THE COMPLETE JAVA INTERVIEW GUIDE

Cracking the

JAVA
Interview

First Edition 2024

Published by SUMIT

https://topmate.io/interviewswithsumit/

https://x.com/SumitM_X

https://www.youtube.com/@wearetechies

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

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.

7

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 captives, 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:
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 e.g.there are many things you can do without objects e.g. static methods. Also, primitive variables are not objects in Java.

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 is superclass is resolved to the static method declared in the superclass.

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.

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

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.

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.

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.

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

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

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()).

18. 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();
   }
}
```

19. 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:

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

20. Can an Interface Have Constructors?

Answer:
No, interfaces cannot have constructors.

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

21. 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

22. 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.

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

24. Can we execute a program without main() method?

Answer:
- 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:

- o 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:
  - o 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:
  - o Applets: Run through lifecycle methods like init() and start(). No main needed.
  - o Servlets: Use init(), service(), etc., managed by a servlet container.
  - o JavaFX: Starts with Application.launch(), which internally might use a main but not explicitly in your code.
- Frameworks and Custom Launchers:
  - o Some frameworks might use bytecode manipulation or other JVM capabilities to define alternative entry points or simulate a main method.
- JVM Behavior:
  - o Even if you can execute some code (like static blocks), the JVM will look for a main method to keep the application running. If not found, it will terminate after running initializers.

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.

25. 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

26. 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.

27. 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.

28. 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.

29. 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.

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.

30. 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.

31. 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.

32. What is the difference between Class.forName() and ClassLoader.loadClass()?

Answer:
Class.forName():

- Loads and initializes the class.
- Syntax: Class.forName("com.example.MyClass");
- Initializes: Static blocks and fields are executed immediately.
- Use Case: 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");
     - Initializes: Class is initialized later, when it's first used.
     - Use Case: When you want to delay the initialization for better control.

Key Difference

     - Class.forName(): Loads and initializes.
     - ClassLoader.loadClass(): Loads only, initializes later

33. 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

34. 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.

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

35. 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.

36. What are dynamic proxies in Java, and how can they be used?

Answer:
Dynamic proxies allow the creation of proxy instances at runtime that can intercept method calls and add additional behavior (e.g: logging, access control).

Use java.lang.reflect.Proxy and an InvocationHandler to create a dynamic proxy.
They are widely used in Spring AOP

37. 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.

38. 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 anwser something like this - I have implemented several performance optimizations, including:

1. <u>Profiling and Monitoring:</u> Used tools like VisualVM and JProfiler to identify and resolve performance bottlenecks.

2. <u>Efficient Data Structures:</u> Replaced inefficient data structures with more appropriate ones (e.g., using HashMap instead of ArrayList for frequent lookups).

3. <u>Database Optimization:</u> Improved query performance by optimizing indexes and using batch processing for database operations.

4. <u>Caching:</u> Implemented caching strategies (e.g., using @Cacheable annotation) to reduce redundant computations and database calls.

5. <u>Concurrency Improvements:</u> Optimized multithreading and used concurrent collections to handle high-throughput scenarios efficiently

Try to include examples from your project

39. 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.

40. Explain shallow copy vs deep copy in the context of Java cloning.

Answer:
A shallow copy of an object copies the fields of the original object to the new object. If the field is a primitive type, it copies the value. If the field is a reference type (like an object), it copies the reference, meaning both the original and the cloned object will reference the same instance.

In contrast, a deep copy creates a new instance for every referenced object in the original object, resulting in two completely independent objects.

41. [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();}}
```

42. 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.

