has above average communication skills

## **Coding Feedback**

To write a java code to swap 2 count the occurrence of characters inside a given String. The candidate has coded correctly

## **Recommendation And Other Strengths**

Recommended( If candidate is not Proxy): Need confirmation from the Technical team to check for it. If candidate found non Proxy, he is a good candidate and can be selected for further evaluation rounds.

1. The candidate is good in all mandatory skill
2. Is above average in good to have skills
3. Coded well as per the requirement given

### **Questions asked**

- Which version of Java are you working with?
- Can you please share your screen for a coding round?
- Write a Java code to count the occurrence of each character in a given string.
- What is metaspace in Java?
- Explain optional classes and how they are used.
- What is the difference between streams API and collections API?
- Explain the difference between findFirst and findAny methods.
- How can microservices be scaled for scalability?
- What is the circuit breaker design pattern and how is it implemented?
- Explain the concept of service discovery and its importance.
- What is the difference between PUT and POST methods in HTTP?
- Differentiate between HTTP status codes 400 and 403.
- What is the difference between GET request and Gateway timeout error?
- Explain the differences between HTTP status codes 202 and 203.
- What is ACID property in databases?
- What is indexing in databases and how does it work?
- What is Zookeeper and what role does it play in Kafka?
- Explain the concept of brokers in Kafka.
- Can Kafka function without Zookeeper?
- What is the Singleton design pattern? If you defend any class which having the least single the glass would be having the single residuality to have the chance of cookie. having the one agent.
- What each Factory patterns?
- Actually the factory pattern it is if the fact that it allowed. electricity of the object creation. if you interactive pattern suppose for example of if you be having the
- For example support if you be heavy one interface and In one interview having that one action method. So apart from that, if you define that to Service classes classy implemented this sent me the best there will be liked it. We Dependable at the class

class. We get implemented interface our work classes all rights, same abstract method. So for our Wednesday be in what's the while? somebody support if you want to start for particular class so we can defend it we can pass it object of object class in it

### **Summary of answers**

**\*\*Topics:\*\***

**\*\*Project and Technical Skills:\*\***

- Working on a market rental project for truck drivers
- Implementing mobile APIs for travel apps
- Experience with Java, REST API, microservices
- Knowledge of meta space, optional classes
- Understanding of streams API and collections API
- Explaining findFirst and findAny methods
- Scaling microservices horizontally and vertically
- Circuit breaker design pattern implementation
- Service discovery and registry using Eureka server
- Difference between PUT and POST methods
- Handling HTTP status codes like 400 and 403
- Database experience with MySQL
- Understanding ACID properties and indexing in databases

**\*\*Kafka and Zookeeper:\*\***

- Knowledge of Kafka and Zookeeper
- Explanation of Zookeeper's role in handling brokers and topics
- Understanding brokers' role in managing messages and partitions
- Lack of clarity on some Kafka concepts like leader and necessity of Zookeeper

**\*\*Miscellaneous:\*\***

- Limited experience with Kafka (less than a year)
- Familiarity with design patterns like Singleton

**\*\*Candidate's Answers Summary:\*\***

**- \*\*Singleton Pattern:\*\***

- Defending a class with Singleton pattern.
- Achieving Singleton pattern using lazy initialization and eager initialization.
- Discussing the creation of instances in Singleton pattern.

# JAVA DEVELOPER 5 YEARS EXPERIENCED REAL INTERVIEW REPORT

## **For Mandatory Skills**

core java :

- data hiding and how we can achieve in java --yes
- custom immutable class with array list as a reference --yes
- JVM m/r internal architecture --yes
- initial capacity of array list n how it increase their size dynamically--partially yes
- internal working of hash set--yes
- use of atomic integer--yes
- constructor chaining with parameterized constructor in base class only--yes
- use of semaphore--yes
- predicate consumer n supplier in java 8 --yes
- why clone() is protected in java --yes

spring boot:

- spring security--JWT--yes
- spring bean validation--partially yes
- how we can configure two db in one spring boot project--yes
- bean factory n application context--yes
- can we use the singleton bean scope in multithreaded env--no

restapi:

- idempotent n non idempotent--no
- @pathvariable n @requestparam--yes

data structure:

- how we can find the circular linked List--yes
- linear n non linear data structure--yes

design pattern:

- use of builder design pattern--yes

- Tell me about yourself and the technologies you have worked on.
- Have you worked with data structures and algorithms?
- Explain data hiding and how it can be achieved in Java.
- How can you create a custom immutable class in Java?
- Can you explain JVM memory internal architecture?
- Where are static variables stored in memory?
- How does ArrayList internally work in terms of initial capacity and size increase?
- Explain how HashSet internally works.
- What is the return type of the put method in HashMap?
- When would you prefer using AtomicInteger?
- What is the output of a given program involving base and derived classes?
- Differentiate between Predicate, Consumer, and Supplier in Java 8.
- Explain the use of Semaphore to avoid deadlock conditions.
- Why is the clone method protected in Java?
- How can you achieve bean validation in Java?

- How can you configure two databases in a Spring Boot project?
- What are the differences between PathVariable and RequestParam?
- Explain idempotent methods in HTTP.
- Can Singleton bean scope be used in a multi-threaded environment?
- How can you identify a circular linked list?
- Differentiate between linear and non-linear data structures.
- Explain the Builder design pattern and its role in creational design patterns.

