

Java & Spring Knowledge Boost

Copyright © 2025 Titus. All rights reserved.

No part of this document may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without prior written permission of the author.

A brief about the document:

It is not just a list of questions, it is a practical guide to help you learn, revise, and connect theory with real-world scenarios.

- It contains, Java conceptual questions, output-based questions, practical scenarios and implementations, and stream-based problems.
- It also covers Spring conceptual topics, REST API discussions, and detailed Spring scenarios.
- You will also find testing questions, indirect and tricky questions, coding exercises of Array and String, as well as a structured set of questions arranged by experience level.
- Functional / Behavioural Questions.
- To support quick revision, the document also includes a Java cheat sheet and a Spring cheat sheet.
- On top of that, it also includes:
 - **Reference articles** for in-depth learning
 - **Video links** to understand internal implementations.
 - **Code examples** that show how concepts work.
 - **Real-time examples** provided where and when they are needed

| Category | Count | Page Number |
|------------------------------------|-------|-------------|
| Java Conceptual Questions | 106 | 19 |
| Output Based Java Questions | 67 | 149 |
| Java Scenarios and Implementations | 50 | 185 |
| Stream Based Scenario Questions | 50 | 213 |
| Spring Conceptual Questions | 130 | 228 |
| Rest API Questions | 30 | 380 |
| Spring Scenario Questions | 56 | 403 |
| Testing Questions | 25 | 457 |
| Indirect Questions | 366 | |

| Category | Count | Page Number |
|------------------------------------|------------|-------------|
| Array coding Questions | 25 | 470 |
| String coding Questions | 25 | 476 |
| Functional / Behavioural Questions | 31 | 482 |
| Experience wise Questions | | 484 |
| Java Cheat Sheet | | 491 |
| Spring Cheat Sheet | | 495 |
| Total | 961 | |

Interviewers usually focus on five major areas, and this document is designed to cover each one of them

1. Theory interview questions and Indirect questions.
2. Algorithm based interview questions.
3. Output based interview questions.
4. Scenario based interview questions.
5. Machine coding round.

Interview Tips (Technical):

1. Understand Core Java Concepts

Know the basics of Java well, including Object-Oriented Programming (inheritance, polymorphism, encapsulation, abstraction), data structures, algorithms, exception handling, multithreading, and the Collections Framework. Be ready to explain how these are used in your backend projects.

2. Learn Spring Framework Well

Be familiar with Spring and Spring Boot. Understand dependency injection, AOP, Spring Data JPA, Spring Security, and transaction management. Be able to explain how you used these features in your applications. Know the common Spring annotations.

3. Know RESTful API Design

Understand REST principles like statelessness, resource-based design, and proper use of HTTP methods (GET, POST, PUT, DELETE). Be ready to explain how to create secure, fast, and well-documented REST APIs using Spring.

4. Show Problem-Solving and System Design Skills

Be prepared to talk about tough technical problems you solved. Show that you can think about system design, performance, scalability, and fault tolerance. If you have experience with microservices or other architectures, explain it.

5. Be Good at Testing and Debugging

Explain how you test backend applications using unit tests, integration tests, or end-to-end tests. Mention tools like JUnit and Mockito. Show that you have written clean, maintainable, and well-tested code.

Interview Tips (General):

1. Discuss past projects using the STAR method

- Situation: Describe the context or challenge you faced.
- Task: State your specific responsibility in your project.
- Action: Explain the steps you took, decisions made, and collaboration involved to complete the task
- Result: Share the outcome and impact, preferably with metrics, e.g., reduced report generation time by 50%.

2. Communicate Clearly: Explain the concepts with examples or real time examples where you have used in your project

3. Highlight Collaboration Skills: Backend development involves working with frontend, DevOps, or QA teams. Share examples of teamwork and communication to show you fit in a collaborative environment.

4. Ask Smart Questions: Prepare 2–3 thoughtful questions about the team, architecture, or development practices. This shows interest and that you think beyond coding.

Click on any question to go straight to the answer.

Table of Contents

| | | |
|----|--|----|
| 1 | Why datatypes are important in Java? | 19 |
| 2 | What is reference type in java? | 20 |
| 3 | In how many ways we can create the object? | 20 |
| 4 | Explain about 4 pillars of Object-oriented programming (OOPS) (Frequent Question). | 21 |
| 5 | How is abstraction different from encapsulation? | 23 |
| 6 | Questions on Encapsulation? | 24 |
| 7 | Questions on Inheritance? | 25 |
| 8 | What is constructor in java? | 26 |
| 9 | Difference between == and .equals() in Java | 28 |
| 10 | Is Java Purely Object-Oriented? | 29 |
| 11 | Difference between final, finally, and finalize()? | 29 |

| | | |
|----|---|----|
| 12 | Is Java pass-by-value or pass-by-reference? | 30 |
| 13 | What is immutability, how to achieve it? | 31 |
| 14 | Why String is immutable in Java?..... | 32 |
| 15 | What is Object class in java and tell its methods? | 33 |
| 16 | Deep Copy vs Shallow Copy? | 34 |
| 17 | Explain the difference between String, StringBuilder, and StringBuffer. | 35 |
| 18 | What are Access Modifiers in Java? | 36 |
| 19 | What are Wrapper Classes in Java and Why Do We Need Them? | 38 |
| 20 | What is Autoboxing and Unboxing in Java? | 39 |
| 21 | What is Type casting? | 39 |
| 22 | What is a static keyword in Java (Commonly asked Question)? | 41 |
| 23 | What are Nested Classes?..... | 44 |
| 24 | What is the difference between method overloading and method overriding? (Commonly asked question) | 46 |
| 25 | What are the rules for overriding a method in Java? | 47 |
| 26 | What is Shadowing in java and its types?..... | 49 |
| 27 | What are super and this in Java? How are they used?..... | 50 |
| 28 | Explain about Abstract classes in Java | 52 |
| 29 | Explain about Interfaces in Java | 55 |
| 30 | Interface: Indirect questions | 55 |
| 31 | Interface vs Abstract Class | 57 |
| 32 | What is Serialization in java?..... | 57 |
| 33 | What is transient in java?..... | 59 |
| 34 | What is marker interface in java?..... | 59 |
| 35 | Explain about Association, Aggregation and Composition? | 61 |
| 36 | Why “Composition over inheritance”? | 62 |
| 37 | What is multi-tasking? | 63 |
| 38 | Difference between Process and Thread? | 64 |
| 39 | What is Java Memory Model (JMM)?..... | 65 |
| 40 | What is multi-threading how is it different from multi-tasking? | 66 |
| 41 | What is Thread, how to create threads in Java? | 67 |
| 42 | What are Runnable and Callable interfaces, and their differences? | 68 |

| | | |
|----|---|-----|
| 43 | What is Thread life cycle? | 70 |
| 44 | What is Executor Service and what are the uses??..... | 71 |
| 45 | What is Synchronization?..... | 72 |
| 46 | Difference Between Synchronized Method and Synchronized Block? | 73 |
| 47 | Difference of wait() , notify() and notifyAll()? | 75 |
| 48 | Difference of sleep(), wait() and join()? | 76 |
| 49 | Difference between Volatile and Atomic? | 77 |
| 50 | Difference between Future and CompletableFuture?..... | 78 |
| 51 | How Java handles exceptions? | 79 |
| 52 | What is the difference between checked and unchecked exceptions?..... | 80 |
| 53 | What is the difference between throw and throws? | 81 |
| 54 | What are custom exceptions and how create it?..... | 83 |
| 55 | Explain equals() and hashCode() contract?..... | 84 |
| 56 | What do you know about Collection in Java? | 86 |
| 57 | What is the difference between List, Set and Map? | 87 |
| 58 | ArrayList vs LinkedList? | 89 |
| 59 | ArrayList vs Vector? | 91 |
| 60 | HashMap vs TreeMap vs LinkedHashMap? | 91 |
| 61 | HashMap vs HashTable? | 94 |
| 62 | ConcurrentHashMap vs SynchronizedHashmap ? | 95 |
| 63 | Why ConcurrentHashMap won't allow null keys and null values?? | 96 |
| 64 | HashSet vs LinkedHashSet vs TreeSet? | 98 |
| 65 | Explain about Queue?..... | 100 |
| 66 | Explain different types of Queue?..... | 100 |
| 67 | When will you use each queue?..... | 100 |
| 68 | How ArrayList works internally?..... | 101 |
| 69 | How HashMap works internally? | 102 |
| 70 | How HashSet works internally?..... | 104 |
| 71 | How LinkedHashMap and LinkedHashSet works internally? | 105 |
| 72 | How TreeMap works internally? | 105 |
| 73 | How TreeSet works internally?..... | 108 |
| 74 | How ConcurrentHashMap works internally? | 108 |

| | | |
|-----|---|-----|
| 75 | Iterator vs List Iterator? | 110 |
| 76 | Fail-Fast vs Fail-Safe? | 111 |
| 77 | How FAIL-FAST iterator works? why it throws concurrent modification exception? | 112 |
| 78 | How FAIL-SAFE iterator works? why it does NOT throw concurrent modification exception?..... | 113 |
| 79 | Comparable and comparator, explain their differences? | 113 |
| 80 | List.of() vs Collections.unmodifiableList? | 114 |
| 81 | What do you know about generics in Java?..... | 115 |
| 82 | What is Garbage Collection?..... | 117 |
| 83 | What are the different types of Garbage Collectors?..... | 117 |
| 84 | How Garbage Collections works internally? | 118 |
| 85 | Difference between WeakReference and SoftReference?..... | 119 |
| 86 | Can you list out the Java 8 features? | 120 |
| 87 | What are default methods and why are they are introduced? | 121 |
| 88 | Why Java 8 introduced static methods to the Interfaces? | 122 |
| 89 | What is optional and it uses? | 123 |
| 90 | What is functional interface and why they are introduced in Java? | 125 |
| 91 | Explain different types of functional interfaces? | 127 |
| 92 | Explain about Lambda expression? | 129 |
| 93 | Difference between lambda expression and anonymous class? | 129 |
| 94 | Explain about Method references and uses?..... | 131 |
| 95 | Explain about Streams? | 131 |
| 96 | Can you list of streams operators you have used or you know?..... | 134 |
| 97 | Difference between Streams and Collections? | 136 |
| 98 | Difference between map and flatMap? | 136 |
| 99 | How reduce works? | 137 |
| 100 | forEach vs collect()?..... | 137 |
| 101 | How can you optimize a Stream pipeline for better performance? | 138 |
| 102 | Difference between sequential stream and parallel stream?..... | 138 |
| 103 | How do you handle exceptions in streams? | 139 |
| 104 | What are disadvantages or when not to Use Java Streams? | 141 |
| 105 | What's the difference between "this" in a lambda vs. an anonymous? | 142 |

| | | |
|-----|--|-----|
| 106 | Features of different versions? | 143 |
| 107 | Output-Based 1: How many String objects will be created in memory from below code? | 149 |
| 108 | Output-Based 2: How many String objects will be created in memory from below code? | 150 |
| 109 | Output-Based 3: String ==? | 150 |
| 110 | Output-Based 4: String == and equals? | 150 |
| 111 | Output-Based 5: String concatenation? | 151 |
| 112 | Output-Based 6: valueOf()? | 151 |
| 113 | Output-Based 7: Substring()? | 151 |
| 114 | Output-Based 8: String == , equals and intern()? | 152 |
| 115 | Output-Based 9: On StringBuilder. | 152 |
| 116 | Output-Based 10: On StringBuilder, Checking the reference and content. | 152 |
| 117 | Output-Based 11: Array reference comparison vs. content comparison. | 153 |
| 118 | Output-Based 12: Pass by Value..... | 153 |
| 119 | Output-Based 13: Passing the reference. | 154 |
| 120 | Output-Based 14: Passing the reference and reassigning. | 155 |
| 121 | Output-Based 15: Passing reference of the array..... | 155 |
| 122 | Output-Based 16: Passing reference of the array's copy..... | 156 |
| 123 | Output-Based 17: Final variable. | 156 |
| 124 | Output-Based 18: Final array..... | 156 |
| 125 | Output-Based 19: Final Object. | 157 |
| 126 | Output-Based 20: Multiple Exceptions. | 157 |
| 127 | Output-Based 21: Try with return and finally..... | 157 |
| 128 | Output-Based 22: Try and Finally with returns. | 158 |
| 129 | Output-Based 23: Execution flow with static..... | 158 |
| 130 | Output-Based 24: Execution flow with instance. | 159 |
| 131 | Output-Based 25: Overriding: Method overriding basics. | 160 |
| 132 | Output-Based 26: Overriding: Variable hiding. | 160 |
| 133 | Output-Based 27: Overriding: Constructor Execution Order. | 161 |
| 134 | Output-Based 28: Overriding: Static method hiding. | 161 |
| 135 | Output-Based 29: Overriding: Accessing Child-Specific Method. | 162 |

| | | |
|-----|--|-----|
| 136 | Output-Based 30: String Concatenation. | 163 |
| 137 | Output-Based 31: Overriding: Type Casting 1. | 163 |
| 138 | Output-Based 32: Overriding: Type Casting 2. | 164 |
| 139 | Output-Based 33: Overriding: Field Access. | 164 |
| 140 | Output-Based 34: Overriding: Private Method in Parent. | 165 |
| 141 | Output-Based 35: Overriding: Protected Method Access | 165 |
| 142 | Output-Based 36: Overriding: Parent public, child private | 166 |
| 143 | Output-Based 37: Overriding: Checked Exceptions.... | 167 |
| 144 | Output-Based 38: Overriding: Un-Checked Exceptions..... | 167 |
| 145 | Output-Based 40: Overriding: Method Call in Constructor | 168 |
| 146 | Output-Based 41: Overriding: Method Call in Constructor with static | 169 |
| 147 | Output-Based 42: Interface Default Method vs Class Method | 169 |
| 148 | Output-Based 43: Overriding: Diamond Problem (Interface) | 170 |
| 149 | Output-Based 44: Abstract Class: Main Method..... | 171 |
| 150 | Output-Based 45: Abstract Class: Private Method..... | 171 |
| 151 | Output-Based 46: Abstract Class: Multiple abstract classes in chain..... | 171 |
| 152 | Output-Based 47: Overloading: Type Promotion | 172 |
| 153 | Output-Based 48: Overloading: With Varargs vs Exact Match..... | 172 |
| 154 | Output-Based 49: Overloading: Autoboxing vs Widening | 173 |
| 155 | Output-Based 50: Overloading: Primitive Overloading | 173 |
| 156 | Output-Based 51: Overloading and Inheritance | 174 |
| 157 | Output-Based 52: Overloading: Reference vs Object Type..... | 175 |
| 158 | Output-Based 53: Super and This: Constructor | 175 |
| 159 | Output-Based 54: Super and This: Constructor | 176 |
| 160 | Output-Based 55: Super and This: Constructor Chaining | 176 |
| 161 | Output-Based 56: Super and This: Variable Access | 177 |
| 162 | Output-Based 57: Functional Interface: Can a functional interface have default methods?..... | 178 |
| 163 | Output-Based 58: Functional Interface: Is Child class is a functional interface? | 178 |
| 164 | Output-Based 59: Functional Interface: Will the code below compile?..... | 179 |
| 165 | Output-Based 60: Functional Interface: Is this a valid Functional Interface? | 180 |
| 166 | Output-Based 61: Thread: Starting the thread twice? | 180 |

| | | |
|-----|--|-----|
| 167 | Output-Based 62: Thread: Call run() and start()? | 181 |
| 168 | Output-Based 63: Thread: Join ? | 181 |
| 169 | Output-Based 64: Thread: Create a deadlock | 182 |
| 170 | Output-Based 65: Instance vs Static ? | 183 |
| 171 | Output-Based 66: Static variable declaration? | 184 |
| 172 | Output-Based 67: Tricky question on runnable | 184 |
| 173 | Java Scenario 1: If – Else Conditions or Switch. | 185 |
| 174 | Java Scenario 2: ATM Operations - Switch. | 185 |
| 175 | Java Scenario 3: Traffic Light Action - Switch. | 186 |
| 176 | Java Scenario 4: E-commerce Cart Total Calculation (For loop). | 186 |
| 177 | Java Scenario 5: Password Strength Checker. | 186 |
| 178 | Java Scenario 6: Printing Even and Odd Numbers with Two Threads | 187 |
| 179 | Java Scenario 7: Sequential Execution of Two Threads Printing Numbers (1,2,3...) and Letters(A,B,C,...) in Java. So that it prints (A,1B,2...) | 188 |
| 180 | Java Scenario 8: Producer Consumer Problem | 188 |
| 181 | Java Scenario 9: Simulating the deadlock | 190 |
| 182 | Java Scenario 10: Increment the shared counter with threads | 190 |
| 183 | Java Scenario 11: Bank Account Withdrawal | 191 |
| 184 | Java Scenario 12: CompletableFuture Scenario | 191 |
| 185 | Java Scenario 13: Payment Gateway Integration | 192 |
| 186 | Java Scenario 14: Notification Service | 193 |
| 187 | Java Scenario 15: File Export System | 193 |
| 188 | Java Scenario 16: Authentication Strategies | 193 |
| 189 | Java Scenario 17: Employee Payroll System | 194 |
| 190 | Java Scenario 18: Bank Account Types | 194 |
| 191 | Java Scenario 19: Custom Exceptions | 195 |
| 192 | Java Scenario 20: Closing resources | 196 |
| 193 | Java Scenario 21: Creating Custom Exception | 196 |
| 194 | Java Scenario 22: Global Math Operations | 197 |
| 195 | Java Scenario 23: Logging | 198 |
| 196 | Java Scenario 24: Creating Custom Functional Interface | 198 |
| 197 | Java Scenario 25: Collections - 1 | 198 |

| | | |
|-----|---|-----|
| 198 | Java Scenario 26: Collections - 2 | 199 |
| 199 | Java Scenario 27: Collections - 3 | 199 |
| 200 | Java Scenario 28: Collections - 4 | 199 |
| 201 | Java Scenario 29: Collections - 5 | 200 |
| 202 | Java Scenario 30: Collections - 6 | 200 |
| 203 | Java Scenario 31: Collections - 7 | 201 |
| 204 | Java Scenario 32: Collections – 8 | 201 |
| 205 | Java Scenario 33: LRU Cache Implementation | 202 |
| 206 | Java Scenario 34: Task Scheduler by Priority..... | 202 |
| 207 | Java Scenario 35: Detecting Duplicate Usernames..... | 203 |
| 208 | Java Scenario 36: Auto-Suggest in Search..... | 203 |
| 209 | Java Scenario 37: Stack – Parentheses Balancer | 204 |
| 210 | Java Scenario 38: Thread safe list | 205 |
| 211 | Java Scenario 39: Counting API Requests Per User | 205 |
| 212 | Java Scenario 40: ArrayList vs LinkedList..... | 206 |
| 213 | Java Scenario 41: Large file handling | 206 |
| 214 | Java Scenario 42: Memory Leaks..... | 207 |
| 215 | Java Scenario 43: Optimize the GC | 207 |
| 216 | Java Scenario 44: Best coding practices..... | 208 |
| 217 | Java Scenario 45: Performance Optimization..... | 209 |
| 218 | Java Scenario 46: Implementing the Retry..... | 210 |
| 219 | Java Scenario 47: Implementing stack using arrays..... | 210 |
| 220 | Java Scenario 48: Implementing stack using 2 Queues | 211 |
| 221 | Java Scenario 49: Implement Queue using Array | 212 |
| 222 | Java Scenario 50: Implement Custom HashMap | 213 |
| 223 | Streams 1: Remove duplicates without distinct() | 213 |
| 224 | Streams 2: Get duplicates | 213 |
| 225 | Streams 3: Sort in descending order | 214 |
| 226 | Streams 4: Practice Questions on Sort..... | 214 |
| 227 | Streams 5: Find Max Number in a list | 215 |
| 228 | Streams 6: Check all numbers even or not | 215 |
| 229 | Streams 7: Numbers starting with 1 | 216 |