43. What happens `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.

44. **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:

```java
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.

45. **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?**

```java
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.

```java
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.

Chapter 4: static keyword

46. 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.

47. Can a static method be abstract ?

Answer:
No, a static method cannot be abstract in Java. Abstract methods are meant to be overridden in subclasses, but static methods are bound at compile-time and are not associated with any instance of the class. Since static methods belong to the class itself and not to any instance, making them abstract would contradict the purpose of abstract methods, which require a concrete implementation in a subclass.

48. Can classes be static?

Answer:
Yes , inner classes can be static

49. 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.

1. **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

2. 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.

3. How can you make Araylist 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.toUnmodifiableList(): Converts a stream into an immutable list (Java 10+).
4.Custom Immutable Class: For maximum control, create your own immutable list wrapper.

4. 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
```

5. 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.

6. 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
```

7. How does the LinkedList class implement the List, Deque, and Queue interfaces simultaneously?

Answer:
Discuss how LinkedList implements these interfaces, providing methods for list operations (add, get, remove), queue operations (offer, poll), and deque operations (addFirst, removeLast)..

8. 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.

9. 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.

10. 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.

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

Answer:
Using mutable objects as keys in a HashMap can lead to unexpected behavior because the hash code may change after the object is inserted, leading to incorrect bucket assignments and failed lookups or updates.

12. [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.

13. 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).

14. 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.

15. 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.

16. 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.

17. 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.

18. How does Collections.sort() work internally?

Answer:
Collections.sort() internally uses the List.sort() method, which was introduced in Java 8. This method, in turn, uses the TimSort algorithm, which is a hybrid sorting algorithm derived from merge sort and insertion sort.

TimSort is highly optimized for real-world data, especially for data that is already partially sorted. It has a worst-case time complexity of $O(n \log n)$, and it is stable, meaning it maintains the relative order of equal elements.

19. 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.

20. 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.

21. 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.

Chapter 6: Exception Handling

22. 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.

23. 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

24. 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.

25. 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:
```java
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.

26. 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.

27. 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.

28. What will be the output of this code?

```java
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.

29. 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.

30. 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.

31. **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.

32. 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

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

// Custom Unchecked Exception

```java
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.

Chapter 7: Enums

33. Can an enum extend another class in Java?

Answer:
No, enums cannot extend another class in Java because they implicitly extend java.lang.Enum, which is a final class. However, an enum can implement interfaces.

34. 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:

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

35. 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.

36. 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 {
```

```java
        @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

37. What happens if you do not define serialVersionUID in a Serializable class?

Answer:
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.

38. 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.

39. 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.

40. What is the readResolve() method work 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.

41. 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.

42. 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.

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

Answer:
The serialization will fail with a NotSerializableException. To fix this, mark the non-serializable member as transient so it is not serialized, or ensure that all members of the class are serializable.

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

Chapter 9: Generics

45. 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.

46. 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.

<u>Contravariance (? super T):</u> 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.
>
> Covariance is used when you need to read from a structure, while contravariance is used when you need to write to it.

47. 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.