## JAVA DEVELOPER 4 YEARS EXPERIENCED REAL INTERVIEW REPORT

### **For Mandatory Skills**

Java 8: The candidate answered for Metaspace concepts, Streams API vs Collections API, Flatmap

Micro Services: Answered for Scaling attributes, Circuit breaker design pattern, Testing of micro services, Micro services intercommunication. Not used practically

Rest API: Answered for http 400 vs 500, http 202 vs 203, Http 401 vs 403

MySQL: The candidate answered for Indexing, Views

Kafka: Answered for Queuefull exception, Partition, zookeeper

Design Pattern: Answered for Factory design pattern

## **Questions asked**

- Introduce yourself quickly with technical terms
- Which version of Java are you currently working on?
- Why are you looking for a change after two years at Amdoc?
- What is a meta space in Java 8?
- How can the size of meta space be increased?
- Explain the difference between Streams API and Collections API.
- What is flat map in streams?
- Write a Java code to find maximum combinations of characters like ABC.
- How do microservices communicate?
- Explain the circuit breaker design pattern.
- What is a fallback method in microservices?
- How can you make microservices scalable?
- Differentiate between horizontal and vertical scaling.
- How do you test your microservices?
- Explain the differences between HTTP status codes 400 and 500.
- What is the difference between HTTP status codes 2002 and 2003?
- Define nonauthoritative information in HTTP status codes.
- Describe the usage of proxy servers in HTTP status codes.
- Explain the differences between HTTP status codes 4001 and 403.
- Define authorization and authentication in APIs.
- Provide an example of forbidden access in HTTP status codes.
- What is an index in a database table?- What are views?
- You have worked on Kafka.
- How do you handle it?
  
- What is partition?
- What is a zookeeper?
- What is cluster?
- And what all design patterns?
- What each Factory?

# JAVA DEVELOPER 5 YEARS EXPERIENCED REAL INTERVIEW REPORT

## **For Mandatory Skills**

core java:

- encapsulation n abstraction--yes
- can we overload the method by changing the return type--yes
- auto boxing n unboxing --yes
- custom runtime exception class--yes
- when we prefer array list n linked list--yes
- why we need to override the equals n hash code()--yes
- diff types of class loader--no
- can we call the run () directly instead of start--yes
- hash map n hash table --no
- scope of protected n default access specifier--yes
- use of optional--yes

spring boot:

- flow of sprint boot--yes
- life cycle method of spring bean--yes
- how we can handle exception in spring boot--yes
- spring profile concept --no
- bean factory n application context--partially yes

restapi:

- put n post--yes
- @pathvariable n @requestparam--yes
- want to create the restapi which is responsible to generate the xml format--no

hibernate:

- update n merge()--partially yes
- diff auto ddl operation of hibernate--yes
- get() n load()--no

maven :

- what is mean of mvn clean install commands--no

## **Questions asked**

- Tell me about yourself and the technology you work on.
- Have you worked with Hibernate or Spring Boot?
- What are the differences between encapsulation and abstraction in Java?
- Can we create an object for an abstract class?
- Explain method overloading and method overriding with an example.
- What is autoboxing and unboxing in Java? Provide an example.
- How can we create a custom runtime exception class?

- When do we prefer using ArrayList and LinkedList in Java?
- Why do we need to override equals method in a bean class?
- What is the purpose of the Observer class in Java?
- Explain the flow in a Spring Boot application starting from the main class.
- What annotations are used in a Spring Boot application?
- What are the lifecycle methods of a Spring Bean?
- How can exceptions be handled in Spring Boot?
- What is the difference between update and merge methods in Hibernate?
- What are auto DDL operations in Hibernate?
- Explain the difference between PUT and POST methods.
- What are the differences between @PathVariable and @RequestParam in Spring?
- How can we generate JSON format data in Spring?
- Write a program to print all prime numbers between 1 and 21.- What is the difference between collection of an object and a stream?
- Can you explain about your current project?
- Are you working on a support project or a development project only?
- Do you work on agile methodology?
- How many days does one Sprint go in case of agile?
- Have you worked with Jenkins before?
- Can you explain what Agile means to you?
- Have you heard of CR? What are the full forms of CR?
- Have you used Scrum Master in your projects?
- Any feedback you would like to share?

# **JAVA MICROSERVICES 10 YEARS EXPERIENCED REAL INTERVIEW REPORT**

## **For Mandatory Skills**

**Java :** The Candidate is working as architect from last 7 years and probably he will not write java code and candidate can guide the team members.

## **SpringBoot/Microservices Arch/ Design :**

I have given scenarios based question application Imagine for developing a social media platform similar to Twitter. Users can create posts, comment on posts, like posts, and follow other users. How would you design the backend architecture to support these functionalities while ensuring Security, scalability and performance including cloud aspect. The Candidate was able to explain Micro services architecture, Spring Boot best practices, cloud computing concepts, AWS cloud, CI/CD , Docker , Demonstrated a average understanding of project management methodologies and best practices.

The candidate demonstrated a good understanding of design patterns in Java applications.

Suggested improvement in recognizing and applying commonly used design patterns to enhance code structure and maintainability.

The candidate's design approach seemed to good a comprehensive consideration of scalability and performance aspects having deeper exploration of strategies for designing scalable and high-performance Java applications.

The candidate has demonstrate a comprehensive understanding of EKS concepts and experience in deploying and managing containerized applications using Kubernetes.

## **For Good To Have Skills**

The candidate has experience in AWS and Kubernetes technology selection during the design of application.

## **For Soft Skills**

Verbal : Good

## **Coding Feedback**

The candidate is architect from last 7 year and will not write the Java code. Since coding test was not mandatory i did not asked the question about this topic.

## **Recommendation And Other Strengths**

The candidate demonstrated a good understanding of design patterns in Java applications and Microservices, also having good understanding in Cloud , CI/CD Kubernetes.

Questions asked

- Tell me something about yourself and your profile, please.
- What is your tech stack?
- Can you explain your recent experience of designing?
- What would be your architecture for a social media platform similar to Twitter?
- How would you design such an application considering performance, cloud aspects, and security?
- What are the APIs you can use for the application scenario mentioned earlier?
- When would you choose GraphQL over REST APIs?
- Have you ever used messaging systems like Kafka?
- Can you tell me about different types of event-based architectures in the market?
- Can you describe your project requirements?