| | | |
|-----|---|-----|
| 230 | Streams 8: Find Palindrome Strings | 217 |
| 231 | Streams 9: Find the longest word in a list | 217 |
| 232 | Streams 10: List of the questions on filter (For Practice)..... | 217 |
| 233 | Streams 11: Merge two unsorted arrays into single sorted array..... | 218 |
| 234 | Streams 11: Get top 3 elements from the list | 218 |
| 235 | Streams 12: Get 3 rd highest element from the list | 219 |
| 236 | Streams 13: Check if two strings are anagrams or not | 219 |
| 237 | Streams 14: Sum of all digits in a number | 220 |
| 238 | Streams 15: Common elements between two arrays | 220 |
| 239 | Streams 16: Reverse each word of a string | 220 |
| 240 | Streams 17: Count the number of occurrences of a given String. | 221 |
| 241 | Streams 18: Convert the list of sentences into unique words. | 221 |
| 242 | Streams 19: More Questions on flatMap..... | 222 |
| 243 | Streams 20: Find all the Longest words in a list | 222 |
| 244 | Streams 21: Questions on reduce | 222 |
| 245 | Streams 22: Find the longest common prefix using Java streams: | 223 |
| 246 | Streams 23: Max Product in a given array | 223 |
| 247 | Streams 23: First non-repeating character in a String | 224 |
| 248 | Streams 24: Most Repeated Number in a given list. | 225 |
| 249 | Streams 25: Frequency of words or count of each word in a String | 225 |
| 250 | Streams 26-35: Scenario based questions on Student Data..... | 226 |
| 251 | Streams 36-45: Scenario based questions on Employee Data..... | 226 |
| 252 | Stream 46-50: Scenario based questions on City Data..... | 227 |
| 253 | Spring 1: What is Spring boot? | 228 |
| 254 | Spring 2: What are the advantages of Spring boot?..... | 228 |
| 255 | Spring 3: What are embedded containers / servers in Spring and what is the default one? how to add Jetty for example? | 229 |
| 256 | Spring 4: Difference between Spring and Spring Boot?..... | 229 |
| 257 | Spring 5: Explain Spring MVC architecture?..... | 230 |
| 258 | Spring 6: What is @SpringBootApplication?..... | 231 |
| 259 | Spring 7: What is Dependency Injection? | 231 |
| 260 | Spring 8: What is IOC Container? | 233 |

| | | |
|-----|---|-----|
| 261 | Spring 9: What is the difference between @Component, @Service, and @Repository? | 235 |
| 262 | Spring 10: What is application.properties file? | 236 |
| 263 | Spring 11: What are Spring boot starters?..... | 238 |
| 264 | Spring 12: How many ways dependency injection can be done or Types of Dependency Injection?..... | 238 |
| 265 | Spring 13: What do you mean by Annotation-based container configuration? | 239 |
| 266 | Spring 14: Explain about @Autowired? | 240 |
| 267 | Spring 15: Explain about @Qualifier and @Primary? | 241 |
| 268 | Spring 16: Difference between application.properties vs application.yml? | 243 |
| 269 | Spring 17: What is the use of @Value / How to load config values dynamically? | 243 |
| 270 | Spring 18: What is the use of @ConfigurationProperties / How to load config values dynamically? | 244 |
| 271 | Spring 19: How do profiles work in Spring boot, Explain about @Profile? | 244 |
| 272 | Spring 20: How spring auto configuration works internally?..... | 246 |
| 273 | Spring 21: Difference between @Bean vs @Configuration | 247 |
| 274 | Spring 22: Difference between @Bean vs @Component | 249 |
| 275 | Spring 23: What is Spring Bean | 249 |
| 276 | Spring 24: What are bean scopes? | 250 |
| 277 | Spring 25: What is the bean life cycle? | 253 |
| 278 | Spring 26: Can we create beans conditionally, if yes how? | 255 |
| 279 | Spring 27: How does @Conditional works internally? | 257 |
| 280 | Spring 28: Difference between CommandLineRunner and ApplicationRunner? | 257 |
| 281 | Spring 29: How to handle Circular dependencies? | 258 |
| 282 | Spring 30: What is eager loading and lazy loading of beans?? | 260 |
| 283 | Spring 31: What is Dispatcher Servlet?..... | 261 |
| 284 | Spring 32: How embedded server works in Spring boot? | 262 |
| 285 | Spring 33: What is ORM? | 263 |
| 286 | Spring 34: What is Hibernate and how it works? | 264 |
| 287 | Spring 35: What is JPA and how does it different from Hibernate? | 265 |
| 288 | Spring 36: JDBC vs Hibernate vs JPA?..... | 268 |
| 289 | Spring 37: How do you use @Entity, @Table, @Column in JPA? | 268 |

| | | |
|-----|--|-----|
| 290 | Spring 38: What are best practices or rules while creating the Entity? | 270 |
| 291 | Spring 39: What are the different types of relationships in JPA and explain @OneToOne with example..... | 270 |
| 292 | Spring 40: Explain @OneToMany or @ManyToOne with example | 272 |
| 293 | Spring 41: Explain @ManyToMany with example | 274 |
| 294 | Spring 42: What is cascading in JPA and when would you use it? | 275 |
| 295 | Spring 43: How does eager loading and lazy loading works in JPA? | 276 |
| 296 | Spring 44: Explain Repositories, like CrudRepository, JpaRepository, and PagingAndSortingRepository? | 278 |
| 297 | Spring 45: What is JPQL and how does it different from HQL?..... | 280 |
| 298 | Spring 46: What is @Query (Custom Queries) and why should we use them??..... | 280 |
| 299 | Spring 47: What is @NamedQuery and @NamedNativeQuery?..... | 281 |
| 300 | Spring 48: What is the use of @Transactional?..... | 282 |
| 301 | Spring 49: What is propagation in Transaction?..... | 285 |
| 302 | Spring 50: How transaction works under the hood?..... | 288 |
| 303 | Spring 51: What is N+1 problem how do you fix it? | 290 |
| 304 | Spring 52: Difference Between JOIN and JOIN FETCH? | 290 |
| 305 | Spring 53: How does locking works in Spring JPA? | 291 |
| 306 | Spring 54: Explain JPA Architecture? | 293 |
| 307 | Spring 56: what is Persistence context in JPA? | 298 |
| 308 | Spring 57: what are different states on an Entity? | 300 |
| 309 | Spring 58: How Cache works in JPA? | 300 |
| 310 | Spring 59: How to connect to multiple databases?..... | 302 |
| 311 | Spring 60: How do you create Custom repository in Spring JPA? | 306 |
| 312 | Spring 61: What is connection pooling, how it helps in Spring?..... | 306 |
| 313 | Spring 62: What do you know about @NoRepositoryBean? | 307 |
| 314 | Spring 63: What is Spring Security? | 308 |
| 315 | Spring 64: How do you enable Security in Spring applications? | 309 |
| 316 | Spring 65: What is @EnableWebSecurity? | 310 |
| 317 | Spring 66: What are the key features of Spring Security? | 311 |
| 318 | Spring 67: What is authentication and different types of it? | 311 |
| 319 | Spring 68: What is the authentication flow in Spring security? | 312 |

| | | |
|-----|--|-----|
| 320 | Spring 69: What is JWT and how does it differ from Basic auth? | 314 |
| 321 | Spring 70: What is structure of the JWT? | 316 |
| 322 | Spring 71: How server validates the JWT token? | 317 |
| 323 | Spring 72: How to implement JWT authentication in Spring boot? | 318 |
| 324 | Spring 73: How does SecurityFilterChain works and list down the filters? | 319 |
| 325 | Spring 74: Internals on SecurityFilterChain? | 321 |
| 326 | Spring 75: What is UserDetailsService and how it works? | 322 |
| 327 | Spring 76: Explain about Password Encryption and why it is important? | 323 |
| 328 | Spring 77: What is CSRF and how it works in Spring? | 324 |
| 329 | Spring 78: What is CORS, how we need to resolve it?..... | 325 |
| 330 | Spring 79: What is Outh2 and how does it differ from JWT? | 325 |
| 331 | Spring 80: How do you implement Oauth2 in Spring? | 326 |
| 332 | Spring 81: What is role-based access?..... | 326 |
| 333 | Spring 82: Difference between Role and Grant?..... | 328 |
| 334 | Spring 83: What is method level security? | 328 |
| 335 | Spring 84: What is Spring Boot Actuator, and why is it useful? | 329 |
| 336 | Spring 85: How do you enable and configure Actuator endpoints? | 329 |
| 337 | Spring 86: List out some important Actuator endpoints? | 330 |
| 338 | Spring 87: What is Micrometer?..... | 330 |
| 339 | Spring 88: How to create Custom Endpoint in Actuator?..... | 330 |
| 340 | Spring 89: How to secure Actuator endpoints and why to secure? | 332 |
| 341 | Spring 90: What is AOP? | 332 |
| 342 | Spring 91: What are the core components of AOP?..... | 334 |
| 343 | Spring 92: How proxy works in AOP?..... | 336 |
| 344 | Spring 93: Which classes or methods in Spring cannot be declared as final, and why?? | 337 |
| 345 | Spring 94: What is Self-Invocation Problem? | 338 |
| 346 | Spring 95: How to do Scheduling in Spring boot? | 339 |
| 347 | Spring 96: What is the difference between fixedDelay and fixedRate? | 340 |
| 348 | Spring 97: How can you make tasks run in parallel? | 341 |
| 349 | Spring 98: How can we handle scheduled task failures? | 342 |
| 350 | Spring 99: How to run a task in the background in Spring boot? | 342 |

| | | |
|-----|--|-----|
| 351 | Spring 100: How @Async works behind the scenes? | 344 |
| 352 | Spring 101: Can you run Database transactional tasks in Async? | 345 |
| 353 | Spring 102: How cache works in spring boot? | 345 |
| 354 | Spring 103: How @cachable works? | 346 |
| 355 | Spring 104: What is the importance of the key while caching? | 346 |
| 356 | Spring 105 Can you cache based on the condition?..... | 346 |
| 357 | Spring 106: How to update the cache?..... | 347 |
| 358 | Spring 107: How to delete the cache?..... | 347 |
| 359 | Spring 108: How cache work behind the scenes? | 348 |
| 360 | Spring 109: How CocurrentMapCacheManager works? | 348 |
| 361 | Spring 110: What is proxy and how does it work Spring? | 348 |
| 362 | Spring 111: How a CGLIB Proxy works with @Bean? | 350 |
| 363 | Spring 112: What is filter in Spring boot and how it will used? | 350 |
| 364 | Spring 113: How do you create a custom filter? | 351 |
| 365 | Spring 114: What is interceptor and its uses?..... | 352 |
| 366 | Spring 115: How to create an Interceptor?..... | 353 |
| 367 | Spring 116: Difference between Filter and Interceptor? | 354 |
| 368 | Spring 117: Have you written any custom Annotation, if yes how you have written it? 355 | |
| 369 | Spring 118: @Configuration vs @Component? | 356 |
| 370 | Spring 119: What are best techniques to handle exceptions in Spring? | 357 |
| 371 | Spring 120: How Database indexing improves the performance? | 362 |
| 372 | Spring 121: What is externalization and how would you achieve it? | 362 |
| 373 | Spring 122: Auditing in Spring JPA? | 363 |
| 374 | Spring 123: Explain application flow of the Spring boot or What will happen when we call run ()? | 366 |
| 375 | Spring 124: What are different strategies of ID generation in Hibernate? | 367 |
| 376 | Spring 125: What is singleton? | 369 |
| 377 | Spring 126: How Cross Origin works in Spring? | 372 |
| 378 | Spring 127: How @Retryable works in Spring? | 373 |
| 379 | Spring 128: How proxyMode works in Spring and how does it help? | 375 |
| 380 | Spring 129: How to deploy Spring boot application using jar or war? | 376 |

| | | |
|-----|---|-----|
| 381 | Spring 130: Explain different kind of paging techniques? | 378 |
| 382 | Rest 1: What is REST API and what are its uses?..... | 380 |
| 383 | Rest 2: What is difference between PUT and PATCH? | 382 |
| 384 | Rest 3: Can you tell what are the annotations you used while developing REST APIs? 383 | |
| 385 | Rest 4: Difference between @RestController and @Controller?..... | 384 |
| 386 | Rest 5: Difference between @RequestMapping and @GetMapping?..... | 385 |
| 387 | Rest 6: Difference between @PathVariable and @RequestParam?..... | 386 |
| 388 | Rest 7: What is Response Entity and how it will be used?..... | 387 |
| 389 | Rest 8: Different types of HTTP status codes?..... | 388 |
| 390 | Rest 9: How Spring send JSON by default / How Spring converts Java object to JSON while returning response from API? | 389 |
| 391 | Rest 10: Different types of headers? | 390 |
| 392 | Rest 11: What is content negotiation? | 391 |
| 393 | Rest 12: How can Spring send XML if default is JSON?..... | 392 |
| 394 | Rest 13: What is Idempotency?..... | 393 |
| 395 | Rest 14: Why Patch is not idempotent?..... | 393 |
| 396 | Rest 15: Can we make POST as idempotent?..... | 393 |
| 397 | Rest 16: How do REST APIs handle versioning, and why is it important?..... | 394 |
| 398 | Rest 17: How do you validate the Request from client in Spring boot? | 394 |
| 399 | Rest 18: How to create custom validators in spring? | 395 |
| 400 | Rest 19: How do you implement caching in REST APIs?..... | 397 |
| 401 | Rest 20: How can you handle rate limiting in REST API? | 398 |
| 402 | Rest 21: Rate limiting vs API throttling?..... | 398 |
| 403 | Rest 22: What are different types of authentications used in REST APIs?..... | 398 |
| 404 | Rest 23: Difference between URI and URL? | 398 |
| 405 | Rest 24: How do you implement pagination in REST API?..... | 399 |
| 406 | Rest 25: What is the difference between synchronous and asynchronous communication in RESTful web services? | 399 |
| 407 | Rest 26: What are the best practices while designing the REST API? | 400 |
| 408 | Rest 27: How do you document the APIs and their details?..... | 400 |
| 409 | Rest 28: SOAP VS REST? | 400 |

| | | |
|-----|---|-----|
| 410 | Rest 29: How do you invoke the other APIs or Inter-service communication in a microservice in Spring? | 401 |
| 411 | Rest 30: How do you implement search functionality?..... | 402 |
| 412 | Spring Scenario 1: Get by User ID or Email. | 403 |
| 413 | Spring Scenario 2: Debug the slow API. | 403 |
| 414 | Spring Scenario 3: Performance optimization..... | 404 |
| 415 | Spring Scenario 4: DB Migrations | 405 |
| 416 | Spring Scenario 5: File upload and download..... | 405 |
| 417 | Spring Scenario 6: Asynchronous Processing..... | 407 |
| 418 | Spring Scenario 7: Handling Configuration | 407 |
| 419 | Spring Scenario 8: Apply configuration without reboot | 409 |
| 420 | Spring Scenario 9: Large file handling..... | 411 |
| 421 | Spring Scenario 10: Caching. | 412 |
| 422 | Spring Scenario 11: Validations. | 413 |
| 423 | Spring Scenario 12: Scheduling the task. | 414 |
| 424 | Spring Scenario 13: Dependency Injection. | 415 |
| 425 | Spring Scenario 14: Custom Bean..... | 416 |
| 426 | Spring Scenario 15: Inject bean on condition. | 417 |
| 427 | Spring Scenario 16: Inject bean on multiple conditions. | 417 |
| 428 | Spring Scenario 17: Dynamic bean selection, based on parameter from client..... | 418 |
| 429 | Spring Scenario 18: Load initial cache data..... | 420 |
| 430 | Spring Scenario 19: Logging in Spring boot..... | 421 |
| 431 | Spring Scenario 20: Monitoring the Spring application..... | 422 |
| 432 | Spring Scenario 21: Post idempotency | 422 |
| 433 | Spring Scenario 22: Custom Annotation + Interceptor..... | 423 |
| 434 | Spring Scenario 23: Rate Limiting | 425 |
| 435 | Spring Scenario 24: Custom Query method..... | 426 |
| 436 | Spring Scenario 25: Custom Query | 426 |
| 437 | Spring Scenario 26: Lazy vs Eager / N+1 problem | 427 |
| 438 | Spring Scenario 27: save() or saveAll()..... | 428 |
| 439 | Spring Scenario 28: Partial Update | 429 |
| 440 | Spring Scenario 29: Soft Deletes..... | 430 |

| | | |
|-----|--|-----|
| 441 | Spring Scenario 30: Auditing | 431 |
| 442 | Spring Scenario 31: Handling null values | 432 |
| 443 | Spring Scenario 32: Transaction rollback failed..... | 433 |
| 444 | Spring Scenario 33: Read-only transaction | 434 |
| 445 | Spring Scenario 34: Event driven..... | 435 |
| 446 | Spring Scenario 35: Improve initial loading time or Cold Start Optimization Techniques | 435 |
| 447 | Spring Scenario 36: Public some APIs | 436 |
| 448 | Spring Scenario 37: Only ADMIN operation..... | 437 |
| 449 | Spring Scenario 38: Monitoring the Application | 438 |
| 450 | Spring Scenario 39: Inter-service communication..... | 438 |
| 451 | Spring Scenario 40: Searching Million Records | 439 |
| 452 | Spring Scenario 41: Process Million Records (NOTE: It is about processing not searching) | 439 |
| 453 | Spring Scenario 42: OutOfMemory due to huge data..... | 441 |
| 454 | Spring Scenario 43: Writing to DB by two threads | 442 |
| 455 | Spring Scenario 44: Parallel API call..... | 443 |
| 456 | Spring Scenario 45: Secure the App..... | 444 |
| 457 | Spring Scenario 46: Prevent SQL Injection | 445 |
| 458 | Spring Scenario 47: Prevent XSS attacks | 446 |
| 459 | Spring Scenario 48: Best Practices while designing the REST API | 447 |
| 460 | Spring Scenario 49: Design the Entity Relationships | 448 |
| 461 | Spring Scenario 50: End to End High level Design | 450 |
| 462 | Spring Scenario 51: Measuring Execution Time | 453 |
| 463 | Spring Scenario 52: Centralized Exception Handling | 454 |
| 464 | Spring Scenario 53: Transaction Management..... | 454 |
| 465 | Spring Scenario 54: Validate Tenant for all service methods | 455 |
| 466 | Spring Scenario 55: Custom Annotations | 455 |
| 467 | Spring Scenario 56: Load huge data where pagination is not possible | 456 |
| 468 | Testing: Questions on Testing | 457 |
| 469 | Array Coding Problems | 470 |
| 470 | String Coding Problems..... | 476 |

| | | |
|-----|--|-----|
| 471 | Functional/Behavioural Questions | 482 |
| 472 | Junior Level Developer..... | 484 |
| 473 | Mid-Level Developer..... | 486 |
| 474 | Senior Developer 7+ | 489 |
| 475 | Java Cheat Sheet..... | 491 |
| 476 | Spring Cheat Sheet | 495 |
| 477 | November | 502 |
| 478 | December..... | 502 |

Java course reference link: [\(1\) Java Tutorial For beginners - YouTube](#)

Java 8 reference link (In depth):

[Lambda Expressions in Java 8 | Java 8 Features | Crash Course](#) 

Spring boot course reference link:

To learn Spring on a high level go through this playlist: [Spring 6 and Spring Boot Tutorial for Beginners - YouTube](#) It contains, Spring boot, Web, JPA and security.

1 Why datatypes are important in Java?

Java is a **strongly-typed language**, which means we must tell the compiler what kind of data we are working with. Here is why data types matter:

1. Memory and Performance

Different data types use different sizes of memory.

Example: int uses less memory than long or double.

Using the right type helps our program run faster and use less memory.

2. To catch errors

Java checks the code at compile time.

If we try to assign a number to a String, it throws an error *before* the program runs.

3. Only Valid Operations Allowed

Each type supports specific actions.

- We can do math with numbers (int, double)
- We can use string methods with String
- We can't mix them up randomly
This keeps the logic clean and predictable.

4. Easier to Read and Maintain

When we declare a variable like int age = 25; it is clear that age is meant to hold number.

That makes the code easier to read, understand, and maintain.

5. Helps Represent Data Correctly

Java gives us different types for different kinds of data:

- Whole numbers: byte, short, int, long
- Decimals: float, double
- Characters: char
- True/false: boolean

Choosing the right one helps us represent and process the data properly.

Indirect Questions:

1. What would happen if you try to add a String and an int in Java? Why?

If we add a String and an int, Java converts the int to a String and adds.

"Age" + 25 gives "Age25".

2. Is it good use String isActive = "true"; instead of boolean isActive = true?

Using String isActive = "true" instead of a boolean is not ideal, it can lead to bugs since any string can be assigned, not just true or false.

3. Is Java strongly typed language?

Yes, Java is a strongly typed language. Variables must be declared with a specific data type. Before a variable can be used, its data type (e.g., int, String, double) must be explicitly specified during declaration.

2 What is reference type in java?

In Java, reference types (non-primitive types) store memory addresses (references) to objects rather than the actual data. Unlike primitive types, they can hold null, support methods, and are stored in the heap memory.

1. What's the default value of an object member in a class?

Default value of object is null.

2. Is String a primitive or object?

Object

3. What happens when you assign one object to another?

Both references now point to the same object.