48. **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
}
```

49. 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.

50. 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).

51. **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:
You can write the method like this:

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

Using the @SafeVarargs annotation suppresses the unchecked warnings that arise from using varargs with generics. Without this annotation, there is a risk of heap pollution, which occurs when a generic array is created at runtime but the type information is erased.

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.

52. 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. Internally, a string is backed by a character array (char[]), which holds the actual string data. Java strings are immutable, meaning once a string is created, its content cannot be changed. This immutability is facilitated by storing strings in a special memory area called the "string pool," where identical string literals are stored only once to save memory. When a new string is created, the JVM first checks the string pool; if the string already exists, it reuses the existing object, otherwise, it creates a new one.

53. 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 determines that there are no more references to an object, it marks the object for garbage collection and invokes its finalize() method. During the execution of finalize(), if the object assigns a reference to itself to a static variable or another object, it becomes reachable again, effectively "resurrecting" the object. However, this is generally discouraged because the finalize() method can introduce unpredictable behavior, and objects resurrected in this way will not be finalized again if they become unreachable in the future.

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

Answer:
- <u>Java 7 and earlier</u>: Parallel GC (Throughput Collector).
- <u>Java 8</u>: Parallel GC.
- <u>Java 9 - Java 17</u>: G1 GC (Garbage-First).
- <u>Java 18+:</u> G1 GC (default), with ZGC and Shenandoah as alternatives.

55. 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.

56. What happens in the memory when we use `new` Keyword?

Answer:
Cover below points:
- A new object is created in the heap memory.- A reference to the object is returned and stored in the stack.
- The constructor of the object is called to initialize it.
- The object reference points to the memory address of the newly created object.

57. 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.

58. 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.

59. 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.

60. What are different areas in Java Memory ?

Answer:
Below are different memory areas in Java:

- Heap: Stores objects and class instances.
- Stack: Stores method calls, local variables, and references to objects in the heap.
- Method Area: Stores class structures (e.g., runtime constant pool, field and method data).
- PC Register: Stores the address of the current JVM instruction being executed.
- Native Method Stack: Holds information for native (non-Java) method execution.

61. [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.

62. 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.

63. 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.

64. How can we monitor Garbage Collection activities in Java?

Answer:
To monitor garbage collection activities succinctly:

1. Use JVM Flags: Enable detailed GC logging with -XX:+PrintGCDetails and -Xloggc:<file> for log analysis.

2. JVisualVM: Utilize its Visual GC plugin for real-time monitoring of GC events, heap usage, and pause times.

GCViewer or GCEasy: Analyze GC logs offline for performance metrics and tuning suggestions.
Java Mission Control (JMC): For in-depth, low-overhead profiling including garbage collection with Java Flight Recorder.

JMX and jstat: For programmatic access to GC stats or simple command-line monitoring.

These tools help track GC frequency, duration, heap usage, and other vital metrics for optimizing application performance.

65. 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.

- Who Allocates: The operating system allocates memory to the JVM, and the JVM allocates it to objects during runtime.

66. 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:
1800K - 90K = 1710K

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:

22320K - 20610K = 1710K

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:
real=0.06 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.

67. 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

68. 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.

69. 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).

70. 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

71. 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.

72. 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.

73. 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.

74. 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.

75. What will be the output of the following code snippet?

```java
class Test {
    public void print(String s) {
        System.out.println("String");
    }

    public void print(Object o) {
        System.out.println("Object");
    }

    public static void main(String[] args) {
        Test test = new Test();
        test.print(null);
    }
}
```

Answer:

String

This behavior occurs because String is more specific than Object.

76. 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

77. 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

78. 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.

79. [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.

80. 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()

81. 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.

82. 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.

83. Explain the difference between Stream

API map and flatMap?

Answer:

map: Transforms each element in the stream.

    List<String> strings = Arrays.asList("a", "b");

    List<Integer> lengths =
    strings.stream().map(String::length).collect(Collectors.t
    oList());

flatMap: Flattens nested structures.

    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());

84. 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
```

85. 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.

86. 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 = list.spliterator();

87. **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.

88. What is the Java Module System introduced in Java 9, and it's 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.

89. What is a record in Java, and it's 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.

90. 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.

Syntax:
public sealed class Shape permits Circle, Square, Triangle { ... }
public final class Circle extends Shape { ... }
public non-sealed class Triangle extends Shape { ... }

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.

91. 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

92. 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 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.

93. 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.

94. 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.

95. 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.

96. What are the different ways to achieve synchronization in Java?

Answer:
If the interviewer asks this .
Tell these 5👇

- Synchronized methods
- Synchronized blocks
- Locks (such as ReentrantLock)
- Atomic variables

- Thread-safe collections (like ConcurrentHashMap, ConcurrentLinkedQueue, etc)

97. 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.

98. 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:
<u>Use CountDownLatch</u> when you have a one-time event, such as waiting for multiple services to start, and <u>CyclicBarrier</u> when you have repetitive tasks that need to synchronize at a certain point, like simulation steps.