# **JAVA DEVELOPER 6 YEARS EXPERIENCED REAL INTERVIEW REPORT**

## **Questions asked**

- Can you tell me about yourself, your skill sets, and the projects that you worked on?
- Do you have experience in Docker and Kubernetes?
- What about Kafka and Zookeeper?
- Any experience with monitoring tools like Grafana, ELK, or Kibana?
- What are solid design principles?
- What is a predicate in Java?
- What is the Diamond problem of multiple inheritance?
- What is the @Qualifier annotation in Spring?
- What are the differences between default and protected access modifiers in Java?
- Can you please share your screen and open the test coding window?
- Explain the logic for finding the most repeated city in an array of cities.

## **For Mandatory Skills**

Did well in Java

Good attitude.

I would recommend hiring him as i felt during the interview - candidate is sincere and has worked hard to get his basics right.

Candidate is good in Java coding and good theoretically.  
not much experience in Kafka

Overall -

Candidate performed well in core java and datastructures theoretically.

Candidate performed well in Java 8 questions theoretical ones.

Candidate has good knowledge and experience in springboot.

Communication skills are igood

Good in coding skills.

Topics discussed -

Stream API YES

Map, filter, flatmap YES

Sort hashmap by values YES

What is entryset and Map.Entry YES

Concurrent hashmap vs hashtable YES

Default and protected access specifiers YES

JVM memory model and garbage collection YES

Immutable classes YES

Singleton double check locking YES

Rest controller vs Controller YES

Put vs patch vs post YES

Enableautoconfiguration YES

Qualifier annotations YES

Diamond problem with multiple inheritance YES

SOLID design principles YES

Reference types in java YES

Circuit breaker pattern YES

## JAVA DEVELOPER 4 YEARS EXPERIENCED REAL INTERVIEW REPORT

### **For Mandatory Skills**

Contract between .equals() and hashCode() - no  
Volatile and Transient - not clear with transient  
Why functional interface contains SAM - not clear  
Predicate and Function functional interface - yes  
Different intermediate operations in stream - yes  
Map and FlatMap in java8 - yes  
Failsafe and failfast iterators - yes  
Different classloaders in jvm - yes  
method overriding wrt checkedExceptions - not clear

@Primary annotation usage - yes  
Circular dependency resolution - no  
PointCut in AOP - not working on AOP  
Global exception handling in SB - yes  
Spring boot actuators - yes

candidate is having idea of JSON and XML formats.

PUT and PATCH request - yes  
401 and 403 response codes - not clear with 403  
pathparam and requestParam - yes  
know the use of feign clients.

where and having clause - yes  
Inner join and outer join - yes  
Composite key - yes

### **Questions asked**

- Can you please little bit tell me about yourself, your primary technical skills, and your day-to-day work?
- What are the primary technical skills you are using in your current project?
- Do you have any experience with RESTful web services?
- Which version of Java are you using? Any experience with caching mechanisms like Hazelcast?
- Do you have any experience with RabbitMQ?
- Explain the contract between the 'equals' method and the 'hashCode' method.
- Suppose I have two objects with the same hash code. Will these objects be equivalent by the 'equals' method?
- Explain the use of the 'volatile' keyword.
- What is the advantage of reading data from main memory using the 'volatile' keyword?
- What is the purpose of the 'transient' keyword in Java?
- Why does a functional interface contain a single abstract method?
  
- What are the different intermediate operations in streams?
- What is the difference between 'map' and 'flatMap' in streams?
- How can we resolve circular dependency problems in Spring?
- What is pointcutting AOP?
- How can we achieve global exception handling in Spring Boot?
- What are Spring Boot actuators used for?
- What is the difference between JSON response and XML response?
- What is the difference between a PUT request and a PATCH request?
- Explain the difference between 401 and 403 response codes.
- What is the difference between a path param and a query param?
- What is the purpose of the '@Retryable' annotation?
- What is the difference between 'HAVING' and 'WHERE' clauses in DB queries?
- Explain the difference between inner join and outer join.- Can you tell me this program is valid or not? behind that?
- What are the different types of class loaders in JVM?
- From where will all class loaders load the classes?
- What is the reason why it is correct?
- Which OOPs concept is there in this problem?
- In the context of overriding, is it valid or not? What are the rules for exceptions in overriding?
- Can you please share your screen?
- Create a simple List of string fruits and add some fruits to this list.
- Write a simple Java 8 code to get the frequency of each fruit.
- Is there any other fruit apart from banana that contains the character 'b'?

# **SENIOR JAVA DEVELOPER 10 YEARS EXPERIENCED REAL INTERVIEW REPORT**

## **For Mandatory Skills**

The candidate was fine with coding and completed the round with ease. He had understanding of DSA and Algorithm as expected. His experience in Multithreading was average and needs more revision but he was fine with Designing and Design Patterns. He could answer Spring /Spring Boot concepts and annotations properly. He could answer questions related to Microservices like Circuit Breaker and Rate Limiting. He did not know BulkHead pattern though. He could answer some questions in AWS on DB side, but needs better theory knowledge in AWS service as he could answer few questions in SQS, Global Accelerator etc

## **For Good To Have Skills**

The candidate was overall above average in technology

## **For Soft Skills**

The candidate had average communication skills but he needs to improve on fluency and giving proper examples

## **Coding Feedback**

The candidate was fine with his coding skills and looks hands on

## **Recommendation And Other Strengths**

The candidate was found to be suitable as per JD. He coded with ease and was good with DSA and Algorithms, Spring/Spring Boot and Microservices. He was not fluent with Multithreading and AWS but he can fill the gaps. Hence recommended to be SELECTED