3 In how many ways we can create the object?

1. Using new keyword

```
class Car {
    Car() {
        System.out.println("Car constructor called");
    }
}
Car car = new Car(); // Classic object creation
```

2. Using clone():

```
class Car implements Cloneable {
    int speed = 100;
```

```

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
Car car1 = new Car();
Car car2 = (Car) car1.clone(); // Constructor NOT called

```

3. Using Deserialization (ObjectInputStream)

```

class Car implements Serializable {
    int speed = 100;
}
ObjectInputStream in = new ObjectInputStream(new FileInputStream("car.ser"));
Car car = (Car) in.readObject(); // Constructor NOT called

```

4. Using Reflection (Constructor.newInstance())

```

class Car {
    Car() {
        System.out.println("Reflection constructor called");
    }
}
Constructor<Car> cons = Car.class.getConstructor();
Car car = cons.newInstance(); // Constructor IS called

```

Indirect Questions

1. In which ways of object creation, the constructor doesn't get called?

Constructors don't get called during cloning (clone()) and deserialization (ObjectInputStream.readObject()), as they copy the object state directly.

2. how to create an object without using a new keyword?

We can create an object without new by using clone(), deserialization, or reflection

3.Which interface is must to clone the object?

Cloneable interface

4 Explain about 4 pillars of Object-oriented programming (OOPS) (Frequent Question).

1. Inheritance

Inheritance allows one class to acquire properties (fields) and behaviours (methods) of another. It promotes reusability and a natural hierarchy.

```

class Animal {
    void sound() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}

```

```
class InheritanceDemo {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound(); // inherited from Animal
        dog.bark(); // defined in Dog
    }
}
```

Interview Tip:

Always mention Java supports single inheritance with classes, but multiple inheritance via interfaces.

2. Encapsulation

Encapsulation is about protecting data by wrapping variables (fields) and code (methods) together in a class.

We restrict direct access to fields using access modifiers and expose behaviour through methods.

```
class Student {
    private int marks; // can't access directly from outside

    public void setMarks(int m) {
        if (m >= 0) {
            marks = m;
        }
    }

    public int getMarks() {
        return marks;
    }
}

class EncapsulationDemo {
    public static void main(String[] args) {
        Student s = new Student();
        s.setMarks(85);
        System.out.println("Marks: " + s.getMarks());
    }
}
```

Interview Tip:

Use private fields + public getters/setters to achieve Encapsulation.

3. Polymorphism

Polymorphism means "many forms", the same object can behave differently based on context.

It is of two types: Compile-time (Method Overloading) and Runtime (Method Overriding).

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape");
    }
}

class Circle extends Shape {
    void draw() {
```

```

        System.out.println("Drawing a circle");
    }
}

class PolymorphismDemo {
    public static void main(String[] args) {
        Shape s1 = new Circle(); // Upcasting
        s1.draw(); // Calls Circle's overridden draw()
    }
}

```

Interview Tip:

Always say: Overriding = Runtime Polymorphism, Overloading = Compile-time Polymorphism.
Runtime polymorphism (also called dynamic polymorphism) means the method that gets executed is determined at runtime, not compile time.

4. Abstraction

Abstraction means hiding internal implementation and showing only essential details to the user.
It is implemented using abstract classes or interfaces.

```

abstract class Vehicle {
    abstract void start();
}

class Bike extends Vehicle {
    void start() {
        System.out.println("Bike started");
    }
}

class AbstractionDemo {
    public static void main(String[] args) {
        Vehicle v = new Bike();
        v.start();
    }
}

```

Interview Tip:

Always mention that interfaces provide 100% abstraction (prior to Java 8), and **abstract** classes can have both **abstract** and concrete methods.

5 How is abstraction different from encapsulation?

Abstraction is about hiding complexity and exposing only the required things.

- Example: We press a button to start a washing machine. we don't know (or care) how it fills water, rotates the drum, or drains it, we just know it washes clothes.
- In code: Interfaces and abstract classes are typical tools used for abstraction.

Encapsulation is about hiding data and keeping it safe from outside interference.

Example: A car's internal data like engine temperature or fuel injection logic is hidden from the driver.

In code: we use private fields and public getters/setters to encapsulate.

6 Questions on Encapsulation?

1. How to achieve encapsulation in Java?

Encapsulation is done by keeping class variables private and exposing public getter and setter methods. This restricts direct access to fields from outside the class.

2. Why is encapsulation needed?

Encapsulation keeps data safe from unwanted changes. It hides internal logic and only exposes what is necessary. This leads to cleaner, more reliable code because other parts of the program can't directly mess with the internal state.

3. Why do we make global variables private?

We make them private to prevent direct access or modification from outside the class. If fields are public, anyone can change them anytime, which can break the logic.

4. What is the role of setters and getters in encapsulation?

Setters and getters help us to control how someone uses the class's data.

A **getter** allows others to *see* the value, but not change it directly.

A **setter** allows to *change* it, but only the way we want.

5. If no encapsulation is implemented, what consequences may occur?

Without encapsulation, any part of the program can change the data. This can cause mistakes that are hard to find. One small change can break other parts of the program.

6. Can you provide an example that illustrates the concept of encapsulation?

Imagine a BankAccount class with a private balance. We use deposit() and withdraw() methods to change it. Nobody can change the balance directly. They must go through these methods, which can include checks to prevent negative balances. That is a clean example of encapsulation.

7. How does encapsulation contribute to code organization and maintenance?

Encapsulation helps to keep each part of the code separate. Every class handles its own data and work. So, we can change one part without worrying about breaking others. It also makes it easier for someone new to understand and work on just one part without looking at everything.

8. In what ways does encapsulation enhance code security?

Encapsulation hides important data so that other parts of the program can't touch it directly. Only safe actions are allowed. This protects the data from being changed in the wrong way, either by mistake or on purpose. It is like putting a lock on something valuable so only the right key can open it.

9. What are the potential drawbacks of overusing encapsulation in a program?

If we use encapsulation too much, the code can get bigger and more complicated. We might

add getters and setters even when they aren't needed. It is better to use encapsulation only when it really helps.

7 Questions on Inheritance?

1. Why do we need inheritance in programming?

Inheritance helps reuse code and organize things better. Instead of writing the same features again and again, we just inherit them from a base class.

2. How do we make inheritance happen in code?

We use the extends keyword in Java. Let's say we have a class called Animal, and we want to create a class Dog that reuses Animal's code. We write class Dog extends Animal. Now Dog automatically gets all the public and protected stuff from Animal.

3. What does extends really mean?

It just means "I want to borrow everything public or protected from that class."

4. Can you give a real-life example of inheritance?

Think of a Vehicle class and a Car class. Vehicles can move and carry people. Cars do the same, but also have music systems and AC. So, Car extends Vehicle. This saves us from rewriting shared features.

5. Why can't we use multiple inheritance in Java?

It causes confusion. If two parents have the same method name, Java won't know which one to use, this is the "diamond problem". That is why, Java won't allow multiple inheritance for classes. We need to use interfaces instead.

6. Can we do inheritance in Java without using extends?

Yes. We can use interfaces or composition. For example, instead of making a class extend another, we just include an object of that class inside.

7. Are constructors and private members inherited?

Constructors aren't inherited. The child class can call the parent constructor using super(), but it has to define its own.

Private members are also not inherited directly; We can't use them, but can access them by creating a public method.

8. How do you stop a class from being inherited?

Use the final keyword. If we write final class BankAccount, then no other class can extend it. We can also mark methods as final so they can't be overridden.

9. Why does Java allow multiple interfaces but not multiple classes?

Java allows a class to use many interfaces, but not many classes. That is because interfaces only have method names, not real code. So even if two interfaces have the same method, it is not a problem, the class that implements these interfaces writes its own version.

But classes have real code. If two classes have the same method with different logic, Java won't know which one to use. This creates confusion. So, Java allows a class extend only one class.

8 What is constructor in java?

A constructor in Java is a special method that gets called automatically when we create an object of a class. Its primary purpose is to initialize the newly created object, setting up its initial state or assigning default values to its attributes.

Points to remember:

- It has no return type (not even void).
- Its name is exactly the same as the class.
- We can have multiple constructors in the same class (this is called constructor overloading).

```
class Car {
    String model;

    // Constructor
    Car(String carModel) {
        model = carModel;
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Maruthi");
        System.out.println(myCar.model); // Prints "Tesla"
    }
}
```

When new Car("Maruthi") is called, Java automatically runs the constructor, sets model = "Maruthi"

Indirect Questions:

1. What is difference of constructor and a method?

A constructor is used to initialize an object when it is created, while a method is used to perform actions after the object exists. Constructors don't have a return type and their name must match the class name. Methods can have return types and any name. Basically, constructors set up the object; methods operate on it.

2. Can a class have multiple constructors?

Yes, Java allows constructor overloading, so a class can have multiple constructors with different parameter lists. This helps in creating objects in different ways. For example, one

constructor might set default values, and another might accept values from the user. The compiler figures out which one to call based on the arguments passed.

3. Explain the difference between a default constructor and a parameterized constructor

A default constructor has no parameters and sets fields to default values. It is either defined by us or provided automatically if no other constructors are defined. A parameterized constructor takes arguments and can initialize an object with specific values.

3. Can a constructor have a return type?

No, constructors can't have any return type, not even void. If you try to give it a return type, Java treats it as a regular method, not a constructor. Constructor's return type is class type.

4. How is a constructor invoked in Java?

A constructor is automatically called when we create a new object using the new keyword. For example: new Student("John") invokes the constructor of the Student class

5. What happens if you don't provide a constructor in a class?

If we don't write any constructor, Java automatically provides a default no-arg constructor. It initializes object fields with default values, like 0 for numbers or null for objects.

6. Can you have both a default constructor and a parameterized constructor in the same class?

Yes, and it is very common. The compiler will invoke correct one based on the arguments.

7. Explain the concept of constructor Overloading?

Constructor overloading means writing multiple constructors in the same class with different parameter lists. It allows you to create objects in various ways

8. Can a constructor call another constructor of the same class? How?

Yes, using this(...). It is called constructor chaining. We can call another constructor in the same class from within a constructor using this() as the first line.

9. What is the purpose of a static block in Java, and how is it related to constructors?

A static block runs once when the class is first loaded, before any object is created or constructor is called. It is used to initialize static variables. It is not directly related to constructors. Static blocks are for class-level setup, while constructors deal with object-level initialization.

10. Explain the difference between an instance initialization block and a constructor.

An instance initialization block is a *code block* that runs before the constructor, every time an object is created. It is useful for common setup across multiple constructors. Constructors run after that. Both help initialize the object, but constructors are more flexible and commonly used.

11. Can we override the constructor?

No. Constructors are not inherited, so they cannot be overridden.

12. Can we overload the constructor?

Yes. We can have multiple constructors in the same class with different parameter lists.

13. Can a constructor be static or final?

Static: No, constructors are instance-level, not class-level.

Final: No, constructors cannot be final because they are not inherited and final applies to methods to prevent overriding.

14. Can a constructor have return type?

No, a constructor cannot have a return type, not even void.

If we declare a return type, it is treated as a regular method, not a constructor.

But every time we call **new**, the constructor implicitly returns the instance of the class itself.

9 Difference between == and .equals() in Java

What is ==?

- It checks **whether two references point to the same object in memory**.
- It does not compare values inside the object.

What is .equals()?

- It is a method used to compare the **actual values/content** of two objects.
- For most classes (like String), it is overridden to check logical equality.

```
String a = new String("CodingLyf");
String b = new String("CodingLyf ");

System.out.println(a == b); // false : different objects in memory
System.out.println(a.equals(b)); // true : same content
```

Interview Tip: Output based questions can be asked on this topic. Prepare for them. Will discuss in output-based questions section.

Related Questions:

1. What are different ways to create a String in Java?

We can create strings using literals like

String s = "hello"; -> this goes into the string pool.

Using new keyword:

String s = new String("hello"); -> this creates a new object in heap.

We can also build strings dynamically using StringBuilder or StringBuffer.

String s3 = new StringBuilder().append("he").append("llo").toString();

From a character array using new String(char[]):

char[] arr = {'h','i'}; String s4 = new String(arr);

2. What is the String Constant Pool?

It is a special memory area in Java where string literals are stored.

When we write String a = "hello"; and then String b = "hello";, both point to the same object in the pool so no duplicate objects created.

It is to optimize memory since strings are used a lot and are immutable, so reusing the string is safe and efficient.

3. What does intern() do, and why is it useful?

The intern() method Puts the string in the pool if it is not already there, or gives the pooled version. So even if we create a string using new, we can save the string in the pool.

Example:

```
String s1 = new String("hello");// Creates a new string object in the heap
String s2 = s1.intern();      // Adds "hello" to the pool and returns a reference
to it
String s3 = "hello";         // Directly references the "hello" in the string pool
```

```
System.out.println(s1 == s2); // false (s1 is in heap, s2 is in pool)
System.out.println(s2 == s3); // true (both refer to the same object in the pool)
```

Its primary benefit is memory optimization. By ensuring that only one instance of a unique string value exists in the string pool.

10 Is Java Purely Object-Oriented?

Java is not purely object-oriented. Here is why:

1. It supports primitive data types like `int`, `char`, `boolean`, etc., which are not objects.
2. It allows static methods and variables which belong to the class and not any instance.
3. Some operations like `System.out.println("Hello")` call static methods directly without creating an object.
4. Wrapper classes (like Integer, Double) were introduced to work with primitives in an object-oriented way.

So, Java is object-based and supports OOP concepts, but it is not 100% object-oriented.

11 Difference between final, finally, and finalize()?

They sound similar but do different things:

- final is a keyword. We can use it to make a variable constant, stop a method from being overridden, or prevent a class from being extended.

```
final int age = 25;
```

```
age = 30; // Error - We can't change it
```

```
final class Car {} // Can't extend this class
```

- finally is the block of code in a try-catch that *always* runs, no matter what, even if there is an exception.

```
try {
    int x = 10 / 0;
} catch (Exception e) {
    System.out.println("Error!");
} finally {
    System.out.println("Always runs");
}
```

- `finalize()` is a method from the `Object` class that gets called just before the object is garbage collected. But honestly, no one really uses it anymore, it is considered outdated and unreliable.

```
protected void finalize() throws Throwable {
    System.out.println("Cleanup before GC");
}
```

Indirect Questions:

1. Can we combine abstract and final keywords in a class declaration?

No, we can't use abstract and final together in a class. `abstract` means the class should be extended, while `final` means it can't be extended.

2. Is it possible to declare a variable as both final and abstract?

No, that is not allowed. `final` means the variable can't change, while `abstract` means it has no value or implementation and must be defined later.

3. Can you have static final variables in Java?

Yes, and it is very common. A static final variable is basically a constant. it belongs to the class and its value never changes. For example: `static final double PI = 3.14;`. It is shared across all instances.

4. Are there any built-in classes in Java that are declared as final?

Yes. Classes like `String`, `Integer`, and `Math` are declared as final. That means we can't extend or subclass them

12 Is Java pass-by-value or pass-by-reference?

Java is always pass-by-value, but there is a catch when it comes to objects.

- For **primitive types**, it is straightforward: a copy of the value is passed. So, changing the value inside the method doesn't affect the original.

For primitives: We are sending a copy. Original value doesn't change.

```
void changeValue(int x) {
    x = 20;
}

int num = 10;
changeValue(num);
System.out.println(num); // Still 10
```

- For **objects**, Java still passes by value, but that value is the reference to the object. So, the method gets a copy of the reference. We can use it to change the object's internal state (like modifying a field).

For objects: We are sending a **copy of the reference**, so inside the method we can **change the object's internals**

```
class Dog {
    String name;
}

void renameDog(Dog d) {
    d.name = "Bruno";
}
Dog myDog = new Dog();
myDog.name = "Max";
renameDog(myDog);
System.out.println(myDog.name); // Bruno
```

Reference: [Java: Pass By Value Or Pass By Reference | by Ananya Sen | Nov, 2020 | Medium | Medium](#)

13 What is immutability, how to achieve it?

Immutability means: once an object is created, we cannot change its values.

Instead of modifying the same object, a **new object is created** when we try to change something.

In Java, this concept applies to **strings**, which means that once a string object is created, its content cannot be changed. If we try to change a string, Java does not modify the original one, it creates a new string object instead.

```
String name = "Ravi";
name = name + " Kumar";
System.out.println(name); // Output: Ravi Kumar
```

Here, "Ravi" did not become "Ravi Kumar". A new String "Ravi Kumar" was created and stored in name.

How to achieve immutability in Java?

To make a class immutable, follow these rules:

1. **Make the class final**
So that no one can extend it.
2. **Make all fields private and final**
This avoids direct access and ensures the value doesn't change after assignment.
3. **No setters, only getters**
Don't provide methods to change the field values.
4. **If any field is an object, return a copy, not the original**
So that internal data is not modified from outside. Refer (Output Question No :)

Indirect Questions:

1. How is immutability different from final keyword?

`final` means we can't reassign the variable.

Immutability means the object's internal data can't be changed.

2. Is final object immutable?

No. A final object reference means we can't assign a new object to that variable. But the contents of the object can still be changed.

3. If a class has a mutable field (like a List), how do you stop callers from modifying it?

If we return a mutable field like `ArrayList` directly from a getter, caller can update it.

```
public List<String> getList() {
    return list; // original list exposed
}
```

Someone can now call `getList().add("Hack")`;

To make it immutable we need to return a copy of the List.

```
public List<String> getSubjects() {
    return new ArrayList<>(subjects);
}
```

4. How do you make a class thread-safe without using synchronized?

Make it immutable.

Immutable objects are naturally thread-safe, because no thread can change the state.

5. How would you ensure immutability in multi-threaded programming?

By declaring fields as `final`, making the class itself `final`, and not exposing setters or mutable references.

This way, once the object is created, its state cannot be changed, ensuring thread-safety without extra synchronization.

6. Can we have mutable fields to immutable objects?

Yes, technically we can declare mutable fields inside an immutable object.

7. How would you design a custom immutable class that has `ArrayList` in it?

Check Question no. 3

8. Final vs immutable

`Final` and `immutable` may look similar, but they mean very different things in Java.

`final` is a Java keyword. It places restrictions on variables, methods, and classes. `final` doesn't freeze the object itself. If we make a `List` `final`, we can't assign a new `List` to that variable, but we can still add or remove items from the same `List`.

Immutability is about the design of the object itself. An immutable object does not allow its state to change after it is created.

The classic example is `String`. Once you create "Hello", we can't change its characters. Any modification creates a new object.

14 Why String is immutable in Java?

In Java, `String` is immutable, which means once we create a `String`, we cannot change its value. If we try to modify it, a new `String` object will be created.

Now let's understand why Java made `String` immutable:

1. For security reasons

Strings are used in things like file paths, database connections, and user logins. If Strings were changeable, someone could easily tamper with these values, that would be dangerous. So, Java made String immutable to protect such sensitive data.

2. To work properly in HashMaps and Sets

We often use Strings as keys in HashMap, HashSet, etc. These collections depend on the hashCode() of the key. If the String changes after adding it to the map, its hash code will also change and the map won't be able to find it again.

3. Thread safety

Since Strings cannot be changed, multiple threads can use the same String object without any issue. No need for extra synchronization. This makes programs safer and faster.

4. Memory efficiency using String Pool

Java stores common String values in a special memory area called the **String pool**. As Strings are immutable, Java can reuse them safely by saving memory and improving performance.

15 What is Object class in java and tell its methods?

In Java, Object class in Java is present in `java.lang` package. Every class in Java is directly or indirectly derived from the Object class. If a class does not extend any other class, then it is a direct child class of the Java Object class and if it extends another class then it is indirectly derived.

Here are the most important methods:

1. **toString()**

Returns a string representation of the object.
Useful for printing/debugging.

2. **equals(Object obj)**

Used to compare two objects for logical equality.

3. **hashCode()**

Returns an integer used in hash-based collections like HashMap, HashSet.
If we override equals(), we must also override hashCode().

4. **getClass()**

Returns the runtime class info of the object.

5. **clone()**

Used to create a copy of the object.
By default, it does a shallow copy.
To use it, the class must implement Cloneable.

6. **finalize() (Deprecated)**

Was used to run cleanup code before garbage collection.

Now deprecated. Avoid using it.

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

Used for thread communication (in multithreading).

These are called on objects used as locks (inside synchronized blocks).

Reference: [Object Class in Java - Scaler Topics](#)

16 Deep Copy vs Shallow Copy?

A **shallow copy** makes a new object and copies everything from the old one.

If it is a simple value (like an int), it copies the value itself.

If it is a complex object, it just copies the address/reference, so both still point to the same thing in memory.

Example:

Code link: [JavaSamples/ShallowCopyExample.java at main · CodingLyf-Fullstack/JavaSamples](#)

Explanation: In the below code, we have Person Object, it contains "name" as String and "Address" as object. When we copy a person object, "name" field will be copied as it is but since "Address" is object its reference will be copied not content.

So, when we change copied person name, it won't affect the original person object name but when we change the address, it will affect the address of the original object.