99. How do you handle thread interruption in Java?

Answer:
<u>Thread Interruption:</u> Allows one thread to signal another thread to stop what it is doing and do something else.

<u>Handling:</u> Discuss checking the interrupted status using Thread.interrupted() or Thread.isInterrupted() and reacting appropriately, such as by throwing an InterruptedException or performing cleanup tasks.

<u>Significance:</u> 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.

100.   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.

101.   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.

102.   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().

103. 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.

104. 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).

105. What happens when an exception occurs inside a synchronized block?

Answer:
If an exception occurs inside a synchronized block, the lock is automatically released.

This can lead to unexpected behavior if not handled properly, as other threads might access the object in an inconsistent state.

106. 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.

107.   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
}
```

108. 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.

109. 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.

submit():
- Returns a Future.
- Works with both Runnable and Callable.
- Allows task management, cancellation, and result retrieval.

In short, use submit() for more control and feedback on task execution, execute() for simpler task execution without needing results or control.

110. What is the RejectedExecutionHandler in ThreadPoolExecutor? How can you customize it?

Answer:

RejectedExecutionHandler is an interface for handling tasks that cannot be executed due to the thread pool being full or shut down.

ThreadPoolExecutor provides several implementations like AbortPolicy, CallerRunsPolicy, DiscardPolicy, and DiscardOldestPolicy.

You can customize it by implementing the RejectedExecutionHandler interface to define custom rejection logic.

111. 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.

112. 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.

113. 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.

114.   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:

```java
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.

Example:

Profiling tools and benchmarks are used to measure and optimize throughput and latency in concurrent applications.

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.

Example:
```
@Benchmark
public void testMethod() {
        // Code to benchmark
    }
```

Chapter 15 : Junit

115. 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.

116. 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.

117. How to test Exception in JUnit?

Answer:
Use @Test(expected = ExceptionClass.class)
    or

newer JUnit 5's assertThrows()

118. How would you test asynchronous methods with JUnit?

Answer:

- JUnit 5: Use assertTimeout() or CompletableFuture for timeouts or completion checks.
- Awaitility: A library for more complex async assertions.

119. 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

120. 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.

121. 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).

122. What is difference between ApplicationContext and BeanFactory in the Spring framework?

Answer:

- <u>BeanFactory:</u> Basic container, lazy initialization, minimal features.

- <u>ApplicationContext:</u> Extends BeanFactory, eager initialization, includes additional features like AOP, event handling, and internationalization.

123. 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.

124.   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)

125.   What is the DispatcherServlet in Spring Framework?

Answer:

The DispatcherServlet is a core component in Spring MVC that acts as the front controller. It handles all incoming HTTP requests and delegates them to the appropriate handlers (controllers). It manages the request processing workflow, including view resolution and model population.

126. 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.

127. 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.

128. 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.

129. 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).

130. 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.

131.   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.

132. 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.

133. 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.

134. How to handle exceptions globally in a Spring Boot app?

Answer:

1. @ControllerAdvice:
   - Create a class annotated with @ControllerAdvice to catch exceptions across your application.

- Use @ExceptionHandler to specify which exceptions to handle.
2. @RestControllerAdvice:
   - Similar to @ControllerAdvice but for REST endpoints, automatically returning JSON responses.
3. AOP (Aspect-Oriented Programming):
   - Use aspects to catch exceptions in a cross-cutting manner, useful for more complex handling.
4. 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.

135. What is the difference between @Primary and @Qualifier?

Answer:
A concise comparison of @Primary and @Qualifier in Spring:

- @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:

- o Used for: Explicitly selecting a bean.
- o Applied on: Injection points.
- o Effect: Overrides @Primary for specific injection cases.

Key Points:
- @Qualifier takes precedence over @Primary.
- @Primary helps set defaults; @Qualifier allows specific choices.