Questions asked

- Have a quick background about yourself and your responsibilities.
- Describe your roles and responsibilities in your current company.
- How many people report directly to you?
- Is there a preferred coding test for this interview process?
- What is the internal architecture of ConcurrentHashMap?
- What is the difference between read lock and write lock?
- What is the time complexity of the search operation in a TreeMap?
- Have you worked with multi-threading and concurrency?
- What is a blocking queue?
- What is a CompletableFuture?
- What is the happens-before guarantee in the volatile keyword?
- What is the strategy design pattern?
- Can you explain dependency injection in Spring Boot?
- What is the re-tracing?
- Explain the bulkhead pattern in microservices architecture.
- Explain the circuit breaker pattern in microservices architecture.
- Have you worked with AWS?
- What is the difference between CloudFront and Global Accelerator?- What are Edge locations?
- What is geoproximity routing in Route 53 Cloud?
- What services have you worked on?
- What is the difference between short polling and long polling related to SQS?
- What is visibility timeout in case of SQS?
- What is DAX Dynamo accelerator?
- What kind of database is DynamoDB?
- What do you understand by the concept of eventual consistency?

## **REAL INTERVIEW EXPERIENCES SHARED BY MY CONNECTIONS IN X/TOPMATE**

Below are some interview experiences shared by my connections. I am really grateful and thankful to them for this selfless contribution.

I have added answers to the questions that they have shared with some Tips. Hope this newly added section is helpful in your interview preparation.

### **2 YEARS EXPEREINCED JAVA DEVELOPER INTERVIEW IN A SERVICE COMPANY**

#### **1. Difference between equals and == in Java**

== operator: Compares references for objects (whether they point to the same memory location). For primitives, it checks value equality.

equals() method: Compares the content of objects and can be overridden to implement custom logic, such as comparing the values of two strings.

Example:

```
String a = new String("hello");
String b = new String("hello");
System.out.println(a == b); //false
System.out.println(a.equals(b)); // true
```

I recall mentoring him on this point early in his career, as

it's easy to confuse object references with actual values.

## 2. Explain the final keyword in variable, method, and class declarations.

Final variable: Once assigned, its value cannot be changed.

```
final int x = 10;  
x = 20; // Compilation error
```

Final method: Prevents the method from being overridden by subclasses. Ensures behavior remains unchanged in subclasses.

```
class Parent {  
    final void display() {  
        System.out.println("Parent");  
    }  
}  
  
class Child extends Parent {  
    // Cannot override 'display()' method.  
}
```

Final class: Cannot be subclassed, which is useful for creating immutable classes like String.

```
final class Example { }
```

This topic is often brought up in interviews, especially for junior roles. I once emphasized the significance of immutability when discussing security and optimization.

## 3. What is the @Qualifier annotation in Spring Boot?

@Qualifier is used when multiple beans of the same type are available in the Spring context, and you want to specify

which one to inject.

Example:

```
@Service  
public class ServiceA implements MyService { }  
@Service  
public class ServiceB implements MyService { }  
@Autowired  
@Qualifier("serviceA")  
private MyService myService;
```

#### **4. Can static methods be overridden?**

No, static methods can't be overridden but can be hidden by declaring a static method with the same signature in a subclass.

Example:

```
class Parent {  
    static void show() {  
        System.out.println("Parent");  
    }  
}  
class Child extends Parent {  
    static void show() {  
        System.out.println("Child");  
    }  
}
```

In this case, Child's show() method hides the Parent method.

Understanding the distinction between method overriding

and hiding is essential.

## **5. Which part of memory (heap or stack) is cleaned by the garbage collector?**

The heap is managed by the garbage collector, which removes objects no longer in use. The stack holds local variables and method calls, and is cleaned up automatically when methods return.

This is a foundational memory management question that I'd emphasized during our sessions. It's important to understand the implications of the heap and stack when designing performant applications.

## **6. Why are Strings immutable in Java?**

Java Strings are immutable due to:

Security: Strings are often used for sensitive data like passwords or file paths. Immutability ensures these can't be changed.

Performance: String literals are reused from the string pool, saving memory.

Thread safety: Immutable objects are inherently thread-safe.

This question seemed simple, but you need to articulate deeper insights during the interview.

## **7. Difference between HashSet and TreeSet**

HashSet: Does not maintain order and has constant time ( $O(1)$ ) for add, remove, and contains operations.

TreeSet: Maintains sorted order with  $O(\log n)$  complexity

because it is based on a red-black tree.

Understanding time complexity can make a huge difference when discussing collections in interviews, and make sure to emphasize it.

## 8. Difference between HashMap and HashSet

HashMap: Stores key-value pairs, with keys being unique.

HashSet: Stores only unique values and internally uses a HashMap where values are stored as keys.

**When I went over collections in one of my sessions, some were surprised to learn that HashSet uses HashMap internally. This understanding really helps in the interview.**

## 9. What are the SOLID principles?

S: Single Responsibility Principle — A class should only have one reason to change.

O: Open/Closed Principle — Classes should be open for extension but closed for modification.

L: Liskov Substitution Principle — Subtypes should be replaceable for their base types without affecting the correctness.

I: Interface Segregation Principle — Clients should not be forced to implement interfaces they don't need.

D: Dependency Inversion Principle — High-level modules should not depend on low-level modules; both should depend on abstractions.

## 10. What does the @SpringBootApplication annotation do internally?

The `@SpringBootApplication` annotation is a combination of:

`@Configuration`: Marks the class as a source of bean definitions.

`@EnableAutoConfiguration`: Enables automatic configuration based on classpath settings.

`@ComponentScan`: Scans the package for components and beans.

## 11. What does `@Autowired` do in Spring?

`@Autowired` is used to automatically inject dependencies. Spring resolves the necessary bean from the `ApplicationContext` and injects it into the marked field, constructor, or method.