```
public class Test {
    public static void main(String[] args) {
        // Original object
        Person originalPerson = new Person("John", new Address("Jubliee Hills"));

        // Shallow copy using copy constructor
        Person shallowCopyPerson = new Person(originalPerson);

        // Modifying the shallow copy
        shallowCopyPerson.setName("Jane");
        shallowCopyPerson.getAddress().setStreet("Madhapur");

        //Name is changed in Original Object but Address is changed
        System.out.println("Original Person: " + originalPerson);
        System.out.println("Shallow Copy Person: " + shallowCopyPerson);
    }
}
```

Deep Copy:

A deep copy makes a new object and also makes new copies of everything inside it.

If it is a simple value (like an int), it copies the value itself.

If it is a complex object, it creates a new object in memory with the same data, so nothing is shared between the original and the copy.

We can achieve deep copy using “cloneable” marker interface.

Code link: [JavaSamples/DeepCopyWithCloneable.java at main · CodingLyf-Fullstack/JavaSamples](#)

Explanation: In the below code, we have a Person object that contains name as a String and Address as another object.

When we make a **deep copy** of the person object, both the name field and the Address object are copied independently.

So, when we change the copied person's name, it won't affect the original person's name, and when we change the copied person's address, it also won't affect the original person's address

```
public class DeepCopyWithCloneable {
    public static void main(String[] args) {
        // Original object
        Person originalPerson = new Person("John", new Address("123 Main St"));

        // Deep copy using clone()
        Person deepCopyPerson = originalPerson.clone();

        // Modify the deep copy
        deepCopyPerson.setName("Jane");
        deepCopyPerson.getAddress().setStreet("456 Side St");

        // Print results
        System.out.println("Original Person: " + originalPerson);
        System.out.println("Deep Copy Person: " + deepCopyPerson);
    }
}
```

17 Explain the difference between String, StringBuilder, and StringBuffer.

All three are used to store and work with text in Java. But they behave differently based on mutability and thread safety.

String

- **Immutable:** We can't change a String once it is created.
If we try to change it, Java creates a new object in memory.
- **Thread-safe:** Since we can't change it, it is safe to use in multi-threaded code.
- **Performance:** Slow for repeated changes. If we keep adding or changing, it keeps creating new strings. That wastes memory and slows down performance.

Use when: The value won't change often, or we just want simple text.

StringBuilder

- **Mutable:** We can change the content without creating a new object.
- **Not thread-safe:** Not safe if multiple threads are trying to use the same object.
- **Performance:** Fast. Best for string changes inside loops or single-threaded programs.

Use when: We want better performance and we are working in a single-threaded environment.

StringBuffer

- **Mutable:** Just like StringBuilder, it allows changes to the content.
- **Thread-safe:** All methods are synchronized, so it is safe when multiple threads are using it.
- **Performance:** Slower than StringBuilder. Because of thread-safety (synchronization), it is a bit slower even if we are not using multiple threads.

Use when: We want to change strings in a multi-threaded program where thread safety is important.

Additional Question:

1. Is it good to use StringBuffer in Single threaded application?

No, because it adds overhead of Synchronization which slows down the process.

18 What are Access Modifiers in Java?

Access modifiers are keywords in Java that control who can access a class, method, or variable.

We can think of them like doors, they decide whether something is open to all, only to the family, or just to yourself.

Java has four access modifiers:

1. public

- Means "open to all"
- Any class, anywhere in the project, can access it.

Use it, when we want to make the method or variable available everywhere.

2. private

- Means "only for me"
- The variable or method can only be accessed within the same class.

If we try to use age or displayAge() from another class, we will get a compile-time error.

Use it, when we want to hide data or logic from outside , for encapsulation.

3. protected

- Means "family access"
- Can be accessed:
 - Within the same class
 - In other classes in the same package
 - In subclasses (child classes) even if they are in different packages

Use when: We want to share with related classes (like inheritance), but not with the whole world.

4. default (no keyword)

- If we don't write any modifier, it becomes default access
- Visible only to classes within the same package

```
int salary = 50000; // default access
```

This salary variable can only be used by classes in the same package, not outside. Use when: We are okay to share within the package, but not outside.

19 What are Wrapper Classes in Java and Why Do We Need Them?

In Java, primitive types like int, char, float are not objects. But Java is an object-oriented language, and sometimes, especially while working with collections or frameworks, we need everything to be an object.

That is where wrapper classes come in.

Wrapper classes are object representations of the primitive data types

Primitive Wrapper Class

int Integer

char Character

float Float

double Double

boolean Boolean

long Long

short Short

byte Byte

```
int a = 10;           // primitive
```

```
Integer b = Integer.valueOf(a); // wrapper object
```

Why do we need Wrapper Classes?

1. For Collections like ArrayList, HashSet, etc.

Collections in Java cannot store primitive types directly. They only accept objects. So, if we want to store an int inside an ArrayList, we must use the Integer class.

Example:

```
ArrayList<Integer> numbers = new ArrayList<>();  
  
numbers.add(10);
```

2. For Utility Methods

Wrapper classes have helpful methods.

Example: We want to convert a String "123" into an int.

```
String s = "123";  
  
int x = Integer.parseInt(s);
```

This method `parseInt()` is available only in the `Integer` class, not with primitive types.

20 What is Autoboxing and Unboxing in Java?

In Java, sometimes we need to convert between primitive types (`int`, `char`, `double`, etc.) and their wrapper classes (`Integer`, `Character`, `Double`, etc.).

Java does this automatically in many cases, this is called Autoboxing and Unboxing.

What is Autoboxing?

Autoboxing means converting a **primitive** to its **wrapper** class automatically.

Java does this when we are assigning a primitive to an object, like:

```
int a = 10;
Integer obj = a; // autoboxing: int -> Integer

List<Integer> list = new ArrayList<>();
list.add(100); // Java converts int 100 to Integer automatically
```

So no need to write `Integer.valueOf(100)` , Java does it for us.

What is Unboxing?

Unboxing means converting a **wrapper object** back to its **primitive type** automatically.

```
Integer obj = 50;

int a = obj; // unboxing: Integer -> int
```

Again, Java internally calls `obj.intValue()`, but we don't need to write that.

21 What is Type casting?

Type casting in Java is the process of converting a variable from one data type to another. It can be either widening casting (automatic) or narrowing casting (manual).

1. Widening Casting (Automatic)

Widening casting is when a smaller data type is automatically converted to a larger data type.

This type of casting is done by Java without the need for any explicit instruction from the programmer.

Examples of Widening Casting:

`byte -> short -> int -> long -> float -> double`

```

int num = 10;
double doubleNum = num; // Automatic widening from int to double
System.out.println(doubleNum); // Output: 10.0

```

Narrowing Casting (Explicit)

Narrowing casting is when a larger data type is converted into a smaller one. This type of casting is not done automatically and requires explicit instructions from the programmer using parentheses. There is potential for data loss, so Java requires confirmation for this casting.

Examples of Narrowing Casting:

double -> float -> long -> int -> short -> byte

```

double doubleValue = 10.5;

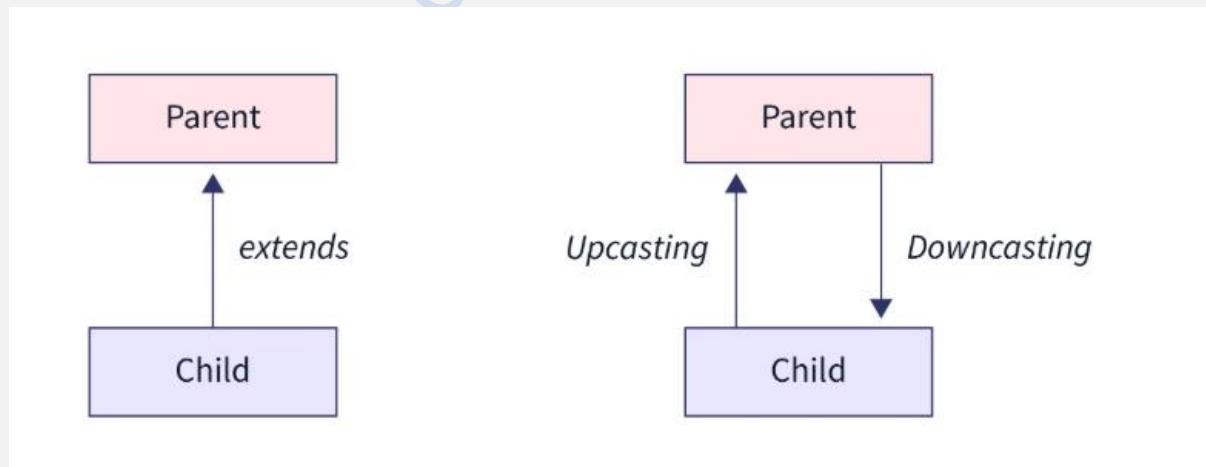
int intValue = (int) doubleValue; // Manual narrowing from double to int
                                  (truncates decimal part)

System.out.println(intValue); // Output: 10 (decimal part is lost)

```

Types of Type Casting in Java

- Primitive Type Casting:** Converting between primitive data types as shown above.
- Reference Type Casting:** Casting between objects of different types within an inheritance hierarchy. This requires upcasting (automatic) or downcasting (explicit).



Upcasting is nothing but, typecasting of a child object to a parent object.

Downcasting refers to typecasting a parent object to a child object. However, this cannot be an implicit typecasting.

Example of Reference Type Casting:

```

class Animal {
    void makeSound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}

public class Test {
    public static void main(String[] args) {
        // Upcasting: Dog to Animal (automatic)
        Dog dog = new Dog();
        Animal animal = dog; // Upcasting

        animal.makeSound(); // Allowed
        // animal.bark(); // Not allowed - Animal reference can't access Dog methods

        // Downcasting: Animal to Dog (manual)
        if (animal instanceof Dog) {
            Dog d = (Dog) animal; // Downcasting
            d.bark(); // Now allowed
        }

        // Unsafe Downcasting - Will throw ClassCastException
        Animal a = new Animal();
        Dog fakeDog = (Dog) a; // Runtime error!
        fakeDog.bark(); // This line won't be reached
    }
}

```

22 What is a static keyword in Java (Commonly asked Question)?

In Java, if we want to access class members, we must first create an instance of the class. But there will be situations where we want to access class members without creating any variables.

In those situations, we can use the static keyword in Java. If we want to access class members without creating an instance of the class, we need to declare the class members static.

The Math class in Java has almost all of its members static. So, we can access its members without creating instances of the Math class.

1. Static Variables

Static variables are shared among all instances of a class. They are stored in the class memory rather than the heap (where instance variables are stored).

When to Use:

- When we want to keep a single copy of a variable that should be shared by all instances of the class.
- For constants that belong to the class rather than any specific instance

```
class Counter {
    static int count = 0; // shared variable

    Counter() {
        count++;
        System.out.println(count);
    }
}

public class Main {
    public static void main(String[] args) {
        new Counter(); // prints 1
        new Counter(); // prints 2
        new Counter(); // prints 3
    }
}
```

2. Static Methods

Static methods belong to the class rather than any particular instance. They can access static variables directly but cannot access instance variables or methods.

In every Java program, we have declared the main method static. It is because to run the program the JVM should be able to invoke the main method during the initial phase where no objects exist in the memory.

When to Use:

- When we want a method that can be called without creating an instance of the class.
- For utility or helper methods that don't require any object state.

```
class MathUtils {
    static int square(int n) {
        return n * n;
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(MathUtils.square(5)); // prints 25
    }
}
```

3. Static Blocks

Static blocks are used for static initialization of a class. They run when the class is loaded.

When to Use:

- To initialize static variables or perform operations that need to happen once when the class is loaded

The static block is executed only once when the class is loaded in memory. The class is loaded if either the object of the class is requested in code or the static members are requested in code.

A class can have multiple static blocks and each static block is executed in the same sequence in which they have been written in a program.

```
class Test {  
    // static variable  
    static int age;  
  
    // static block  
    static {  
        age = 23;  
    }  
}
```

Reference: [Java Static Keyword \(With Examples\)](#)

Indirect Questions:**1. Can a static constructor exist in Java?**

No. Java doesn't support static constructors like C#. But we can use a static block to run code when the class is first loaded. It is usually used to initialize static variables. It runs only once, when the class is loaded into memory.

2. What is the process for accessing static variables in Java?

We can access static variables using the class name, like `ClassName.variable`. We can also use an object, but the class way is better and clearer. Since static variables belong to the class, we don't need an object to use them.

3. What is the difference between static and non-static methods?

Static methods belong to the class and can run without an object. Non-static methods belong to objects, so we need an instance to use them.

4. Can static methods access non-static variables in Java?

No, they can't directly access non-static variables because those belong to an object. Static methods don't have access to this, we can pass an object as a parameter, so they access instance variables or non-static methods.

5. Are static members shared among different instances of a class in Java?

Yes. Static members are shared across all instances of a class. If one object changes a static variable, that change is reflected in all other objects.

6. Can we throw checked exceptions from static block?

From a static block, We cannot throw checked exceptions directly. We must handle them inside the block or wrap them in an unchecked exception.

```
public class Example {
    static {
        try {
            throw new IOException("Checked exception in static block");
        } catch (IOException e) {
            e.printStackTrace(); // handle it here
        }
    }
}
```

Static blocks: Checked exceptions must be caught or wrapped; can't propagate.

7. Why main method is static in java?

Because the JVM needs to call main without creating an object first.

8. How static works in inheritance?

Static methods are not inherited in the same way as instance methods. They are bound at compile time (method hiding), not at runtime (no polymorphism). If a subclass defines a static method with the same signature, it hides the parent's method, but doesn't override it.



23 What are Nested Classes?

In Java, sometimes we need to group classes that are logically related. One class can be declared inside another; this is called a Nested Class.

There are two primary types: Static Nested Class and Non-Static Inner Class.

Static Nested Class

A static nested class is declared with the 'static' keyword inside another class.

It belongs to the outer class itself, not to any instance. This means it can only access the static members of the outer class. It does NOT need an object of the outer class to be created.

```
class DatabaseConfig {
    static String driver = "com.mysql.jdbc.Driver";

    static class Credentials {
        void printDriver() {
            System.out.println("Driver: " + driver); // accessing static
variable
    }
}
```

```

        }
    }

//Usage Example for Static Nested Class
class StaticNestedDemo {
    public static void main(String[] args) {
        DatabaseConfig.Credentials creds = new DatabaseConfig.Credentials();
        creds.printDriver();
    }
}

```

When to Use Static Nested Class:

- When the nested class is a helper or utility related to the outer class.
- When it doesn't need access to instance (non-static) variables/methods.
- It helps in logically grouping classes and keeping code organized.

Non-Static Inner class:

A non-static inner class is an instance-level class defined inside another class.

It is tied to an instance of the outer class. It CAN access all members of the outer class, even private ones.

```

class Car {
    private String model = "Tesla";

    class Engine {
        void start() {
            System.out.println("Starting engine of " + model); // accessing
private outer field
        }
    }
}

//Usage Example for Non-Static Inner Class
class InnerClassDemo {
    public static void main(String[] args) {
        Car myCar = new Car();
        Car.Engine engine = myCar.new Engine();
        engine.start();
    }
}

```

When to Use Non-Static Inner Class:

- When the inner class needs full access to the outer class instance's state.
- When it makes sense conceptually that the inner class is a part of the outer class.
- For example: Car -> Engine, Computer -> CPU

24 What is the difference between method overloading and method overriding? (Commonly asked question)

One of the most common interview questions in Java is the difference between method overloading and method overriding. These two concepts fall under polymorphism but they work very differently.

Method Overloading (Compile-time Polymorphism)

Method overloading means having multiple methods with the **same name but different parameter lists** (either in number, type, or order). It is resolved at compile time.

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
}

class OverloadingDemo {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(10, 20)); // calls int, int
        System.out.println(calc.add(5.5, 4.5)); // calls double, double
        System.out.println(calc.add(1, 2, 3)); // calls int, int, int
    }
}
```

Key Point:

- All overloaded methods must differ in parameters.
- Return type alone is NOT enough to overload.

Method Overriding (Runtime Polymorphism)

Method overriding is when a subclass redefines a method of its parent class with the **same name, return type, and parameters**. It is resolved at runtime.

Output based questions can be asked on rules of Method overriding. Check the next question for rules.

```
class Animal {
    void speak() {
        System.out.println("Animal speaks");
    }
}
```

```

class Cat extends Animal {
    void speak() {
        System.out.println("Cat meows");
    }
}

class OverridingDemo {
    public static void main(String[] args) {
        Animal animal = new Cat(); // Upcasting
        animal.speak(); // Calls Cat's speak() method
    }
}

```

Key Rules:

- Method name, parameters, and **return** type must match exactly.
- Access modifier cannot be more restrictive. (Refer Next question for these rules).
- We can use @Override annotation to help the compiler.

Interview Tip:

Always mention overloading increases readability and flexibility.

Overriding enables dynamic behaviour and is key to runtime polymorphism.

Reference: [Java Method Overloading and Overriding | Medium](#)

Indirect Questions:

1. Can you overload a method by changing only the return type?

No, we cannot overload a method just by changing the return type in Java.

The compiler decides which method to call based on the method name and parameters (not return type).

25 What are the rules for overriding a method in Java?

Mostly, Output based questions can be asked on these rules. We will see those questions in Output based question section. Here are the rules that needs to be remembered.

Rule #1: Only inherited methods can be overridden.

Because overriding happens when a subclass re-implements a method inherited from a superclass, so only inherited methods can be overridden, that is straightforward. That means only methods declared with the following access modifiers: **public**, **protected** and default (in the same package) can be overridden. That also means **private** methods cannot be overridden.

Rule #2: Final and static methods cannot be overridden.

Rule #3: The overriding method must have same argument list.

Rule #4: The overriding method must have same return type (or subtype).

Rule #5: The overriding method must not have more restrictive access modifier.

This rule can be understood as follows:

- If a method is public, we cannot override it with protected, default, or private.
- If a method is protected, we cannot override it with default or private.
- If a method has default (package-private) access, we cannot override it with private

Rule #6: The overriding method must not throw new or broader checked exceptions

Rule #7: Use the super keyword to invoke the overridden method from a subclass.

Indirect Questions:

1. Why can't private methods, static methods, or constructors be overridden in a subclass?

Because private methods aren't visible outside the class, static methods belong to the class not the object, and constructors are meant to initialize that class not its child.

2. What happens if you try to override a final method in a subclass?

We will get a compile-time error, we cannot override the final class

3. Can a subclass provide its own implementation of a static method from the superclass?

Yes, but that is method hiding, not overriding.

4. What happens if a subclass method has the same name but different parameters compared to the superclass method?

That is called overloading, not overriding. Java will treat it as a separate method.

5. Does changing the parameter types in a subclass method still considered as overriding?

No. To override, the method signature must match exactly, same name, same parameters.

6. How does Java differentiate between overriding and overloading in a subclass?

If the method signatures match, it is overriding. If only the names match but parameters are different, it is overloading.

7. Can a public method in a parent class be overridden as protected or private?

No, we can't reduce visibility.

8. Can a protected method in a parent class be overridden as public?

Yes. Increasing visibility is allowed.

9. Is it possible for an overriding method to have a completely different return type than the parent?

We cannot use completely different type. It must be a subtype of the original (covariant return), or else Java throws compile error.

10. How does covariant return type apply in method overriding?

Covariant return types in method overriding allow an overridden method in a subclass to return a more specific type (a subtype) of the return type declared in the superclass's method.

11. How can you call a parent method from a child?

Just use super.methodName(), this tells Java to call the parent method.

26 What is Shadowing in java and its types?

Shadowing happens when a variable or method in a subclass hides (shadows) a variable or method with the same name in the parent class.

1. Variable Shadowing

- If a subclass declares a variable with the same name as a variable in the superclass, the subclass variable shadows the parent one.
- The type of the variable can be different.
- Shadowing is different from overriding because it works on variables, not methods.

```
class Parent {
    String name = "Parent";
}

class Child extends Parent {
    int name = 10; // Shadows Parent's 'name' (different data type)

    void printNames() {
        System.out.println(name);           // 10 (Child's name)
        System.out.println(super.name);     // "Parent" (Parent's name)
    }
}

public class Main {
    public static void main(String[] args) {
        Parent parent = new Child();
        System.out.println(parent.name); //Parent

        Child child = new Child();
        child.printNames();
    }
}
```

2. Method Shadowing (Static Method Hiding)

- If a subclass defines a static method with the same name as a static method in the parent class, the subclass method hides the parent one.
- This is not method overriding because static methods are bound at compile-time (not runtime).
- The method signature must match (same name & parameters).

```
class Parent {
    static void print() {
        System.out.println("Parent's static method");
```

```

    }

}

class Child extends Parent {
    static void print() { // Hides Parent's print()
        System.out.println("Child's static method");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent.print(); // "Parent's static method"
        Child.print(); // "Child's static method"

        Parent obj = new Child();
        obj.print(); // "Parent's static method" (Not overridden!)
    }
}

```

27 What are super and this in Java? How are they used?

this keyword.