136. How can you integrate Spring Boot Actuator with external monitoring and alerting systems such as Grafana?

Answer:
A concise guide on integrating Spring Boot Actuator with Grafana via Prometheus:

1. **Spring Boot Actuator Setup:**
   - o **Add Dependencies:** Include Actuator and Prometheus dependencies in your build tool.
   - o **Configure Actuator:** Enable the Prometheus endpoint in your application properties.
2. **Prometheus Setup:**
   - o **Install Prometheus:** Use Docker or traditional installation.
   - o **Configure Prometheus:** Set up prometheus.yml to scrape your app's /actuator/prometheus endpoint.

3. **Grafana Setup:**
    - o **Install Grafana:** Use Docker or install directly.
    - o **Add Prometheus as Data Source:** Configure Grafana to connect to your Prometheus instance.
    - o **Dashboards:** Import or create dashboards for your metrics.
4. **Alerting:**
    - o Create rules in Prometheus or directly in Grafana for notifications.

137. 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

138. 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.

139.   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();
 }
```

140. 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.

141. 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.

142. 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
}
```

143.   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.

144.   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.

- o Use When: Dealing with complex configurations or needing readability.

yaml
server:
 port: 8080

- **Properties:**
    - o Pros: Simple, widely supported, flat structure.
    - o Use When: Configurations are simple, or compatibility is key.

properties
server.port=8080

- Choose YAML for modern, complex setups.

- Choose Properties for simplicity or legacy systems.

145.   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"
      }
     ]
    }
```

146.   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.

147.   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.

148.   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.

149.   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.

- <u>Using Bean Names</u>: Explicitly specify which bean to inject using the @Resource(name="beanName") annotation.

- <u>Using @Inject with @Named</u>: In addition to @Qualifier, @Inject and @Named (from JSR-330) can be used for disambiguation.

150. 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}}
```

<u>ADDITIONAL POINTS</u>

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 diving 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.

Start building, stay consistent, and by the end of these projects, you'll have the confidence and skills needed to

tackle any Spring Boot challenge that comes your way

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.

151.   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.

152.   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.

153. 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.

154. What is the difference between merge and update in Hibernate?

Answer:
merge: Copies the state of a detached entity to the persistent context and returns a managed instance.

update: Updates the state of a detached entity to the persistent context but does not return a managed instance.

155. 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.

156.   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.

157.   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: Fetch associations in a single query.
2. @Fetch(FetchMode.JOIN): Use join fetching for associations.
3. @BatchSize(size = 10): Optimize collection fetching with batch size.
4. Entity Graphs: Define fetch graphs with @EntityGraph.

158.　　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).

159.　　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.

160.　　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);
```

161. 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.

162. 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);
```

163.   What is hibernate.dialect?

Answer:
hibernate.dialect defines the SQL dialect Hibernate uses to generate database-specific SQL queries.

Eg:
org.hibernate.dialect.MySQLDialect for MySQL.

164.   What is @GeneratedValue in Hibernate?

Answer:
@GeneratedValue specifies the strategy for generating primary key values, such as

AUTO, IDENTITY, SEQUENCE, or TABLE.

165.   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);
```

<u>load():</u>

- 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.

166.   What is the difference between first-level and second-level cache in Hibernate?

Answer:
<u>First-Level Cache:</u> 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.

<u>Second-Level Cache:</u> 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.

167. 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.

168. 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.

169.   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;
```

170. 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).

171. 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.