Example:

`@Autowired`

`private MyService myService;`

Tackle follow-up questions regarding dependency resolution and `@Qualifier` for resolving multiple beans.

## 12. Are singleton beans thread-safe in Spring?

Spring beans are singleton by default, meaning a single instance is shared across the application. However, singleton beans are not thread-safe if they maintain mutable state.

You must manage concurrency either with synchronization or by using patterns like `ThreadLocal`.

**During mentoring sessions, I had often stressed that understanding thread safety is crucial, especially for**

**long-running services in Spring Boot. You can also share a scenario where you used ThreadLocal to handle thread safety in a multi-threaded environment.**

### **13. Difference between Abstraction and Encapsulation?**

Abstraction: Hides implementation details and only exposes functionality.

Encapsulation: Bundles data and methods together, restricting access to details through access modifiers like private and public.

**I often encourage to use real-world analogies when explaining these concepts. You can share an example of a car abstraction, encapsulating engine details while providing simple operations like start and stop.**

## **3 YEARS EXPERIENCE IN SERVICE COMPANY**

### **1. Do you have any experience with RESTful web services?**

**Answer:**

Yes, I have worked extensively with RESTful web services. I have experience in designing and implementing REST APIs, adhering to REST principles like statelessness, resource-based architecture, and proper use of HTTP methods such as GET, POST, PUT, and

**DELETE.** I've also used tools like Postman for testing REST APIs and integrated Swagger for API documentation.

**Note:**

While answering, the key is to mention any hands-on experience, including the frameworks you've used, such as Spring Boot for building REST services. Focus on the projects you've implemented and highlight any RESTful web service best practices you followed.

**2. Which version of Java are you using? Any experience with caching mechanisms like Hazelcast?**

**Answer:**

I have been using Java 8 for most of my projects. However, I am also familiar with Java 11 features like local variable syntax for lambda expressions, HTTP client, and more. Regarding caching, I have experience with Hazelcast for distributed caching in a microservices architecture. It helped us reduce the load on the database by storing frequently accessed data in memory.

**Note:**

Make sure you highlight specific versions of Java and any advanced features you are using. If you've worked with Hazelcast, Redis, or any other caching mechanism, talk about how it improved the performance of your applications.

**3. Do you have any experience with RabbitMQ?**

**Answer:**

Yes, I have used RabbitMQ in a few projects to implement

message queuing between microservices. RabbitMQ helped us decouple services and allowed asynchronous communication. I used Spring Boot's Spring AMQP to integrate RabbitMQ into the application.

Note:

Even if you have limited experience, mention the use cases where RabbitMQ can be beneficial and +how you have used it in your projects. If you've used other messaging systems like Kafka, mention that as well.

---

**4. Explain the contract between the equals method and the hashCode method.**

Answer:

The equals and hashCode methods work together to maintain the consistency of objects in collections like HashMap and HashSet. The contract is:

If two objects are equal according to the equals() method, they must have the same hashCode().

If two objects have the same hashCode(), they are not necessarily equal.

Note:

Be clear about the contract because it is crucial when working with collections. Also, mention scenarios where violating this contract can cause issues with data retrieval in HashMap or HashSet.

---

**5. Suppose I have two objects with the same hash code. Will these objects be equivalent by the equals method?**

Answer:

No, two objects having the same hash code do not necessarily have to be equal according to the equals method. The hash code just determines the bucket in which the object is stored in hash-based collections. For two objects to be considered equal, both hashCode() and equals() should return true.

Note:

This is a common misconception. Emphasize that hash collisions are possible, which is why both methods need to be considered for proper object comparison.

---

## **6. Explain the use of the volatile keyword.**

Answer:

The volatile keyword in Java ensures that the value of a variable is always read from the main memory, not from the thread's local cache. This is useful in multi-threaded applications where multiple threads may update the value of a shared variable.

Note:

Mention that volatile ensures visibility of changes across threads, but it doesn't guarantee atomicity. For more complex cases like increments, synchronized or Atomic classes should be used.

---

## **7. What is the advantage of reading data from main memory using the volatile keyword?**

Answer:

The advantage of using volatile is that it ensures all threads see the most up-to-date value of a variable. This eliminates issues caused by thread-local caches where different

threads might be working with stale data.

Note:

Highlight that while volatile is useful for visibility, it doesn't solve all synchronization issues. For example, it doesn't protect against race conditions.

---

## **8. What is the purpose of the transient keyword in Java?**

Answer:

The transient keyword is used in serialization. It marks a variable so that it is not included in the serialization process. This is useful for variables that represent sensitive information, like passwords, or for variables that are not meant to be saved.

Note:

Give examples of when and where you've used transient. It's a small concept, but demonstrating practical usage helps reinforce understanding.

---

## **9. Why does a functional interface contain a single abstract method?**

Answer:

A functional interface is meant to represent a single functionality, which is why it contains only one abstract method. This makes it suitable for lambda expressions and method references in Java 8.

Note:

You can mention examples like Runnable or Comparator. The beauty of functional interfaces is how they simplify

functional programming in Java.

---

## 10. What are the different intermediate operations in streams?

**Answer:**

Intermediate operations in streams are those that return another stream and are used to transform data. Examples include:

*filter()*  
*map()*  
*sorted()*  
*distinct()*

**Note:**

Focus on how intermediate operations are lazily executed, meaning they don't trigger processing until a terminal operation (like `collect()`, `forEach()`) is called.

---

## 11. What is the difference between map and flatMap in streams?

**Answer:**

`map()` transforms each element in a stream into another form, typically one-to-one mapping.

`flatMap()` transforms each element into a stream of elements, flattening the structure. It is used for one-to-many mappings, especially when working with collections of collections.