- Refers to the **current object** of the class.
- Used when we want to refer to the current instance variable or method.
- Also used to call another constructor in the same class.

Use Cases of **this**:

- Resolving name conflicts between instance variables and parameters.
- Passing current object as argument: someMethod(**this**)
- Chaining constructors: **this(...)** to call another constructor in same **class**.

```

class Student {
    String name;

    Student(String name) {
        this.name = name; // distinguishes instance variable from constructor
parameter
    }

    void display() {
        System.out.println("Name: " + this.name);
    }
}

class ThisKeywordDemo {
    public static void main(String[] args) {
        Student s = new Student("CodingLyf");
        s.display();
    }
}

```

super keyword

- Refers to the parent **class**.
- Used to call parent **class's** constructor, method, or variable.
- Helps when subclass overrides methods or hides variables.

```
class Animal {
    String type = "Animal";

    void sound() {
        System.out.println("Some animal sound");
    }
}

class Dog extends Animal {
    String type = "Dog";

    void printType() {
        System.out.println(this.type); // Dog
        System.out.println(super.type); // Animal
    }

    @Override
    void sound() {
        super.sound(); // calls Animal's sound method
        System.out.println("Dog barks");
    }
}

class SuperKeywordDemo {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.printType();
        dog.sound();
    }
}
```

Use Cases of super:

- Calling overridden method from parent class.
- Accessing hidden fields in superclass.
- Calling parent class constructor using **super()**.

Interview Tip:

- **this** is about the current object, **super** is about inheritance.
- Always mention that **super(...)** must be the first statement in a constructor.
- **this(...)** also must be the first if used, but can't combine both in same constructor.

Practice a few scenarios involving constructor chaining and method overriding and you will easily master these for interviews!

Reference: [This and Super Keyword in Java - Scaler Topics](#)

Indirect Questions:

1. In what situations would you need to use this in a constructor?

When the constructor parameter names are the same as instance variable names, this helps avoid confusion. For example: `this.name = name;` tells Java we are assigning the value to the object's field, not just the parameter.

2. Can this be used to call a method or access a variable?

Yes. We can use 'this' to access the instance variables or call other methods in the same class. It is helpful when local variable names clash or when we want to make it clear that we are working with the current object's fields or behavior.

3. Does this always refer to the current class instance?

Yes, it always refers to the current object of the class we are working with. It is how we point to the actual instance the code is running on. It doesn't refer to the class or other objects, only the current one.

4. In what situations would you use super() or this() in a constructor?

Use `super()` when we want to call a parent class constructor. Use `this()` to call another constructor in the same class.

5. Can it be possible to put super() or this() anywhere in a Java program?

No. We can only use `super()` or `this()` as the very first line inside a constructor. If we try to place it anywhere else, the compiler will throw an error.

28 Explain about Abstract classes in Java

An abstract class in Java is a class that cannot be instantiated directly. It is meant to be extended by other classes. Think of it as a **partial blueprint** for subclasses. It defines some behaviour, but leaves other parts for subclasses to implement.

Why to use abstract classes?

- To enforce common behaviour in subclasses
- To provide a base structure but allow custom logic in specific classes

```
abstract class Shape {
    abstract void draw(); // abstract method (no body)

    void move() {
        System.out.println("Moving the shape"); // concrete method
    }
}

class Circle extends Shape {
    @Override
```

```

    void draw() {
        System.out.println("Drawing a circle");
    }
}

class AbstractClassDemo {
    public static void main(String[] args) {
        Shape s = new Circle(); // Allowed, reference is abstract, object is
concrete
        s.draw();
        s.move();
    }
}

```

Rules of Abstract Classes

- It Can have abstract methods (without body) and concrete methods (with body)
- It Cannot be instantiated directly (new Shape() is not possible)
- It Must be extended by subclasses
- Subclasses must implement all abstract methods, unless the subclass is also abstract

When to use abstract classes:

- When we have a common base but not every detail is known in base class
- When some methods should have default implementation, but others should be overridden

Interview Tip:

- Mention we use abstract class when we want to provide common behaviour with shared code.
- Prefer interface if we only need method declarations.

Real-Time Example: Payment System

Imagine we are building a payment system for an e-commerce app. We need a common structure for all payment methods (e.g., CreditCard, UPI, NetBanking).

Code: [JavaSamples/AbstractDemo.java at main · CodingLyf-Fullstack/JavaSamples](#)

Indirect Questions:

1. How can we achieve abstraction?

In Java, abstraction is done using abstract classes and interfaces. We hide implementation details and show only the required functionality.

2. Can you provide a real-life example of abstraction?

Think of an ATM machine. We interact using a screen and buttons, withdraw, deposit, check balance, but have no idea what's happening behind the scenes. That is called abstraction.

3. Are there any built-in abstract classes in Java?

Yes. Java has built-in abstract classes like `AbstractList`, `InputStream`, `Reader`.

4. Can we create an instance of an abstract class in Java?

No, We can't.

But can create an object of a concrete subclass that implements the abstract class.

5. Can abstract class has constructor?

Abstract classes can have constructors. We can't create an object from them directly, but the constructor runs when a subclass is instantiated. It is useful for initializing the shared properties.

6. Can we declare an abstract method as static or private?

No, abstract methods can't be static or private. abstract means the method must be overridden in a subclass. But static means it belongs to the class and can't be overridden. private means it can't be accessed outside the class

7. Can an abstract class extend another abstract class?

Yes

8. Can an abstract class implement the interface?

Yes, it can implement the interface and don't need to override the methods. However, any concrete subclass extending that abstract class must override those methods.

In the below, get() is not overridden in Child class so it must be overridden any class that extends Child.

```
interface Parent
{
    void show();
    void get();
}

abstract class Child implements Parent
{
    public void show()
    {
        System.out.println("Inside Child");
    }
}
```

9. Is it possible to create reference variable to abstract class and assign the concrete class object?

We can't instantiate an abstract class, but we can create a reference variable of its type and assign it to a concrete subclass object.

```
abstract class Animal {
    abstract void sound();
}

class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Animal a = new Dog();      // abstract class reference, concrete object
        a.sound();                 // prints "Bark"
    }
}
```

29 Explain about Interfaces in Java

An interface in Java is like a contract. it *defines a set of methods* that a class must implement. Interfaces help achieve 100% abstraction (before Java 8) and allow multiple inheritance.

Think of it as a rulebook, any class that 'signs' the interface must follow the rules (i.e., implement the methods).

Key Features of Interface:

- All methods are public and abstract by default (until Java 7)
- From Java 8 onwards, interfaces can have default and static methods
- Interface cannot have constructors
- All variables in interface are public, static, and final
- A class can implement multiple interfaces (solves the multiple inheritance issue)

Interview Tip:

- Use interface when we only need to define the contract, not implementation
- We can combine multiple interfaces using `implements A, B, C` unlike classes
- Interfaces support default and static methods (since Java 8)

Real-World Example: Payment Gateway Integration

Let's say You are building an app that supports multiple payment providers like Razorpay, Paytm, and PhonePe.

Code: [JavaSamples/InterfaceDemo.java at main · CodingLyf-Fullstack/JavaSamples](#)

30 Interface: Indirect questions

1. Can a class implement multiple interfaces?

Yes, Java supports multiple interface implementation.

This *is* one way Java supports multiple inheritance of *type*.

A class can implement many interfaces, separated by commas.

2. If two interfaces have methods with the same signature, what happens?

It will compile; the method needs to be implemented once in the class.

But if both interfaces have default methods with same name, we must override it.

Java forces to resolve conflicts manually.

3. Can an interface have a *constructor*?

Interfaces can't have constructors because they can't be instantiated directly.

Interfaces are meant for defining contracts, no for object creation.

Only classes have constructors in Java.

4. What's the difference between abstract class and interface if both can have abstract methods?

Abstract class can have state (variables, constructor, etc), interfaces can't (except constants).

We can extend only one abstract class, but implement multiple interfaces.

Interfaces are ideal when we just need to define a structure.

5. Have you used any built-in Java interfaces in your code?

Yes, common ones are `Runnable`, `Comparable`, `Serializable`, `AutoCloseable`.

These are part of core Java libraries.

6. What happens if you don't implement all methods of an interface?

Then the class must be declared `abstract`.

Java will not compile it if we don't implement all the methods.

7. Can we define variables inside an interface?

Yes, but all variables are implicitly `public static final` (constants).

We cannot define instance variables.

8. Can interface methods be private or protected?

No, all methods are `public` by default in classic interfaces.

Since Java 9, `private` methods are allowed for internal use only. `protected` is not allowed.

9. Can an interface extend another interface?

Yes. One interface can extend one or more interfaces.

This helps in grouping related contracts.

10. Why are default methods introduced in Java 8 interfaces?

To allow interfaces to have some method implementation.

This helps add new methods without breaking existing code.

Mainly used for backward compatibility and utility methods.

11. Can an interface have variables? What type are they by default?

Yes, interfaces can have variables, which are by default public, static, and final constants. This means they are accessible from anywhere, belong to the interface itself rather than any specific object, and their values cannot be changed after being initialized.

31 Interface vs Abstract Class

Think of Abstract Class as a semi-built house:

- Some walls, roof, and structure are already there
- Subclass (child class) needs to complete it

Think of Interface as a blank contract or blue print of the house:

- It just gives blue print of the paper and says what things must be done.
- We decide how to implement it

| Feature | Abstract Class | Interface |
|-------------------------------|--|--|
| Keyword | abstract class | interface |
| Methods | Can have both abstract and concrete methods | Abstract by default (before Java 8), can have default and static methods (Java 8+) |
| Inheritance | Single inheritance (extends one class) | Multiple inheritance (implements many interfaces) |
| Variables | Can be any type (instance, static, final, non-final) | Only public static final (constants) |
| Constructors | Yes (can have constructors) | No (cannot be instantiated) |
| Access Modifiers | Can use any (private, protected, etc.) | All members are public by default |
| Default Implementation | Yes (can provide method implementations) | Yes (via default methods since Java 8) |

32 What is Serialization in java?

Serialization is the process of converting a Java object into a byte stream. This byte stream can be stored in a file, sent over a network, or saved in a database.

Deserialization is the reverse , converting the byte stream back into a Java object.

- In a microservices architecture, different services communicate with each other over the network, often through REST APIs, gRPC, or message queues (like Kafka or RabbitMQ). Serialization allows the data objects to be converted into a byte stream so that they can be transmitted across network boundaries.
- **For example**, when a service needs to send a complex object as part of a request or a message, making the object **Serializable** ensures it can be converted into a format (e.g., JSON, XML) suitable for network transfer.
- In a RESTful service, serialization is commonly used to convert objects to a JSON or XML format for HTTP communication between a client and server. In Java, libraries like Jackson, Gson, or JAXB handle serialization (converting objects to JSON or XML) and deserialization (parsing JSON or XML back into objects)

Important Points:

- The class must implement **Serializable** interface to make it serializable
- **transient** keyword is used for fields which we don't want to save
- **SerialVersionUID** should be declared to maintain version compatibility

Indirect Questions:

1. How do you make a class serializable in Java?

To make a class serializable in Java, it must implement the `Serializable` interface. This is a marker interface with no methods.

2. What is serialVersionUID?

`serialVersionUID` is a unique identifier for a `Serializable` class. It is used during deserialization to ensure that the sender and receiver of a serialized object have loaded classes for that object that are compatible

3. What happens if you don't define `serialVersionUID` in a Serializable class?

If `serialVersionUID` is not explicitly defined, the Java runtime will generate one automatically based on various aspects of the class. This can lead to unexpected `InvalidClassException` errors if the class is modified, as the generated `serialVersionUID` might change

4. When will be serialization useful?

Serialization is useful when we need to convert an object into a byte stream so it can be stored or transferred, then later rebuilt.

Typical cases:

- Saving an object's state to a file or database.
- Sending objects over a network (like Rest APIs)

In REST APIs, serialization is what happens when the Java object (like a DTO or entity) is converted into JSON or XML before sending it in the HTTP response.

Likewise, when the API receives JSON/XML, it is deserialized back into a Java object. So, without serialization, the REST API couldn't exchange data with clients.

- Caching objects in memory or on disk.

33 What is transient in java?

In Java, the `transient` keyword is used to tell the JVM: "Do NOT save this variable during serialization." That means if an object is converted into a byte stream, all `transient` fields will be ignored.

Why is this useful?

Let's say we have a password field or some sensitive/confidential data , we don't want it stored in files or transferred.

Or maybe we have a field like a file stream or a socket, which is not serializable.

Possible Indirect Questions:

[How do you prevent certain fields from being serialized?](#)

We can prevent certain fields from being serialized by marking them with the `transient` keyword.
Transient fields are ignored during the serialization process.

34 What is marker interface in java?

A Marker Interface in Java is an interface that has no **methods or fields**.

It is completely empty. We use it just to mark a class, like saying "this class has a special quality".

Think of it like sticking a 'special sticker' on a class.

JVM see that sticker and do something different for that class.

Examples for Marker Interfaces in Java:

- Serializable: tells JVM that objects of this class can be serialized.
- Cloneable: allows an object to be cloned using Object.clone().

Real-World Analogy:

Think of a movie ticket with 'VIP' stamped on it.

The stamp itself doesn't do anything, but the staff sees it and allow into a special lounge.

Similarly, JVM checks if a class implements certain marker interfaces and behaves accordingly.

Indirect Questions:

1. Have you written any custom marker interface?
2. Can you implement a custom marker interface for example?

Auditable -> Marks entities whose changes should be logged (e.g., created/updated timestamps).

```
//Marker interface with no methods
interface Auditable { }

class LibraryItem implements Auditable {

    private String title;
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;

    public LibraryItem(String title) {
        this.title = title;
        this.createdAt = LocalDateTime.now();
    }

    public void updateTitle(String newTitle) {
        this.title = newTitle;
        this.updatedAt = LocalDateTime.now();
    }

    @Override
    public String toString() {
        return "LibraryItem{" +
            "title='" + title + '\'' +
            ", createdAt=" + createdAt +
            ", updatedAt=" + updatedAt +
            '}';
    }
}
```

Main class:

```
public class AuditDemo {
    public static void main(String[] args) {
        LibraryItem book = new LibraryItem("Clean Code");
        book.updateTitle("Clean Code - Updated");

        LogChange(book);
    }

    static void logChange(Object obj) {
        if (obj instanceof Auditable) {
            System.out.println("Audit log: " + obj);
        }
    }
}
```

```
}
```

3. Can we achieve marker interface with annotations?

Yes, In Spring framework we can achieve this with annotation.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Marker {}
```



```
@Marker
public class MyClass {}
```

35 Explain about Association, Aggregation and Composition?

These three are used to define how objects are connected to each other.

Think of them as the strength of relationship between two classes.

Let's go from basic to strong: Association -> Aggregation -> Composition.

1. Association

- This is the most basic connection between two classes.
- Example: A Student attends a college. Student and College are related.
- But both can exist without each other. One doesn't depend on the other.

```
class Student {
    String name;
    College college; // Association
}

class College {
    String name;
}
```

2. Aggregation

- A more specific type of association where one object "has-a" relationship with another object.
- It means one class contains another, but the child can still exist on its own.
- Example: A Library has Books. Even if Library is deleted, Books can still exist elsewhere.
- Example: A Team object and a Player object. The team contains multiple players but a player can exist without a team.

```
class Library {
    List<Book> books; // Aggregation
}

class Book {
    String title;
}
```

3. Composition

- The strongest form of association, also a "has-a" relationship.
- If the parent is deleted, child also goes.
- Example: Consider the case of Human having a heart. Here Human object contains the heart and heart cannot exist without Human.

```
class House {
    Room room = new Room(); // Composition
}

class Room {
    int size;
}
```

Indirect Questions:

1. Can you explain the difference between aggregation and composition?

Aggregation is a weak relationship, child can exist on its own.
In composition, child's life depends on the parent.

2. How would you model a car and its engine in Java?

Car and Engine are tightly connected, so it is composition.
If the car's gone, engine usually has no use also

3. What's the best way to model a Book and Library in OOP?

Books can exist without a Library. they can belong to another.
So, it is aggregation.

4. Is a bank and account aggregation or composition?

That is aggregation, accounts can be moved to another bank.
They don't die with the bank.

5. Can an object be part of multiple compositions?

No, in composition, the parent fully owns the child.
One child can't belong to multiple owners like that.

Code: [JavaSamples/CompositionDemo.java at main · CodingLyf-Fullstack/JavaSamples](#)

36 Why “Composition over inheritance”?

In object-oriented programming (OOP), we find two fundamental relationships that describe how objects relate and interact with each other: '**IS-A**' and '**HAS-A**'

In OOP, '**IS-A**' is a relationship where one object is a subtype of another. A Car is a Vehicle; a Dog is an Animal; this relationship is manifested through inheritance. The '**HAS-A**' relationship is

where one object contains or uses another object. A Car has an Engine; a Dog has a Tail; this relationship is realized through composition

Inheritance means one class can use the code from another class by extending it. That might be helpful, but it is not always a good idea.

Why? Because it creates a strong link between the two classes. If the parent class changes, it can break the child class. Also, Interfaces has N number of methods, we need to inherit all of them even though we don't need.

Reference: [Composition over Inheritance \[Object Oriented Programming\]](#)

[Favoring Composition Over Inheritance | by Sheldon Cohen | Medium](#) (Code might be C# but theory is good to understand)

37 What is multi-tasking?

Multitasking allows a system (or program) to run more than one task simultaneously. For example, we might be listening to music while downloading a file and editing a document, all happening at the same time.

In Java, multitasking is handled through multithreading or multiprocessing, depending on the context.

Types of Multitasking in Java

There are two main types of multitasking:

1. **Process-based Multitasking**
2. **Thread-based Multitasking**

1. Process-Based Multitasking

Each task is a separate process, meaning a complete, independent program running in its own memory space.

Example

- Opening a browser, opening a code editor, and opening a video player at the same time.

Points to remember:

- Each process runs independently.
- It requires more memory since each process has its own memory and resources.
- Switching between processes involves context switching, which is expensive.
- In Java, this is done using the ProcessBuilder or Runtime.exec().

2. Thread-Based Multitasking

Each task is a separate thread, but all threads run within a single process. They share the same memory space.

Example:

- A video player: one thread handles video decoding, another thread handles audio, another thread handles sub titles.

Points to remember:

- Lightweight compared to processes.
- Threads share memory, which makes data sharing easy but also introduces the need for synchronization.
- Faster context switching between threads.
- Most commonly used in Java via Thread class or implementing Runnable.

38 Difference between Process and Thread?

What is a Process?

A process is an independent running program.

- When we start a Java application (like java MyApp), the JVM (Java Virtual Machine) creates a new process.
- Each process has its own memory space, meaning it doesn't share data with other processes.
- Two processes cannot directly access each other's memory. They need special communication methods like sockets or inter-process communication (IPC).
- Processes are handled by the OS, but threads are what Java developers often use to get things done efficiently inside a single app.

Real-world example:

Opening two different apps on your computer, like a browser and a text editor, each runs as a separate process. They don't interfere with each other.

What is a Thread?

A thread is a smaller unit of execution within a process.

- A Java process (i.e., your running app) can have multiple threads running inside it.
- All threads in the same process share memory (especially the heap). This makes data sharing fast and easy.
- Each thread still has its own call stack (local variables), so they don't clash at every level.
- Threads are commonly used for multitasking, doing more than one thing at the same time.

Real-world example:

In a video calling app (like Zoom), multiple threads may be running:

- One thread handles audio
- Another handles video
- A third listens for chat messages
- All of these threads live inside one single process: the Zoom app

Indirect Questions:

1. What is thread pool?

- A thread pool is a collection of reusable threads managed by the JVM.

- Instead of creating a new thread every time (which is costly), tasks are submitted to the pool and executed by existing threads.
- In Java, we typically use ExecutorService / Executors.newFixedThreadPool().

2. Explain object-level thread locks.

- When we mark a method or block as synchronized (non-static), the lock is taken on the current object (this).
- Only one thread can access the synchronized block/method of that object at a time.
- But different instances of the class have different locks.

39 What is Java Memory Model (JMM)?

The Java Memory Model explains how threads share data. Imagine every thread has its own notepad (working memory) where it keeps copies of variables from the main whiteboard (heap). A problem appears when one thread updates the whiteboard but another thread is still looking at its old notes.

Java Memory Model establishes rules and guarantees regarding the visibility, ordering, and atomicity of operations on shared variables, ensuring consistent and predictable behaviour in multithreaded environments.

Here are the key parts:

1. **Heap Memory:** Stores objects and shared data. All threads can access it.
2. **Stack Memory:** Each thread has its own stack, storing method calls and local variables.
3. **Working Memory (per thread):** A thread often keeps a copy of variables in its own cache. This is faster, but can cause problems if other threads change the original variable in the heap.

3 rules JVM follows in multi-threading

1. Visibility Rule

- Without synchronization, Change made by one thread may stay in cache and not seen by others.
- Use volatile or synchronized so updates are visible across threads.

2. Ordering Rule

- JVM and CPU may reorder instructions for speed.
- It uses the **happens-before** rule: if action A happens before B, then B will always see A's changes.

3. Atomicity Rule

- The JMM guarantees that certain operations appear as a single, indivisible unit
- Reads/writes to int, boolean, etc. are atomic.

- For long and double, use volatile or synchronized for atomicity.

Indirect Questions:**1. How JMM Affects Threads**

When multiple threads run in Java, they don't directly read/write main memory all the time. Instead, each thread has its own working memory (CPU cache).

Thread-local caching: Each thread may keep variables locally to avoid constant memory access.

Main memory: Shared among all threads. Updates in one thread are not immediately visible to others unless synchronized properly.

If Thread A updates a variable, Thread B may not see the update immediately.

Use volatile, synchronized blocks, or atomic variables.

2. What is ThreadLocal in java and where would you use it?

ThreadLocal is a special kind of variable in Java where each thread gets its own independent copy.

- Normally, if multiple threads use the same variable, they share it.
- With ThreadLocal, each thread has its own private copy, invisible to other threads.

Think of it like giving each worker in a team their own notepad. Even if they're all writing notes, no one can see or mess with anyone else's notes.

Use cases:**Database connections / transactions (per thread):**

In frameworks, a thread might handle one request.

A ThreadLocal can store the current DB connection or transaction

User/session context:

In a web app, when a request comes in, we can store the current user or request ID in a ThreadLocal.

Later, anywhere in the call chain, we can fetch it without explicitly passing it as a parameter.

40 What is multi-threading how is it different from multi-tasking?

Multithreading is a programming concept where a single process can perform multiple tasks concurrently using multiple threads.

Each thread is like a lightweight sub-process. They run in parallel (or appear to) and share the same memory and resources of the parent process.

Multitasking means the operating system is running *multiple programs* at once.

Example:

- We are running a Java app, a browser, and a PDF reader all at the same time.
- Each of these runs as a separate process.

- The OS switches between them so fast that it feels like they are all running at the same time.

Multitasking = Our computer runs many apps at once.

Multithreading = A single app handles many tasks at once internally.

41 What is Thread, how to create threads in Java?

A **Thread** is just a lightweight worker inside the Java program.

Every Java program starts with one which is called a main thread. But if we want the program to do more than one thing at the same time (like download a file and show a progress bar), we need more threads.

Real-Life Example:

Imagine we're running a restaurant alone. We take orders, cook, serve, one at a time. That is a single-threaded program.

Now hire more staff, one handles orders, another cooks, another serves. Things move faster. That is **multi-threading**.

Java lets us create such workers inside the program.

There are **two ways to create a Thread**

1. By extending the Thread class:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start(); // never call run() directly
    }
}
```