Example:

```
@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 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.

172. 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.

173. What Is Service Discovery and How Does It Work?

Answer:
Service discovery enables microservices to dynamically find and communicate with each other without hardcoding locations.

Implementation:

1. Tools:
    - Eureka: A service registry where microservices register themselves and discover other services.
    - Consul: Provides service registration, discovery, and health checks.
    - Zookeeper: Maintains a directory of service

instances.

2. Integration:
  - Registration: Microservices register with the service registry.
  - Discovery: Services query the registry to find the location of other services.
  - Example (Eureka): Use `@EnableEurekaClient` to register and `DiscoveryClient` to discover services.

Benefits:
  - Dynamic Scaling: Handles service instance changes automatically.
  - Decoupling: Avoids hardcoded service endpoints.

Challenges:
  - Consistency: Ensure the registry is up-to-date.
  - Latency: Minimize discovery latency.

174.   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.

175. 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.

176. 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.

177. 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).

<u>Benefits:</u>
- Scalability: Independent scaling of read and write services.
- Performance: Optimizes data access and modification.
- Flexibility: Different data models for reads and writes.

<u>Challenges:</u>
- Complexity: Increased system complexity and data synchronization.
- Eventual Consistency: Delays in data consistency.

<u>Example:</u>
In an e-commerce system, separate services handle order creation (commands) and order queries, allowing optimized performance for each.

178.   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.

<u>Implementation in Java Microservices:</u>

- 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.

179. 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.

180.   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.

181. 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.

182. 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.

183.   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.

184.   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:
Here's a streamlined 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.

185. 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.

186. **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:
Here's a streamlined 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.

Conclusion:

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.

Chapter 19: Java design patterns

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.

187. 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:

188.   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.

```java
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;
   }
}
```

189.   How to resolve issues with reflection and cloning in Singleton?

Answer:
Prevent Reflection:
Throw an exception in the constructor if the instance already exists.

```java
private Singleton() {
  if (instance != null) {
     throw new IllegalStateException("Instance already
created!");
  }
}
```

Prevent Cloning:

Override the `clone()` method to throw `CloneNotSupportedException`:

```java
@Override
protected Object clone() throws
CloneNotSupportedException {
  throw new CloneNotSupportedException();
}
```

190.   Which is the best way to create a Singleton: Enum or Bill Pugh Singleton?

Answer:
<u>Enum Singleton (Better for simplicity and safety)</u>

- 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.

<u>Bill Pugh Singleton (Better for flexibility)</u>

- 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.

<u>So you should:</u>
- 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.

<u>Common Use Cases:</u>
> - Creating complex objects with many optional components.
> - Constructing objects where the construction process is independent of the parts that make up the object.

<u>Example Scenario:</u>
Building a complex configuration object for a web application, where various settings and options can be specified.

<u>Interview Focus:</u>
> - Implementing the Builder pattern to provide a fluent API for object creation.
> - Discussing immutability and how to handle optional parameters.

191.   **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.

192.   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.

193.   How would you implement logging in Java using Decorators?

Answer:

Here's an implementation of logging using the Decorator Pattern in Java:

Interface:

```java
interface Processor {
    void process(String data);
}
```

Concrete Component:

```java
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.

194. 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.

Mastering these six design patterns will equip you with a solid foundation for solving complex design problems and handling common scenarios encountered in software development. Understanding their use cases, benefits, and potential pitfalls will help you not only in coding exercises but also in articulating your design decisions during interviews.

Chapter 20: 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.

195.   What is the Difference Between Clustered and Unclustered Indexes?

Answer:
Very commonly asked interview Question.

A <u>clustered index</u> determines the physical order of data in a table, meaning the rows are stored on disk in the same order as the index.

An <u>unclustered (or non-clustered) index</u>, 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.

196.   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.

197. 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.

198. What is the difference between a procedure and a function in a database?

Answer:
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.

199. 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 SLQ 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.

200.   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;

201.   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.

202.   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.

203.   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.

204.   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.

Chapter 21 : <u>Fifteen Methods that interviewers love</u>

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 22: 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.

- <u>Key Concepts:</u> Cache-aside pattern, write-through and write-back caches, TTL (Time-To-Live) settings.

<u>CV Tip:</u> 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.

- <u>Key Concepts:</u> 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 23: 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 occurance 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 (@ControllerAdvise), 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 occurance) --> 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 occurance.
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 processsed 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.

241

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 Controlller 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?

Chapter 24: 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 !!