**Note:**

Provide examples showing the difference. This can be tricky, so be prepared with a scenario where `flatMap` is

essential, such as working with nested lists.

---

## 12. How can we resolve circular dependency problems in Spring?

**Answer:**

Circular dependencies in Spring can be resolved by:

Using `@Lazy` on one of the dependencies to delay its initialization.

Breaking the dependency by restructuring the application's design.

Using setter injection instead of constructor injection.

**Note:**

Circular dependencies are a design flaw, so the best solution is to refactor and redesign the components to avoid them altogether.

---

## 13. What is pointcutting in AOP?

**Answer:**

Pointcutting in AOP (Aspect-Oriented Programming) defines where advice (additional behavior) should be applied in the code. A pointcut refers to a specific join point, like method execution, where the advice will be triggered.

**Note:**

Explain how AOP is useful for separating cross-cutting concerns, such as logging, transaction management, and security. Be sure to mention practical use cases.

---

## **14. How can we achieve global exception handling in Spring Boot?**

Answer:

Global exception handling in Spring Boot can be achieved using `@ControllerAdvice` along with `@ExceptionHandler`. This allows centralizing exception handling logic across multiple controllers.

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(value = Exception.class)
    public ResponseEntity<Object>
    handleException(Exception e) {
        return new ResponseEntity<>("Error occurred: " +
        e.getMessage(),
        HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Note:

Explain the importance of clean error handling and centralized logging, which reduces duplicate error handling code in controllers.

---

## **15. What are Spring Boot actuators used for?**

Answer:

Spring Boot Actuators are used to monitor and manage a Spring Boot application. They provide various production-ready features like health checks (`/health`), metrics (`/metrics`), and environment details.

Note:

Explain the use of Actuators in production environments and how they integrate with monitoring tools like Prometheus and Grafana for application monitoring.

---

**16. What is the difference between a JSON response and an XML response?**

Answer:

JSON (JavaScript Object Notation) is lightweight, easier to read, and widely used in RESTful APIs. XML (Extensible Markup Language) is more verbose but offers better support for document-centric data, namespaces, and schemas.

Note:

Highlight that JSON is generally preferred in modern applications due to its simplicity, but XML is still used for complex document-based systems.

---

**17. What is the difference between a PUT request and a PATCH request?**

Answer:

PUT: Used for full updates. The entire resource is replaced with the new data.

PATCH: Used for partial updates. Only the fields that need to be updated are sent in the request.

Note:

Emphasize that PUT is idempotent (multiple identical

requests have the same effect), while PATCH is more efficient for partial updates.

---

## **18. What is the role of the Spring Boot starter dependency?**

Answer:

Spring Boot starter dependencies are a set of convenient dependency descriptors that simplify the process of adding dependencies to a Spring Boot application. For example, using `spring-boot-starter-web` automatically includes all necessary dependencies for building web applications, such as Spring MVC and Tomcat.

Note:

Mention that starters reduce the need to specify individual dependencies, which simplifies the build configuration. It's also worth noting how this approach helps in maintaining consistency across projects by ensuring that compatible versions of libraries are included.

---

## **19. Can you explain what a @Configuration class is in Spring?**

Answer:

A `@Configuration` class is a class annotated with `@Configuration` that defines one or more `@Bean` methods. It indicates that the class can be used by the Spring IoC container as a source of bean definitions. The `@Bean` methods within the class return instances of beans that Spring manages.

### Note:

Illustrate with a simple example, showing how a configuration class can help in defining beans and how it can be used to encapsulate configuration logic in a single place. Discuss the importance of using configuration classes over XML configurations for better readability and maintainability.

---

## **20. What are the different scopes available for Spring beans?**

Answer:

Spring supports several bean scopes, including:

**Singleton:** A single instance is created for the Spring container and shared across all requests.

**Prototype:** A new instance is created every time the bean is requested.

**Request:** A new instance is created for each HTTP request (only in a web application).

**Session:** A new instance is created for each HTTP session (only in a web application).

**Global Session:** A new instance is created for each global HTTP session (for portlet-based web applications).

### Note:

Discuss the implications of each scope, especially the performance and memory considerations. Singleton is the default scope, while Prototype can lead to performance issues if not managed properly.

---

## **21. What is dependency injection, and how is it**

## **implemented in Spring?**

**Answer:**

Dependency Injection (DI) is a design pattern that allows the creation of dependent objects outside of a class and provides those objects to a class through constructors, setters, or methods. In Spring, DI is achieved through the use of annotations like @Autowired, @Inject, and @Resource, or by XML configuration.

**Note:**

Emphasize the benefits of DI, such as reduced coupling and improved testability. Provide an example of constructor injection and setter injection to illustrate both methods.

---

## **22. What is the significance of @Transactional in Spring?**

**Answer:**

The `@Transactional` annotation in Spring defines the **scope of a single database transaction**. It can be applied at the method or class level. When a method is annotated with `@Transactional`, Spring automatically manages the transaction boundaries starting a transaction before the method runs and committing or rolling it back afterward.

**Key Benefits:**

- \* Ensures atomicity of a method (all or nothing)
- \* Automatically rolls back the transaction in case of an exception

- \* Reduces boilerplate code for manual transaction handling

#### Important Point:

- > By default, Spring only rolls back transactions for unchecked exceptions (subclasses of `RuntimeException` or `Error`).
- > If a method throws a checked exception, the transaction will not roll back automatically unless explicitly configured using `rollbackFor`.

#### Note:

Helps maintain data integrity, especially during operations involving multiple database changes (e.g., insert + update).