2. By implementing the Runnable interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread is running...");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

```
}
```

Possible Indirect Questions:**1. What happens if we call run() instead of start()?**

It runs like a normal method inside the main thread, no new thread is created. So, there's no parallel execution.

2. Can you run two threads on the same object?

Yes, we can create multiple Thread objects and pass the same Runnable.

Each thread will run its own run() but share the same object state.

Sample: [JavaSamples/ThreadDemo.java at main · CodingLyf-Fullstack/JavaSamples](#)

3. Why is Runnable preferred in real-world projects?

Because Java allows only one class to be extended, but multiple interfaces can be implemented.

So Runnable gives more flexibility, especially in complex designs.

4. Can we create threads without extending the Thread class?

Yes , just implement Runnable or use lambda/anonymous classes with Thread.

Extending Thread isn't required at all.

5. Is Java thread-safe by default?

No, most Java code is not thread-safe unless we explicitly handle it.

We need to use synchronization, locks, or concurrent APIs.

6. Why threads are useful in Java?

Java uses threads heavily for better performance. For example:

- In **web servers**, one thread handles each user request.
- In **desktop apps**, one thread shows the UI, another handles background tasks.
- Java provides built-in tools like Thread, Runnable, ExecutorService to manage threads.

42 What are Runnable and Callable interfaces, and their differences?

In Java, when we want to run some tasks using threads. we usually write code that goes inside Runnable or Callable. Both run tasks in parallel, but there is one key difference:

Runnable doesn't return anything. Callable returns a result.

Another Difference is Runnable cannot throw Checked Exceptions, Whereas Callable can throw.

Runnable:

Think of Runnable like sending your friend to buy something, but we don't wait to hear what happened.

We just say: "Go do this" and move on.

Runnable is used when we don't need a result back. It is used to run some code.

```

class MyTask implements Runnable {
    public void run() {
        System.out.println("Task is running");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new MyTask());
        t.start();
    }
}

```

Callable

Now think of Callable like sending a delivery guy and expecting him to come back with your package. We will use it when we need result from the task.

```

class MyCallable implements Callable<String> {
    public String call() throws Exception {
        return "Task completed!";
    }
}

public class Main {
    public static void main(String[] args) throws Exception {
        ExecutorService service = Executors.newSingleThreadExecutor();
        Future<String> future = service.submit(new MyCallable());

        String result = future.get(); // waits for the result
        System.out.println(result);

        service.shutdown();
    }
}

```

Interview Tip:

Runnable is useful when we just want to execute code. Callable is better when we need the result or want to handle exceptions properly.

Callable is usually used with ExecutorService and Future to get results.

Indirect Questions:

1. Can threads return values in Java?

Yes, by using Callable with ExecutorService, threads can return values.
The result is accessed through a Future object.

2. Why does Runnable not support checked exceptions?

Because Runnable.run() doesn't declare throws Exception, so checked exceptions aren't allowed.
We have to handle them manually inside the method.

3. How would you handle result + exception in a thread?

Use Callable and submit it to an ExecutorService.

Then call future.get() , it gives the result or throws the exception.

4. How cancel the long running callable

We can wrap the Callable in a Future using an ExecutorService. Then call future.cancel(true) to try interrupting the running task.

```
ExecutorService executor = Executors.newSingleThreadExecutor();

Callable<String> longTask = () -> {
    //Code
};

Future<String> future = executor.submit(longTask);

System.out.println("Cancelling task...");
future.cancel(true);
```

43 What is Thread life cycle?

In Java, a thread goes through various states from its creation to termination. Understanding the thread life cycle is crucial for effective multithreaded programming. Here are the states in a thread's life cycle:

Thread States

1. New (Born State)

When a thread is created (using new Thread() or implementing Runnable) but not yet started.

The thread exists but isn't yet scheduled for execution.

2. Runnable (Ready-to-Run)

After calling start() method, the thread enters the runnable state.

The thread is ready to run and waiting for the thread scheduler to allocate CPU time.

Note: This is sometimes called "Ready" state when the thread is waiting for CPU.

3. Running

When the thread scheduler selects the thread, it moves to the running state.

The run() method is executing.

4. Blocked/Waiting

Blocked: When a thread is temporarily inactive (e.g., waiting for I/O, waiting to acquire a lock).

Waiting: When a thread waits indefinitely for another thread to perform an action (using wait(), join(), etc.).

Timed Waiting: When a thread waits for a specified time (using sleep(time), wait(timeout), etc.).

5. Terminated (Dead)

When the thread completes its run() method or is terminated abnormally.

The thread cannot be restarted (calling start() on a dead thread throws IllegalThreadStateException).

Indirect Questions:

1. What happens if you call start() twice on the same thread?

It throws IllegalThreadStateException. A thread can be started only once.

2. Can a thread go from RUNNABLE to TERMINATED without entering BLOCKED or WAITING?

Yes, if it finishes execution without needing to wait or get blocked.

3. Can a thread be started twice? Why or why not?

No. Once it is dead, it is dead. We have to create a new thread object.

4. What happens if you call start() after run() was already called directly?

It still works. but run() just ran like a normal method, not a thread. Now start() creates a proper thread.

44 What is Executor Service and what are the uses??

Managing threads manually can become messy , like handling 50 workers without any manager. **ExecutorService** is that manager. It decides which thread does what and when. Saves memory, improves performance.

ExecutorService is a high-level API in Java that makes it easy to run and manage multiple threads. Instead of creating a new thread every time, we submit tasks to an executor, and it runs them using a pool of threads behind the scenes.

Technical Notes:

The **ExecutorService** interface is part of the **java.util.concurrent** package

It extends the **Executor** interface, which defines a single method **execute(Runnable)** for executing tasks.

Executors is a utility class in Java that provides factory methods for creating and managing different types of ExecutorService instances.

Always remember to call **shutdown()** when we are done.

```
class MyTask implements Runnable {
    public void run() {
        System.out.println("Running in: " + Thread.currentThread().getName());
    }
}

public class Main {
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(3); // 3 threads in pool
    }
}
```

```

        for (int i = 1; i <= 5; i++) {
            service.execute(new MyTask());
        }

        service.shutdown(); // important to release resources
    }
}

```

Indirect Questions:**1. How do you manage multiple threads in a clean way?**

Use ExecutorService to handle tasks. It creates a thread pool, assigns work, and reuses threads.
No need to manually start or stop threads each time.

2. What's the difference between execute() and submit()?

execute() runs a task but gives no result back.
submit() runs a task and returns a Future .so we can get the result or catch exceptions.

3. Can ExecutorService handle exceptions from tasks?

Yes , but only if we use submit() and check the result with Future.get().
If we use execute(), exceptions may be lost unless we log them manually inside the task.

4. What happens if you don't shut down an executor?

The app will keep running because the executor's threads stay alive.
Always call shutdown() to release resources and stop the thread pool cleanly.

5. How would you design thread pool based on system load?

- By Using ThreadPoolExecutor, ThreadPoolExecutor is a Java class that manages a pool of worker threads to execute tasks asynchronously.
- It allows us to reuse threads, control the number of concurrent threads, queue tasks, and define policies for handling extra tasks when the pool is full.

Reference: [Understanding ThreadPoolExecutor in Java | by Pranav Tiwari | Medium](#)

45 What is Synchronization?

In multithreaded applications, multiple threads can attempt to access and modify the same shared resources (e.g., variables, objects, files) concurrently. Without synchronization, this concurrent access can lead to unpredictable and incorrect results.

Synchronization in Java is a mechanism that controls access to shared resources by multiple threads in a multithreaded environment. It ensures that only one thread can execute a critical section of code

Real-World Example (I did not get any best example than this 😊)

- Imagine a public toilet with one door and a key.
- If one person is inside and locks the door, no one else can enter.
- Only after they unlock and exit can the next person go in.

This is how synchronized works in Java. It gives the first thread a key (a monitor/lock) and blocks others until it is done.

Indirect Questions:

1. How do you avoid race conditions in multithreaded Java code?

Use synchronized, Locks, or thread-safe data structures to ensure only one thread accesses critical code at a time.

This prevents threads from messing with shared data at the same time.

2. What happens if two threads access the same variable at the same time?

If that variable isn't protected (no lock), both threads might read or write at the same time, causing unpredictable results.

This is how race conditions and weird bugs happen.

3. Can two threads call the same synchronized method at once?

No, if the method is synchronized on the same object, only one thread can enter it at a time.

The other thread waits until the first one releases the lock.

4. Does synchronized make Java objects thread-safe?

Only the synchronized parts are safe, the rest of the object can still be accessed unsafely.

5. How would you ensure that a piece of code is executed by only one thread at a time?

Use synchronized block or method.

6. Is Synchronized thread safe?

Yes, synchronized is a keyword and a mechanism used to achieve thread safety in languages like Java, ensuring that only one thread can access a shared resource or critical section of code at a time, thus preventing data inconsistency

7. If static synchronized method and instance synchronized called simultaneously on same object, will it conflict?

No, they won't conflict.

- A static synchronized method locks on the Class object (ClassName.class).
- An instance synchronized method locks on the current instance (this).

46 Difference Between Synchronized Method and Synchronized Block?

As a part of Synchronization, we will lock some resource (method or object) using **synchronized** keyword.

But we don't always want to lock everything. Locking too much can slow down the app. So, we have two options: Lock entire method called **synchronized methods** or lock only that particular section of the code calling **synchronized blocks**

Real time example:

In our cupboard, there can be multiple drawers. We have two options:

1. Have a single door and lock entire cupboard (Synchronized Method).
2. Have multiple doors and lock only particular cupboard where valuable things are placed. This is actually a better option (Synchronized Block).

Synchronized Method:

When we add synchronized to a method, the entire method gets locked.

```
public synchronized void updateBalance() {  
    // Whole method is locked  
    balance += 100;  
}
```

Code: [JavaSamples/SynchronizedMethodExample.java at main · CodingLyf-Fullstack/JavaSamples](#)

Synchronized Block

Here, we only lock the particular section of the code.

```
public void updateBalance() {  
  
    synchronized (this) {  
        balance += 100; // Only this part is locked  
    }  
}
```

Code: [JavaSamples/SynchronizedBlockExample.java at main · CodingLyf-Fullstack/JavaSamples](#)

Indirect Questions:

1. What's the performance impact of using synchronized?

It can slow things down if too many threads keep waiting to enter locked code.
Use it only where needed, or the program will become slow.

2. How do you lock only part of a method?

Just wrap the critical code inside a synchronized block instead of the whole method.
That way, only that small section gets locked. It is faster, cleaner, and safer.

3. Can I use multiple locks in one class?

Yes, we can create different lock objects and sync on each. It is useful when we want to protect different data separately.

Think of it like having separate keys for different drawers instead of locking the whole cupboard.

4. What happens if I synchronize too much?

Too much locking means threads keep waiting, and the app becomes slow or stuck.
It is like blocking the whole road for a bike to pass. It is not efficient at all.

5. If two threads try to access the same synchronized method, what happens to the second one?

It waits and gets into blocked state until the first one releases the lock.

6. Can a thread acquire multiple locks?

Yes, a thread can acquire multiple locks.

7. What if order of acquiring multiple locks is inconsistent by threads?

If multiple threads acquire multiple locks in different orders, it can lead to a deadlock.

47 Difference of wait() , notify() and notifyAll()?

In a multithreaded program, threads often need to wait for some condition to happen or signal other threads when they're done.

That is where wait(), notify(), and notifyAll() come into play. These are methods defined in the Object class , not in Thread class, because in Java, every object can act as a lock.

Real-World Example:

Imagine there's only one phone booth in your office.

- If someone is inside, the others wait.
- When the person inside comes out, they say "done" then only the next person can go.

That is wait() (someone waiting), and notify() (someone getting informed that they can go in).

wait(): Causes the current thread to pause its execution and release the lock it holds on the object.

Thread goes to waiting state.

notify(): wakes up **one** thread that is waiting on the same object

notifyAll(): wakes up **all** waiting threads (only one can proceed at a time)

ProducerConsumer Program, one of the frequently asked Programs

Code: [JavaSamples/ProducerConsumer.java at main · CodingLyf-Fullstack/JavaSamples](#)

Possible Indirect Questions:

1. What's the difference between sleep() and wait()?

sleep() just pauses the thread but doesn't release the lock.

wait() pauses and also gives up the lock so other threads can work.

2. Can I use wait() without synchronization?

No, if we call wait() without holding the object's lock, Java throws an IllegalMonitorStateException.

Always use it inside a synchronized block.

3. Can multiple threads wait on the same object?

Yes, many threads can call `wait()` on the same object.

When `notify()` or `notifyAll()` is called, one or all of them will wake up.

4. What happens if a Thread calls notify where no threads are in waiting state?

If a thread calls `notify()` but no other thread is currently waiting on the same monitor (object), the `notify()` method will simply do nothing.

5. What happens if you call notify() without holding the object's lock?

It throws `IllegalMonitorStateException`. We must own the lock to call `notify()`.

6. What will happen if you call wait() outside synchronized?

It throws `IllegalMonitorStateException`. `wait()` needs the lock too.

48 Difference of sleep(), wait() and join()?

sleep()

- `sleep()` is a static method in the `Thread` class.
- It causes the currently executing thread to pause for a specified time duration (in milliseconds).
- It does not release any locks held by the thread.
- It is used to introduce a delay in the execution of a thread.

wait()

- `wait()` is an instance method defined in the `Object` class, but it is typically called on an object within a synchronized block.
- It is used for thread synchronization and inter-thread communication.
- When a thread calls `wait()`, it releases the lock on the object and is suspended until another thread calls `notify()` or `notifyAll()` on the same object.
- It is crucial for scenarios where one thread needs to wait for a specific condition to be met by another thread.

join()

- `join()` is a method of the `Thread` class.
- It allows one thread to wait for the completion of another thread.
- When thread A calls `threadB.join()`, thread A waits until `threadB` finishes

Possible Indirect Questions:

1. If a thread is sleeping, can another thread enter its synchronized block?

No, because `sleep()` doesn't release the lock. Other threads still have to wait.

2. Will a sleeping thread respond to notify()?

Nope, `notify` only works on threads that called `wait()`, not `sleep()`.

3. How can you ensure that Thread A completes its execution before Thread B starts?

Call `A.join()` inside `B`. That way, `B` waits until `A` finishes.

4. Can join() cause a deadlock?

Yes, if two threads are calling `join()` on each other, they'll wait forever.

5. Can you pause a thread without using sleep() or wait()?

Yeah, use join()

49 Difference between Volatile and Atomic?

Volatile:

Imagine you have two threads, one is writing to a variable; another is reading it.

By default, Java threads can cache variables locally. That means, Thread A might write a value, but Thread B still sees an old one.

When we mark a variable as volatile, we are saying:

"Don't cache this. Always read and write from main memory."

So now, when one thread updates a volatile variable, the change is immediately visible to other threads.

Here is the catch

volatile does not guarantee atomicity for compound operations.

For example, `volatile int count = 0; count++;` is not an atomic operation.

It involves a read, an increment, and a write, which can be interrupted by other threads, leading to race conditions.

To achieve atomicity, we use Atomic Classes.

Atomic:

Atomic classes are for atomicity and visibility. Atomic classes make sure that when one thread changes a value, it does it completely and no one else can mess with it halfway. That is called atomicity. At the same time, visibility means: once a thread updates the value, other threads can immediately see the new value. There's no delay.

```
AtomicInteger count = new AtomicInteger(0);

public void increment() {
    count.incrementAndGet(); // Atomic and safe
}
```

Reference: [Exploring Thread Safety with AtomicInteger in Java](#)

Interview Tip for threads:

- Mention about ExecutorService over raw Thread for pooling and scalability. Use Runnable/Callable instead of extending Thread.
- Tell about synchronized for simple locking, ReentrantLock for advanced control, and AtomicInteger or other atomic classes for counters.

- Mention about apply volatile for shared flags where visibility matters, but remember it doesn't ensure atomicity.

Possible Indirect Questions:

1. How does AtomicInteger work internally?

It uses something called Compare-And-Swap (CAS), a low-level CPU instruction.

2. Would you use volatile for a counter?

No, it is not safe for counters.

Because even though threads can see the latest value, they can still overwrite each other's updates. So, always use AtomicInteger or synchronized when we are doing read-modify-write.

3. What's the difference between synchronized, volatile, and Atomic?

- Volatile is like saying "everyone, please read the latest value." That is it.
- Synchronized adds a lock: only one thread can enter that block at a time.
- Atomic gives fast, lock-free operations but only for specific things like counters or flags.

4. Is AtomicInteger better than synchronized?

Depends on what we are doing. For simple operations like incrementing a number, AtomicInteger is faster. But if we are doing multiple steps together (like check, then update), synchronized gives us a better control. Atomic operations use lock free so they are faster compared to synchronized

5. Can a volatile provide thread safety?

No. volatile does not provide full thread safety. It only ensures visibility of changes across threads, but doesn't make compound actions (like incrementing a counter) atomic.

6. How visibility will be ensured in multithreaded programming.

Using volatile

50 Difference between Future and CompletableFuture?

Future and CompletableFuture , both help with asynchronous programming, but they're quite different in how they work.

Real time example:

Imagine you are ordering a pizza.

Scenario A: You went to the restaurant and ordered and wait there on the table. You are not doing anything else. Just sitting and waiting for your order. That is **Future**.

Scenario B: You place the order through an app from home. While it is being prepared, you watch Netflix, take a shower, and the app notifies us when it is done. That is **CompletableFuture**.

Reference: [CompletableFuture vs. Future in Java - Java Code Geeks](#)

1: If two APIs return values and you want to process them together?

We can call both APIs using CompletableFuture. Use CompletableFuture.allOf() to wait for both to finish. Once they're done, we can get both results and combine them however we want.

2: How do you run a task asynchronously in Java?

Just wrap the task using CompletableFuture.runAsync() or supplyAsync().

This tells Java to run it in the background thread, so the main thread doesn't get stuck waiting.

51 How Java handles exceptions?

An exception in Java is just an unexpected situation that breaks the normal flow of the program. Java handles exceptions using a structured mechanism that involves keywords: try, catch, finally,

try block: This block encloses the code segment that is susceptible to throwing an exception.

catch block: If an exception occurs within the try block, the corresponding catch block is executed. A try block can have multiple catch blocks to handle different exception types.

finally block: This block is optional and contains code that is guaranteed to execute, regardless of whether an exception occurred in the try block or was caught by a catch block. It is typically used for cleanup operations like closing resources.

```
try {
    BufferedReader reader = new BufferedReader(new FileReader("data.txt"));
    String line = reader.readLine();
    System.out.println(line);
} catch (IOException e) {
    System.out.println("Something went wrong while reading the file.");
    e.printStackTrace();
} finally {
    System.out.println("Cleanup code goes here.");
}
```

Possible Indirect Questions:

1. Difference between Exception vs Error

Exceptions are issues in the code that can be handled; Errors are serious problems like memory leaks the program can't handle.

2. Can we write only try block without catch and finally blocks?

No, a try block must be followed by either catch or finally. Java won't compile without it.

3. Does remaining statements in try block execute after exception occurs?

No, once an exception is thrown, the remaining code in that try block is skipped.

4. What Happens When an Exception Is Thrown by the Main Method?

If not caught, JVM prints the stack trace and exits the program.

5. Does a try contain multiple catch blocks?

Yes, we can have multiple catch blocks to handle different exception types.

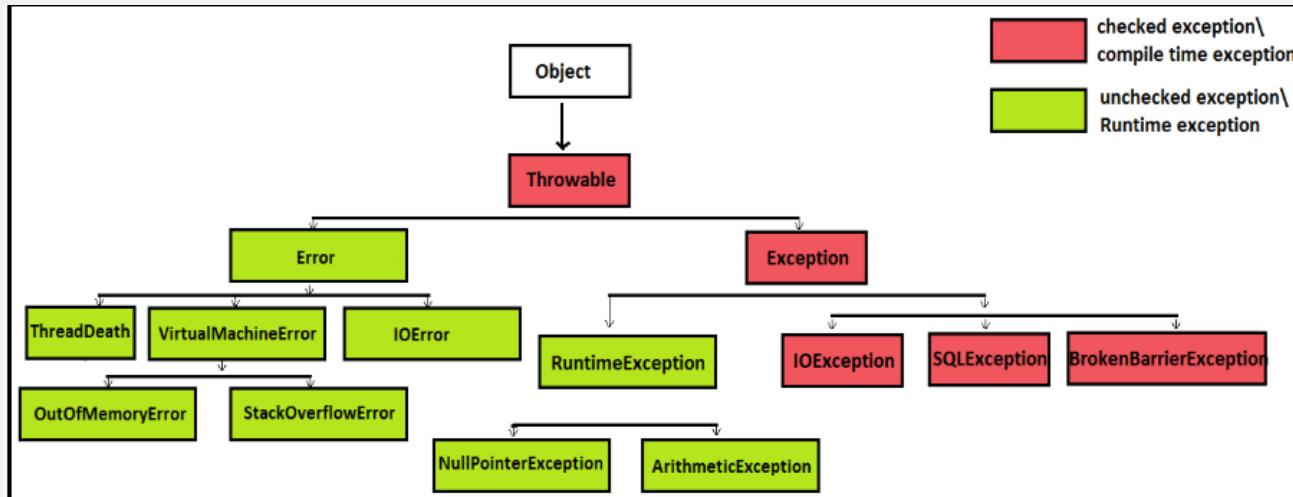
6. What if we return from try, will finally block still execute?

Yes, finally will run even if try returns early.

7. When will finally won't execute?

With System.exit(0) Program exists, so finally will not be invoked.

8. Explain Exception hierarchy



At the top:

- **Throwable**: The root of all errors and exceptions in Java.

Two main branches:

- **Error**
 - Serious problems that applications shouldn't try to handle.
 - Examples: **OutOfMemoryError**, **StackOverflowError**.
 - Usually caused by the JVM or environment.
- **Exception**
 - Problems an application *can* catch and handle.

Interview Tip:

List out all the exceptions you have faced in your project and explain how to handled and how you fixed the errors.

52 What is the difference between checked and unchecked exceptions?

Checked exceptions: Compiler forces us to handle these.

Checked exceptions are checked by the compiler at compile time, and the programmer must handle them (either by catching them or declaring them in the method signature) or they will cause a compilation error

- **IOException**: File not found

- SQLException: Database down
- ParseException: Badly formatted input

Unchecked exceptions: we can handle them, but the compiler won't force us.

These are **runtime bugs**. Errors that happen because of mistakes inside the code.

- NullPointerException – Accessing Null
- ArrayIndexOutOfBoundsException – Accessing Invalid index from an Array
- IllegalArgumentException

```
int divide(int a, int b) {
    return a / b; // ArithmeticException if b is 0
}
```

Indirect questions:

1. Why does Java force you to catch IOException but not NullPointerException?

Because IOException comes from things *outside* the code , like reading a file, connecting to a server. We can't always predict if the file will be there or the server will respond. Java says, "This can fail in real life, so handle it properly."

But NullPointerException? That is the code mistake. we tried to use something that was null. Java cant predict it. So we need to handle explicitly

2. What happens if you don't handle a checked exception?

Java just won't compile the code.

53 What is the difference between throw and throws?

Throw:

The throw keyword is used to **manually** throw an exception in the code

When throw is executed, the normal flow of execution is immediately terminated, and the control jumps to the nearest catch block that can handle the thrown exception. If no such catch block is found, the exception propagates up the call stack.

Real-World Example:

Imagine We are writing a banking app. If a user tries to withdraw more money than they have, we throw an InsufficientFundsException.

```
public class BankAccount {
    private double balance;

    public void withdraw(double amount) {
        if (amount > balance) {
            throw new IllegalArgumentException("Insufficient funds!");
    }
}
```

```

        balance -= amount;
    }
}

```

Throws:

`throws` is used in a method's signature to declare the types of checked exceptions that the method might throw. This informs calling methods that they need to either handle these exceptions (using try-catch) or declare them as well (propagating them further).

Real World Example: We are writing a file reader method. Since files can be missing or corrupted, Java forces us to declare `throws IOException` to warn callers.

```

public class FileProcessor {
    // Warns that this method might throw an IOException
    public String readFirstLine(String filePath) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(filePath));
        return reader.readLine();
    }
}

```

| | | |
|-----------------------------|--|---|
| Feature | throw | throws |
| Purpose | Actually throws an exception (at runtime) | Declares that a method might throw exception(s) |
| Used in | Method body (inside the method logic) | Method signature (next to method name) |
| Number of exceptions | Only one exception can be thrown at a time | Can declare multiple exceptions, comma-separated |
| Syntax | <code>throw new ExceptionType("message");</code> | <code>public void method() throws Exception1, Exception2</code> |
| When it is used | When you want to manually throw an exception (like validation fails) | When your method deals with checked exceptions and you don't want to handle them inside |
| Can be used with | Both checked and unchecked exceptions | Mostly used for checked exceptions (for unchecked it is optional) |
| Example | <code>throw new IllegalArgumentException("Invalid input");</code> | <code>void readFile() throws IOException</code> |

Indirect Questions:**1. Can we use `throw` without `throws`?**

We can use `throw` for unchecked exceptions (like `NullPointerException`) without `throws`.

But for checked exceptions (like IOException), if we throw them, we must declare throws or handle them. Otherwise, Java will throw compile-time error.

2. What happens if I don't declare a throws for a checked exception?

If we don't catch it with try-catch or declare it with throws, our code just won't compile.

3. Can we use throw with multiple exceptions?

No. throw is for throwing a single exception instance at a time.

4. Can you override a method and change the exceptions it throws?

Yes, but we can throw narrower checked exceptions.

We can't add new broader checked exceptions in the child class

54 What are custom exceptions and how create it?

Sometimes, Java's built-in exceptions like NullPointerException, IOException, or IllegalArgumentException don't exactly say what went wrong in the project.

Real-world Example:

Let's say we are building an e-commerce app, and a user tries to order more than the available stock. There's no OutOfStockException in Java. In such situations we create Custom Exceptions

How to create it?

We create custom exceptions either extending Exception class or RuntimeException.

When to extend Exception vs RuntimeException?

If the exceptional condition is recoverable or expected to be handled by the caller, consider extending **Exception**.

If the exceptional condition is unexpected or typically indicates a programming error, extend **RuntimeException**.

Examples:

extending Exception: OutOfStockException

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

    public void placeOrder(int quantity) throws OutOfStockException {
        if (quantity > stock) {
            throw new OutOfStockException("Only " + stock + " items left.");
        }
    }
}
```

extending RuntimeException: InvalidAgeException

Let's say we are implementing Registration API where age must be greater than 0. In such scenarios we want to terminate the program by throwing the exception.

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

    public void registerUser(int age) {
        if (age < 0 || age > 120) {
            throw new InvalidAgeException("Age must be between 0 and 120.");
        }
    }
}
```

Reference: [Custom Exceptions in Java. What is a Custom Exception? | by Priya Salvi | Medium](#)

Interview Tip

- Use checked exceptions when the caller must handle the error (e.g., IOException).
- Use unchecked exceptions (RuntimeException) for programming errors that can't be reasonably recovered.
- Use try-catch-finally to handle exceptions and clean up resources.
- Prefer try-with-resources for automatic resource management (e.g., streams, DB connections).
- Create custom exceptions to provide meaningful, domain-specific error messages.
- Always log or rethrow exceptions; never leave catch blocks empty.

55 Explain equals() and hashCode() contract?

The equals method determines whether two objects are equal or not. By default, the implementation in the `java.lang.Object` class compares memory addresses. To compare objects logically, we need to override this method in our class.

The hashCode method returns an integer hash code that represents the object. This is used in hashing-based collections to efficiently locate the objects.

Contract:

- If two objects are equal (`equals()` returns true), they must have the same `hashCode()` value.
- If two objects have the same `hashCode()`, they might still not be equal.
it is called a hash collision.

Reference: [Java "equals & hashCode" Contract | Design Principles](#)

[Understanding the Equals and HashCode Contract in Java | by Kunal Bhangale | Medium](#)

Indirect Questions:

1. Why do we need to override equals and hashCode method?

- Hash Based Collections uses `hashCode()` first to find the “bucket” and then `equals()` to check if the object already exists.

- If we override only equals() but not hashCode(), the collection might put the same “equal” object in multiple places because the hash codes differ.
- If we override only hashCode() but not equals(), we might get weird equality results where two objects in the same bucket are still considered different.

```

class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // @Override
    // public boolean equals(Object o) {
    //     if (this == o) return true;
    //     if (!(o instanceof Person)) return false;
    //     Person p = (Person) o;
    //     return age == p.age && name.equals(p.name);
    // }
    //
    // @Override
    // public int hashCode() {
    //     return Objects.hash(name, age);
    // }
}

public class EqualsHashCode {
    public static void main(String[] args) {
        Person person1 = new Person("CodingLyf", 1);
        Person person2 = new Person("CodingLyf", 1);

        HashSet<Person> set = new HashSet<>();
        set.add(person1);
        set.add(person2);

        System.err.println(set.size());
    }
}

```

Explanation: Here we have two person object that are logically same, same name and same age.

If we don't override equals and hashCode(), Set will store them two times as their hashcodes are different. (we will get set.size() has 2)

If we override equals and hashCode(), Set size will be 1. Means it store the object only once by checking equals and hashCode methods.

2. What if hash function returns same values how it saves values in HashMap?

If two keys have the same hash code, the HashMap puts them in the same bucket (a linked list or a balanced tree from Java 8).

Then it uses the key's equals() method to differentiate them and store/retrieve the right value.

56 What do you know about Collection in Java?

Interview Tip: Collections are the most important part of day-to-day work in real time projects. So, we can expect good number of questions and programs using collections.

Things we need to remember while preparing or working with collections are:

1. Internal working of each of them
2. Performance implications
3. Memory consumption
4. Order of the items
5. Synchronized or not
6. Handling of null values
7. How they maintain sort order or not
8. Differences between them
9. Use cases means, in which scenario we use which collection.

All these points were discussed for each collection in the upcoming questions.

Definition:

In Java, a collection is a framework that provides an architecture for storing and manipulating a collection of objects. In JDK 1.2, a new framework called "Collection Framework" was created, which contains all of the collection classes and interfaces.

Collection: This is the main interface of the framework. It defines fundamental operations that can be performed on any collection of objects, including adding, removing, and checking for the presence of elements.

In Java, there are **two main building blocks** for storing groups of objects:

1. **Collection interface** (java.util.Collection). It is used for things like Lists, Sets, and Queues.
2. **Map interface** (java.util.Map). It is used for key-value pairs, like a dictionary.

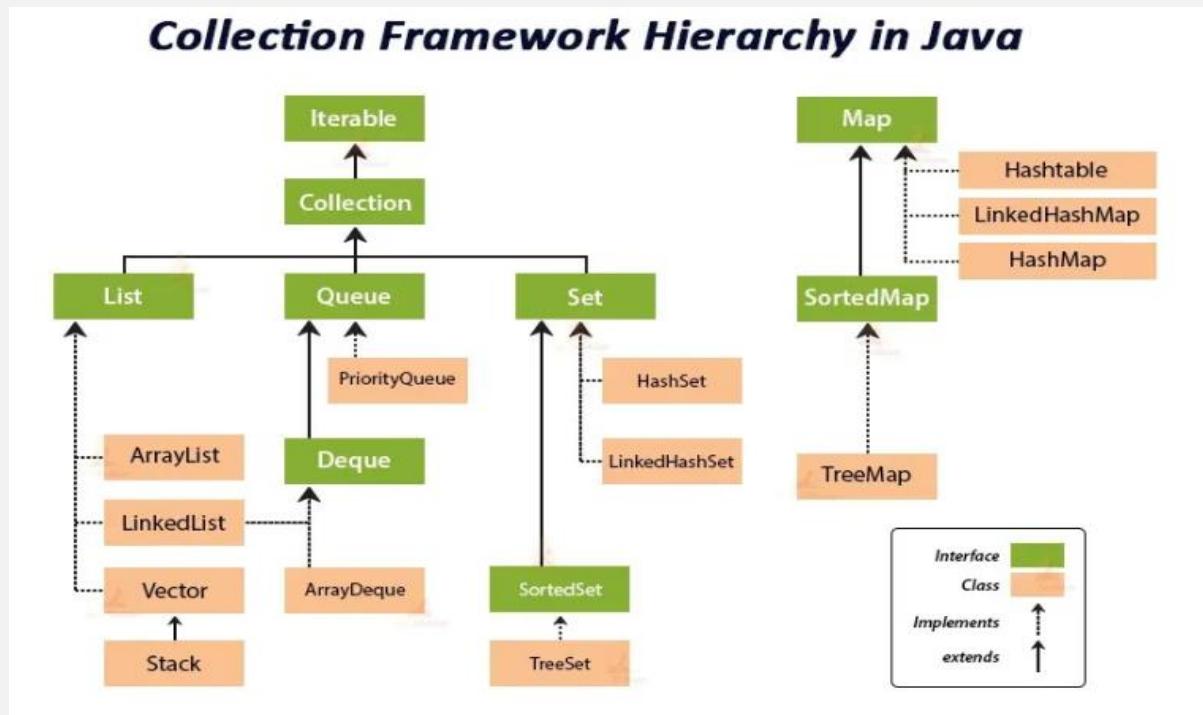
Under the **Collection interface**, we have several types:

- **List**, for an ordered list (e.g., ArrayList, LinkedList, Vector)
- **Set**, to stores unique items (e.g., HashSet, LinkedHashSet, TreeSet)
- **Queue / Deque**, for managing elements in a certain order (e.g., PriorityQueue, ArrayDeque)

Under the **Map interface**, we have:

- HashMap, TreeMap, LinkedHashMap, etc., which store **key-value pairs**.

So, the **Java Collection Framework** is a group of **interfaces and classes** that helps to work with different types of data structures easily.



Reference: [Java Collections Framework](#)

57 What is the difference between List, Set and Map?

List:

A List is like a list of items. It keeps the order in which we add things. We can store duplicates. We can access items by their position (index).

So, if we add apple, banana, and apple again , it will store all of them in the order we added them.

Set:

A Set is like a bag of unique items. It doesn't allow duplicates. Also, it doesn't care about order (unless we use something like LinkedHashSet or TreeSet).

If we try to add apple, banana, and apple again , it'll silently ignore the second apple.

Map:

A Map is for key-value pairs. Think of it like a dictionary where each word (key) maps to a meaning (value). The keys must be unique, but values can repeat.

| List | Set | Map |
|---------------------------|-----------------------------------|--|
| Allows duplicate elements | Does not allow duplicate elements | Does not allow duplicate keys (values can be duplicated) |

| List | Set | Map |
|---|--|---|
| Maintains insertion order | Does not maintain insertion order (except HashSet) | Does not maintain insertion order (except LinkedHashMap) |
| Can add any number of null values | Allows at most one null value | Allows one null key and multiple null values |
| Implementation classes: ArrayList, LinkedList | Implementation classes: HashSet, LinkedHashSet, TreeSet | Implementation classes: HashMap, Hashtable, TreeMap, ConcurrentHashMap, LinkedHashMap |
| Has get(index) method | No get(index) method | No get(index) method, but can get by key |
| Best when we need index-based access | Best when we need a collection of unique elements | Best when we need to store key-value pairs |
| Use ListIterator to traverse | Use Iterator to traverse | Use keySet(), values(), or entrySet() to traverse |

Interview Tip:

Always start by explaining the use-case: “use a List when I need order and allow duplicates, a Set when uniqueness is required, and a Map when I need key-value pairs.”

Mention common implementations and why: “For List, ArrayList for fast access, LinkedList for frequent insert/remove. For Set, HashSet for speed, TreeSet for sorting. For Map, HashMap for general use, TreeMap for sorted keys.”

Tell about performance: “ArrayList get/put is O(1) on average; LinkedList insert/remove is O(1) if at node, but access is O(n). HashMap operations are O(1).”

Always mention real-world scenarios: “Pick LinkedHashMap if I need insertion order, or ConcurrentHashMap for multi-threaded caching.”

Indirect Questions

1. Can a List contain null values?

Yes, a List can hold any number of null values.

2. What about Set and Map?

A Set allows only one null value, and a Map allows one null key and many null values.

3. Can a Map contain null keys?

Yes, a Map (like HashMap) can have one null key.

4. What happens when you add the same element twice in a Set?

The duplicate is ignored because Set keeps only one copy.

5. How would you ensure only unique usernames are allowed?

Store usernames in a Set or use them as keys in a Map.

6. What happens if you try to add a duplicate key in a HashMap?

The old value is overwritten with the new one.

7. Why does not collection interface extend the Map interface?

The Collection interface does not extend Map because they represent different concepts. A Collection is a group of individual elements, while a Map is a group of key-value pairs.

58 ArrayList vs LinkedList?

Internal Structure:

ArrayList: An ArrayList is backed by a dynamic array. This means it can resize itself as needed, but elements are stored contiguously in memory.

LinkedList: A LinkedList uses a doubly linked list. Each element (node) stores a reference to the next and previous nodes.

Access:

ArrayList : Accessing an element by its index (e.g., get(index)) is very fast ($O(1)$ time complexity) because the array provides direct access to elements.

LinkedList: Accessing an element by index is slower ($O(n)$ time complexity) because we need to traverse the list from the beginning or end until we reach the desired index.

Insertions/Deletions:

ArrayList: Inserting or deleting elements in the middle of an ArrayList can be slow ($O(n)$ time complexity) because it may require shifting elements to make space or close gaps. Appending or removing from the end is faster.

LinkedList: Inserting or deleting elements anywhere in the list is fast ($O(1)$ on average for insertions/deletions at the beginning or end) because it only involves updating node references.

Memory Usage:

ArrayList: Generally more memory-efficient as it only stores the element data.

LinkedList: Consumes more memory than ArrayList due to the overhead of storing references to the next and previous nodes.

Use Cases:

ArrayList: Best for scenarios where we need frequent random access to elements and modifications (insertions/deletions) are not a primary concern.

LinkedList: Ideal for scenarios where we need frequent insertions or deletions, especially at the beginning or end of the list, and random access is not a primary requirement.

Reference: [Difference Between ArrayList and LinkedList in Java | by Ramesh Fadatare | Medium](#)

Indirect Questions:

1. How does Java manage dynamic resizing in ArrayList?

When the internal array is full, ArrayList creates a new array with 50% more capacity. It then copies all elements into the new array.