Supports customization via:

- \* Propagation levels (e.g., `REQUIRED`, `REQUIRES\_NEW`)
- \* Isolation levels (e.g., `READ\_COMMITTED`, `SERIALIZABLE`)
- \* Timeouts, readOnly flags, and more

---

## **23. How do you handle transactions in a Spring Boot application?**

#### **Answer:**

Transactions in a Spring Boot application can be handled using the `@Transactional` annotation. It can be used to annotate service layer methods where database operations take place. Spring will manage the transaction lifecycle, automatically committing or rolling back transactions based on success or failure.

#### Note:

Explain the importance of handling transactions properly to avoid data inconsistencies. Mention transaction propagation options like REQUIRED, REQUIRES\_NEW, and how they affect transaction behavior.

---

## **24. What are microservices, and how do they differ from monolithic architecture?**

Answer:

Microservices are an architectural style that structures an application as a collection of loosely coupled services. Each service is independently deployable, scalable, and can be developed using different technologies. In contrast, a monolithic architecture consists of a single, unified codebase where all components are tightly integrated.

### Note:

Discuss the advantages of microservices, such as improved scalability, flexibility in technology stacks, and easier maintenance. Also, mention the challenges, like service discovery, inter-service communication, and the need for more complex deployment strategies.

---

## **25. How do you ensure security in a Spring Boot application?**

Answer:

Security in a Spring Boot application can be ensured using Spring Security, which provides comprehensive security services for Java applications. It allows for authentication, authorization, and protection against common security

vulnerabilities. Configuration can be done using Java configuration classes or XML.

Note:

Discuss how Spring Security integrates seamlessly with Spring Boot, the importance of securing RESTful services, and using JWT for stateless authentication. Mention common security practices like role-based access control (RBAC) and securing sensitive endpoints.

---

**26. Can you explain the purpose of Swagger in RESTful APIs?**

Answer:

Swagger is a framework for API documentation that helps design, build, document, and consume RESTful APIs. It provides a user-friendly interface that allows developers and consumers to understand the API's endpoints, parameters, and response formats.

Note:

Explain how Swagger integrates with Spring Boot through the `springfox-swagger2` and `springfox-swagger-ui` dependencies. Highlight the benefits of using Swagger for API testing and documentation, which helps improve collaboration between frontend and backend teams.

---

**27. What is the purpose of Spring Boot DevTools?**

Answer:

Spring Boot DevTools is a set of tools that helps improve

the development experience by providing features like automatic restarts, live reload, and configuration properties that can be overridden for development. It allows developers to see changes without having to restart the server manually.

Note:

Mention how DevTools can significantly enhance productivity during development by minimizing downtime and making it easier to test changes on the fly.

---

**28. What is a DTO (Data Transfer Object), and why do you use it?**

Answer:

A DTO (Data Transfer Object) is an object used to transfer data between software application subsystems or layers. DTOs help in bundling multiple data attributes into a single object to reduce the number of method calls and can be useful for minimizing the amount of data transferred over the network.

Note:

Explain how DTOs can simplify API communication by providing a clear structure for data and potentially enhancing performance by limiting data exposure. Discuss when to use DTOs versus directly exposing entity classes.

---

**29. How do you perform unit testing in a Spring Boot application?**

**Answer:**

Unit testing in a Spring Boot application can be performed using the JUnit and Mockito frameworks. You can create test cases for your services, controllers, and repositories to ensure each component behaves as expected. Spring Boot provides excellent support for writing tests, with annotations like `@SpringBootTest` to set up the application context and `@MockBean` to mock dependencies.

**Note:**

Discuss the importance of testing in the development lifecycle and how unit tests can help catch bugs early. Provide examples of writing a simple test case for a service or controller.

---

## **30. What are the benefits of using Spring Boot?**

**Answer:**

The benefits of using Spring Boot include:

**Convention over Configuration:** Reduces the need for extensive XML configuration.

**Embedded Servers:** Supports embedded servers like Tomcat, which simplifies deployment.

**Auto-configuration:** Automatically configures beans based on dependencies present in the classpath.

**Production-ready features:** Includes metrics, health checks, and externalized configuration.

**Note:**

Highlight how Spring Boot accelerates the development process, making it easier to create stand-alone, production-ready applications with minimal setup.

## **6 YEARS EXPERIENCED JAVA DEVELOPER INTERVIEW EXPERIENCE IN A PRODUCT COMPANY**

### **Round 1: Technical Interview (70–75 minutes)**

The first round was conducted with an experienced professional with 6 years of experience. The questions were focused on distributed systems, message queues, caching, and core Spring Boot concepts.

Here's a breakdown of the questions and approach to answering them:

#### **1. Distributed Systems: Saga, 2PC, 3PC**

**Question: Explain the differences between the Saga pattern, Two-Phase Commit (2PC), and Three-Phase Commit (3PC) in distributed systems.**

**Answer:**

**Saga Pattern:** A sequence of local transactions. If one transaction fails, compensating transactions undo the changes. Best for long-running transactions without the need for strong consistency.

**2PC:** Involves two phases — Prepare and Commit. All participants prepare to commit, and if all are successful, the coordinator tells them to commit. Guarantees strong consistency but can lead to blocking if the coordinator fails.

**3PC:** Introduces a CanCommit phase between Prepare and Commit. This reduces blocking but adds more complexity.

It's less commonly used due to overhead and failure scenarios.

## **2. Kafka Implementation and Use Cases**

**Question:** How do you implement Kafka in a system, and what are its typical use cases?

**Answer:**

Kafka is a distributed event streaming platform. It handles high throughput, low latency, and fault tolerance. Common use cases include real-time analytics, log aggregation, and message brokering. He implemented Kafka in a logging service where multiple microservices push logs to Kafka, which are then processed by a consumer service for monitoring.

## **3. Redis Implementation and Use Cases**

**Question:** How have you implemented Redis, and when should it be used?

**Answer:**

Redis is a key-value store primarily used for caching, pub/sub messaging, and distributed locks. He used Redis to cache frequently accessed data like product information in an e-commerce system to reduce load on the database and improve response times.

## **4. Spring Boot Bean Lifecycle**

**Question:** Explain the lifecycle of a Spring Bean.

**Answer:**

The Spring Bean lifecycle starts with instantiation via the constructor. Then, dependency injection occurs, followed by any custom initialization methods like @PostConstruct. The bean is then ready for use. When the context is destroyed, any custom destroy methods, such as those

marked with @PreDestroy, are called.

## 5. Abstract Classes vs. Interfaces

**Question:** When would you use an abstract class if we have interfaces?

**Answer:**

Abstract classes are used when some default behavior needs to be shared across multiple subclasses. They can contain both abstract methods (to be implemented by subclasses) and concrete methods. Interfaces define contracts and allow multiple inheritance (as a class can implement multiple interfaces), whereas an abstract class provides a common base with shared functionality.

## 6. Race Condition Snippet

**Question:** Identify the race condition in this code snippet.

**Answer:**

The interviewer presented a multi-threaded code snippet where two threads accessed and modified a shared variable.

He identified the absence of proper synchronization, which can cause inconsistent results if two threads try to update the value simultaneously. A solution would involve using synchronized blocks or locks.

## 7. Leetcode Problem (LC-1304)

**Problem:** Given n, return any array of n integers such that they sum up to 0.

**Answer:**

The solution involves creating an array with pairs of opposite numbers (e.g., -1, 1, -2, 2) and adding 0 if n is odd. This was a relatively simple Leetcode Medium

problem, which he solved in the extended time frame.

---

## **Round 2: Technical Interview with the Principal Architect (1 Hour)**

This round was initially supposed to be the third round but was conducted second due to a scheduling change. It was focused on data structures, design patterns, and a high-level design (HLD) discussion.

### **1. Simple DSA Question (Without HashMap)**

**Question:** Check if a string has repeated characters without using a HashMap.

**Answer:**

Used a Boolean array of size 256 (ASCII characters) to track occurrences of each character. If a character is found twice, the string has duplicates.

### **2. Singleton Pattern**

**Question:** Where have you used the Singleton pattern in your work?

**Answer:**

Singleton ensures only one instance of a class exists throughout the application. He used it in a logging framework where only one logger instance is required to log messages across different parts of the application, reducing memory overhead.

### **3. Strategy Pattern**

**Question:** Where have you applied the Strategy pattern?

**Answer:**

The Strategy pattern was used in a payment gateway

system. Different payment methods like credit cards, PayPal, and UPI had varying implementations, and he used the Strategy pattern to allow clients to switch between these payment strategies dynamically at runtime.

## **4. Design Patterns in Java**

**Question:** Explain some design patterns that are used internally in Java.

**Answer:**

Builder Pattern: Used in classes like StringBuilder.

Factory Pattern: Used in the Calendar class.

Prototype Pattern: Implemented by the cloneable interface.

Observer Pattern: Used in java.util.Observer and Observable.

## **5. SMS Alert Service HLD**

**Question:** Design a high-level architecture for an SMS alert service with high throughput.

**Answer:**

The system requires a distributed queue (e.g., Kafka) to handle large-scale SMS requests. Producers (services generating SMS alerts) push requests into Kafka.

Consumers (SMS processing services) pull requests, process them, and send them to an external SMS gateway. The system uses Redis to cache frequent templates and a load balancer to distribute traffic evenly across processing nodes.

## **6. Scenario-Based Questions**

**Question:** In the last few minutes, the interviewer asked scenario-based questions on my past experience, such as:

**How did you manage failure scenarios in distributed**

**services?**

**What debugging tools did you use for high-throughput systems?**

---

### **Final Round: Salary Negotiation**

After the technical rounds, He received a call for the salary negotiation round in the evening. The interview process was quite extensive and required a good understanding of distributed systems, design patterns, and high-level design. The final negotiations were straightforward, focusing on the offer package, benefits, and growth opportunities.

---

### **Key Takeaways**

- 1. Preparation:** Cover distributed systems, design patterns, and Spring Boot core concepts if you're targeting product-based companies.
  - 2. Practical Use Cases:** Be prepared to explain real-world applications of design patterns and architectural decisions.
  - 3. Data Structures & Algorithms:** Even if the questions seem simple, the interviewer may limit the use of common tools like hashmaps to test problem-solving skills.
- 
-

# **Chapter 25: My Story**

I began my career at a service-based company, where I found myself working on a legacy product built in C. However, my true passion lay in Java, and I was determined to transition into this space. I proactively pursued several Java trainings within the company, and after two years of perseverance and over 20 interviews, I successfully secured offers from three mid-sized product companies. With a deep interest in the investment banking domain, I made a strategic move 1.5 years later, joining an investment bank. Over the course of my 14.5-year career, I have had the privilege of working with many of the top investment banks, consistently excelling in interviews and seizing opportunities to grow in this dynamic field.

I graduated from a tier-3 college, bottom of Tier-3 so Tier-3 that it has shut down. I worked hard and improved my skills significantly. Through perseverance, I succeeded in securing positions and achieving the salary I had aspired to.

**IF I CAN YOU CAN !!**

© 2024 by SUMIT. All rights reserved.

No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from the publisher.

First Edition:

(Version 1) - Sep-2024

(Version 2) - Oct-2024

(Version 3) - Nov-2024

(Version 4) - Dec-2024

(Version 5) - Feb-2025

(Version 6) - July-2025

**(Version 7) - Aug-2025 (current version)**

Published by SUMIT



<https://topmate.io/interviewswithsumit/>

[https://x.com/SumitM\\_X](https://x.com/SumitM_X)

<https://www.youtube.com/@wearetechies>

<https://medium.com/@sumitmm>