2. Why is LinkedList not preferred for random access?

LinkedList stores elements as nodes with references to next and previous.

Accessing an element at index i requires traversal from head or tail, $O(n)$ time.

So it is slow compared to ArrayList's $O(1)$ where we can access via indexing.

3. If you are inserting an element in the middle of a LinkedList with 10 million items, how will it perform compared to an ArrayList?

LinkedList: Inserting in the middle requires traversing to that position ($O(n)$) and then updating pointers ($O(1)$). So overall $O(n)$.

ArrayList: Inserting in the middle requires shifting all elements after that index ($O(n)$).

For 10 million items:

- Both are $O(n)$, but in practice, LinkedList can be slightly faster for insertion since no shifting of elements is needed, only pointer updates.

Rule:

- Use LinkedList if frequent insertions/deletions in the middle are needed.
- Use ArrayList if random access and iteration speed matter more.

4. If you need to implement a queue that frequently adds and removes elements from both ends, which list would be a better choice and why?

Pick LinkedList over ArrayList.

LinkedList implements Deque, so it is designed for efficient insertions and deletions at both ends in $O(1)$ time.

ArrayList is backed by an array, so adding/removing at the front or middle requires shifting elements, which is $O(n)$.

5. Defining the constructor capacity in ArrayList increases performance?

Yes, it can.

An ArrayList in Java grows dynamically. If we don't set the capacity, it starts with a default size (10) and when more elements are added, it creates a new bigger array (usually 1.5x of old size), copies all elements into it, and then continues.

This resizing and copying cost time and memory.

So, if we know in advance that we will store, say, 10,000 items, defining the constructor like:

`List<Integer> list = new ArrayList<>(10000);` it saves multiple resizes and improves performance, especially in large datasets.

59 ArrayList vs Vector?

Internal Structure

ArrayList and **Vector** both use a **dynamic array** internally.

The difference is,

- **ArrayList** increases size by **50%** when it is full.
- **Vector** increases size by **100%** (it doubles).

So, **Vector** tends to allocate more memory when resizing. That can lead to slightly fewer resizes but more unused memory.

Access

Access speed is the same for both , **O(1)** for reading any element using `.get(index)` because both are array-based. So if the main job is reading by index, both perform well.

Insertions/Deletions

- If we insert/delete at the end, both are fast **O(1)**.
- If we insert/delete in the middle or beginning, performance is bad **O(n)** , because elements have to be shifted.

So again, they behave similarly here.

Memory Usage

- **ArrayList** is more memory-efficient. It resizes conservatively.
- **Vector** tends to waste memory by doubling its size on every resize.

That means if we are tight on memory, go with **ArrayList**.

Thread Safety: This is the biggest difference.

- **Vector** is **synchronized** , meaning all its methods are thread-safe.
- **ArrayList** is **not synchronized** , it is not safe to use with multiple threads unless we handle locking yourself or wrap it with `Collections.synchronizedList()`.

Synchronized does not mean fast. **Vector** is thread-safe but slower due to locking. if we need thread safety, we need to use **CopyOnWriteArrayList** or **ConcurrentLinkedQueue** depending on the use case.

60 HashMap vs TreeMap vs LinkedHashMap?

1. Internal Structure

- **HashMap**: Uses a hash table (like a big index system) to store data.
- **LinkedHashMap**: Same as **HashMap**, but also keeps a linked list to remember the order in which items were added.
- **TreeMap**: Stores data in a sorted tree structure (like a family tree), it keeps keys in order.

2. Order of Elements

- **HashMap**: No order , items can come in any sequence when we retrieve them.
- **LinkedHashMap**: Remembers insertion order. Items come out in the same order we put them in.

- TreeMap: Sorts keys automatically (alphabetically or numerically).

3. Performance (Speed)

- HashMap: Fastest for adding, removing, and finding items.
- LinkedHashMap: Slightly slower than HashMap because it also tracks insertion order.
- TreeMap: Slower than HashMap because it keeps keys sorted.

4. Handling Null Values

- HashMap: Allows one null key and many null values.
- LinkedHashMap: Same as HashMap , one null key allowed.
- TreeMap: Does NOT allow null keys , but allows null values.

When to Use Which?

Use HashMap: When we need fast access and don't care about order.

Use LinkedHashMap: When we need insertion order (like a history log).

Use TreeMap: When we need sorted keys (like a dictionary).

Indirect Questions:

1. If you want to maintain the insertion order of key-value pairs, which Map implementation would you choose and why?

Use LinkedHashMap. It maintains the order in which we added the keys, so when we iterate, we get the elements in the same order.

2. How would you store user records where the keys must always stay sorted alphabetically?

Go for TreeMap. It automatically keeps keys sorted based on natural order (like alphabetical for strings) or a custom comparator.

3. What would happen if you insert multiple elements with the same hashcode into a Map? How is ordering handled in such a case?

In a HashMap, they go into the same bucket. Order is not guaranteed.

In a LinkedHashMap, they're linked in insertion order even in the same bucket.

In a TreeMap, hashcode doesn't matter, only key ordering does.

4. In what scenarios could maintaining a sorted order of keys lead to performance trade-offs?

When using a TreeMap, every put, get, or remove operation takes O(log n) time due to the underlying Red-Black Tree. This is slower than a HashMap's average O(1), but useful when sorted order is required.

5. If you are building a cache that needs predictable iteration order and fast access, which Map would you use and why?

LinkedHashMap is ideal. It offers fast access like a HashMap and maintains order, which is great for things like LRU (Least Recently Used) caching.

6. How can we make HashMap Synchronized?

```
Map<Integer, String> map = new HashMap<>();
Collections.synchronizedMap(map);
```

7. What are the consequences of non-final fields in a class that is used as a key in a HashMap?

Using non-final fields in a class that is used as a key in a HashMap can cause serious issues because HashMap depends on two things:

1. hashCode(): determines which bucket the key goes into.
2. equals(): determines whether two keys are the same inside that bucket.

Problem:

If the fields used in hashCode() or equals() are **mutable** (non-final), and if we change them after inserting the object into the map, then:

- The key's hashCode() may change, so the object is now in the wrong bucket.
- A lookup using the same key (but with changed values) will fail , we will get null even though the entry exists.

```
class Person {
    String name; // mutable, not final

    Person(String name) {
        this.name = name;
    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Person)) return false;
        Person p = (Person) o;
        return Objects.equals(name, p.name);
    }
}

public class HashMapProblem {
    public static void main(String[] args) {
        Map<Person, String> map = new HashMap<>();

        Person p1 = new Person("Raju");
        map.put(p1, "Engineer");

        // Lookup works
        System.out.println(map.get(p1)); // Engineer

        // change the key
        p1.name = "Raghu";

        // Lookup fails
        System.out.println(map.get(p1)); // null
    }
}
```

- When we put p1 in the map, its hash is based on "Raju".
- After changing p1.name to "Raghu", its hash changes but the entry is still stored under the old bucket.
- So, lookup fails, the object exists in the map, but HashMap can't find it anymore.

8. What happens if we have added mutable object to HashMap and change it.

Check above answer

61 HashMap vs HashTable?

Synchronization:

HashMap: Doesn't provide thread safety.

HashTable: Provides thread safety by synchronizing all its methods, ensuring that only one thread can access the table at a time.

Performance:

HashMap: Due to the lack of synchronization overhead, HashMap generally offers better performance than HashTable

HashTable: The synchronization mechanism adds overhead, potentially impacting performance.

Null keys and values:

HashMap: One null key and multiple null values are permitted.

HashTable: Does not allow, Attempting to insert null keys or values will result in a NullPointerException.

Iterators:

HashMap is Fail-fast , if it is modified after an iterator is created a ConcurrentModificationException is thrown

HashTable is fail-safe, meaning modifications to the table during iteration will not cause an exception

When to Use Which

Choose HashMap , when performance is critical and thread safety is handled externally or is not a concern.

Choose HashTable , when thread safety is a requirement and performance is less of a priority.

In modern Java, ConcurrentHashMap is often preferred over HashTable for concurrent access scenarios as it provides better performance while maintaining thread safety.

Indirect Questions:

1. Is HashMap thread-safe by default?

No, HashMap is not thread-safe. If multiple threads access it concurrently and at least one modifies it, we can get unpredictable results.

2. Why does Hashtable not allow null keys or values?

To avoid ambiguity during lookups and to keep the behaviour consistent across threads, Hashtable disallows null keys and values.

3. Which is faster: HashMap or Hashtable?

HashMap is faster because it is unsynchronized. Hashtable uses locks on every operation, which slows things down in single-threaded scenarios.

4. What is the difference between Iterator and Enumerator?

Iterator is newer, supports remove(), and is fail-fast. Enumerator is older, does not support removal, and is not fail-fast.

5. When should we use ConcurrentHashMap over Hashtable?

Use ConcurrentHashMap when we need high-performance thread-safe operations. It allows concurrent reads and partial concurrent writes.

6. Why is Hashtable considered a legacy class?

Because it was part of early Java and later replaced by better alternatives like HashMap and ConcurrentHashMap in the Collections Framework.

Reference: [Difference Between HashMap and HashTable - Scaler Topics](#)

62 ConcurrentHashMap vs SynchronizedHashMap ?

1) Locking Mechanism:

SynchronizedHashMap maintains object level lock. i.e the whole map is locked..

But in ConcurrentHashMap, the whole map is not locked. The map is divided into number of segments and each segment maintains its own lock.

2) Synchronized operations:

In synchronizedHashMap all operations are synchronized. That means, whatever the operation we want to perform on the map, whether it is read or update, we have to acquire object lock.

But in ConcurrentHashMap, only update operations are synchronized. Read operations are not synchronized.

3) Number of threads

Only one thread can enter into a SynchronizedHashMap.

On the other hand, minimum 16 threads can perform update operations on ConcurrentHashMap at a time

4) Null Keys And Null Values

SynchronizedHashMap allows one null key and any number of null values.

ConcurrentHashMap doesn't allow even a single null key and null values.

5) Nature Of Iterators

Iterators returned by SynchronizedHashMap are fail-fast in nature. i.e they throw ConcurrentModificationException if the map is modified after the creation of iterator.

Iterators returned by ConcurrentHashMap are fail-safe in nature. i.e they DON'T throw ConcurrentModificationException if the map is modified after the creation of iterator.

Possible Indirect Questions:**1. What is ConcurrentModificationException when it occurs?**

A ConcurrentModificationException happens when we try to change a list (like adding or removing items) while looping through it. This can happen in single-threaded code during iteration or in multi-threaded code.

2. How does thread safety work in different map implementations?

HashMap is not thread-safe. Hashtable and Collections.synchronizedMap() lock the whole map, while ConcurrentHashMap uses fine-grained locking.

3. Which map implementation is best for concurrent updates in Java?

ConcurrentHashMap is the best choice. It allows safe concurrent access with better performance than synchronized alternatives.

4. Why is the SynchronizedMap slower during multi-threaded access?

Because it locks the entire map for every operation. This creates contention and reduces performance when multiple threads access it.

5. Does ConcurrentHashMap throw ConcurrentModificationException when modified during iteration?

No, it doesn't. It provides a weakly consistent iterator that doesn't fail if the map is modified during iteration.

6. How to make HashMap thread-safe?

We can wrap it using Collections.synchronizedMap(new HashMap<>()), but it will be slower under high concurrency.

7. What are the alternatives to Hashtable for high concurrency?

Use ConcurrentHashMap. It is faster, non-blocking for reads, and designed specifically for multi-threaded environments.

8. Can we use HashMap in multi-threaded scenario?

The problem arises when there is concurrent update on Hash Map. Hence, during such scenarios, we can use the HashTable or ConcurrentHashMap.

9. Can we insert and delete at time same from SynchronizedMap?

Yes, we can insert and delete at the same time from a SynchronizedMap. Collections.synchronizedMap(...) makes individual operations like put(), get(), and remove() thread-safe by wrapping them in synchronized blocks.

63 Why ConcurrentHashMap won't allow null keys and null values??

In a normal HashMap, we can store one null key and many null values. That is fine because everything happens in a single-threaded context.

But in a ConcurrentHashMap, multiple threads read and write at the same time. If null were allowed, two big problems show up:

The Problem with Null Keys

Null Key in HashMap

In a normal HashMap, if we put null as a key, Java just keeps it in bucket 0. Since only one thread works at a time, that is fine.

Problem in ConcurrentHashMap

ConcurrentHashMap is different:

- It splits data into many buckets.
- Keys go into buckets based on their hashCode().
- This allows many threads to work on different buckets at the same time.

But with null:

- There's no hashCode() to calculate.
- If they force all null keys into bucket 0, then all threads must fight over that one bucket.
- This slows things down and makes concurrency messy.

The Problem with Null Values

Imagine a scenario where null values are allowed. If we call map.get(key) and it returns null, what does that mean? In a single-threaded HashMap, null could mean either:

1. The key doesn't exist in the map.
2. The key exists, but its value is explicitly set to null.

In a concurrent setting, this ambiguity becomes a nightmare:

- Thread A checks map.get(key) and gets null, assuming the key doesn't exist.
- But Thread B might have just set the value to null using map.put(key, null).

Now Thread A can't tell whether the key is absent or its value is null, leading to race conditions or incorrect logic. By banning null values, ConcurrentHashMap ensures that get() returning null always means the key doesn't exist

Reference: [Why Nulls Are Banned: The Surprising Reason ConcurrentHashMap Doesn't Allow Null Keys or Values | by The Code Alchemist | Medium](#)

Additional Question:

1. What is the work around to handle the nulls in ConcurrentHashMap?

Use a Placeholder Object: This is the most common and efficient approach. Create a special, unique object that represents null.

```
public static final Object NULL_VALUE = new Object();
ConcurrentHashMap<String, Object> map = new ConcurrentHashMap<>();

// To "put null"
map.put("key", value == null ? NULL_VALUE : value);
```

```
// To get and check
Object value = map.get("key");
if (value == NULL_VALUE) {
    // Handle the case where you *would* have stored null
} else if (value != null) {
    // Handle actual value
} else {
    // Key is absent
}
```

64 HashSet vs LinkedHashSet vs TreeSet?

1) Ordering:

HashSet: Unordered. Elements are stored based on their hash codes, which means there's no predictable order of iteration.

LinkedHashSet: Maintains insertion order. Elements are stored in the order they were added to the set.

TreeSet: Sorted. Elements are stored in a sorted order, either according to their natural ordering or based on a provided Comparator.

2) Implementation:

HashSet uses a HashMap internally to store elements.

LinkedHashSet uses a LinkedHashMap internally, which maintains a doubly-linked list to track insertion order.

TreeSet uses a TreeMap internally, which is a self-balancing binary search tree.

3) Performance:

Hashset: Generally offers the best performance for add, remove, and contains operations ($O(1)$ on average) due to the efficient hashing mechanism.

LinkedHashSet: Performance is slightly slower than HashSet because of the overhead of maintaining the linked list, but still generally fast ($O(1)$ on average for most operations).

Treeset: Performance is generally slower than both HashSet and LinkedHashSet ($O(\log n)$ for add, remove, and contains operations) because of the sorting overhead.

When to use which?

Use HashSet , when we need to store unique elements and don't care about the order they are stored in, and when we need fast performance

Use LinkedHashSet, when we need to store unique elements and need to iterate through them in the same order they were added.

Use TreeSet, when we need to store unique elements in a sorted order.

Indirect Questions

1. How would you maintain insertion order in a Set?

Use LinkedHashSet

Don't say HashSet, it doesn't maintain order

2. What if I need to always keep the elements sorted?

Use TreeSet. It uses Red-Black Tree internally for ordering

3. How would you implement a deduplicated list of elements while keeping insertion order?

LinkedHashSet, Because it is a Set (removes duplicates) + maintains order

4. Which Set would perform best for search operations?

In most cases, HashSet gives O(1) for contains()

TreeSet takes O(log n)

5. Can I store null in a Set?

HashSet and LinkedHashSet allow one null

TreeSet throws NullPointerException (if using natural ordering)

6. What happens if I insert elements in random order, then iterate the Set?

- In HashSet: iteration order will be unpredictable
- In LinkedHashSet: elements come out in the order they went in
- In TreeSet: elements come out sorted

7. How does TreeSet know how to sort the elements?

It uses compareTo() (natural ordering) Or a custom Comparator if passed during construction

8. Why TreeSet does not allow null?

A **TreeSet** compares each element either by Comparable or Comparator. If TreeSet allows null, then it will throw NullPointerException.

9. What happens if you add elements to a HashSet with duplicate hashCode() values but different equals() implementations?

If two objects have the same hashCode() but equals() returns false, the HashSet will store both objects.

Here is why:

- HashSet first checks the bucket using hashCode().
- Within the bucket, it uses equals() to see if the object already exists.
- Since equals() is false, HashSet treats them as different, even though they collide in the hash bucket

Reference: [HashSet Vs TreeSet Vs LinkedHashSet In Java](#)

65 Explain about Queue?

A **Queue** in Java is a linear data structure that follows the **First-In-First-Out (FIFO)** principle, meaning the first element added to the queue will be the first one to be removed. Queues are widely used in scenarios like task scheduling, buffering and producer consumer problems.

Since the Queue is an interface, we cannot provide the direct implementation of it.

Queues in Java can be classified into **bounded** and **unbounded** implementations based on their capacity.

1. **Unbounded Queue:** Can grow dynamically (no fixed size).
2. **Bounded Queue:** Has a fixed maximum capacity.

66 Explain different types of Queue?

Types of Queue Implementations in Java

Queues in Java can be categorized into two main types based on their capacity:

1. **Unbounded Queue:** Can grow dynamically (no fixed size).
2. **Bounded Queue:** Has a fixed maximum capacity.

1. Bounded Queues

A bounded queue has a fixed size. Once it is full, we can't just keep adding elements, either the operation fails, or it waits until space is available.

- **ArrayBlockingQueue:** It Uses an array with a fixed capacity. It also supports fair ordering if we want threads to access in a strict first-come, first-served manner.
- **LinkedBlockingQueue:** Works like a linked list and we can set the capacity. If we don't, it defaults to a very large size (Integer.MAX_VALUE).
- **PriorityBlockingQueue:** Instead of strict FIFO, it orders elements based on priority (either natural ordering or a custom comparator).

2. Unbounded Queues

Unbounded queues don't force a hard capacity limit. They'll grow as needed, limited only by the available memory.

- **PriorityQueue:** Keeps elements ordered by natural order or a comparator, but it is not thread-safe.
- **ConcurrentLinkedQueue:** A thread-safe option that works well in highly concurrent scenarios.
- **LinkedList (as Queue):** LinkedList also implement Deque. So, it can be also used as Queue.

67 When will you use each queue?

Interviewer either give a scenario or you need to think of a scenario as an example

ArrayBlockingQueue : For fixed-size producer-consumer problems. For example, a logging system where producers (threads writing logs) and consumers (threads processing logs) should (Refer Question No: 181: **Java Scenario 8:** Producer Consumer Problem).

LinkedBlockingQueue:

Useful when we want flexibility in size (bounded but large by default). For example, handling requests in a web server where we may want to queue thousands of tasks.

```
BlockingQueue<Runnable> taskQueue = new LinkedBlockingQueue<>();

// Add tasks
taskQueue.offer(() -> System.out.println("Task 1 executed"));
taskQueue.offer(() -> System.out.println("Task 2 executed"));

// Execute
Runnable task = taskQueue.poll();
if (task != null) task.run();
```

PriorityBlockingQueue

When tasks have priorities, like a hospital system where emergency patients must be treated before routine checkups(Refer Question No:).

PriorityQueue: When we need ordered processing but don't care about thread safety

ConcurrentLinkedQueue: When multiple threads are adding and removing without blocking.

LinkedList (as Queue): For simple, single-threaded use cases where we just need FIFO behavior.

68 How ArrayList works internally?

ArrayList in Java works internally by using a dynamic array to store its elements. This means it is backed by an Object[] array, which holds the actual data.

```
List<Integer> list = new ArrayList<>(10000);
```

1. Initial Capacity

When we create an ArrayList, it starts with a default capacity , which is usually 10, unless we give a specific value: `List<String> list = new ArrayList<>(); // default capacity = 10`

Capacity here means: how many elements can be stored before the array has to grow.

2. Capacity vs Size

- Capacity -> Total slots available in the internal array
- Size -> How many elements we have added so far

Let's say we created a list with default capacity 10, and added 3 elements then

Capacity = 10 and Size = 3

3. How it Grows

When we keep adding items and the array runs out of space, it automatically grows like this:

- A new array is created with more space (typically 1.5x the old size)
- All elements are copied from the old array to the new one
- The old array is discarded

This resizing is done inside a method called `grow()`.

Example:

Old capacity = 10 and new capacity = 15

This resizing takes time, so adding many items at once can temporarily slow things down.

4. Adding Elements

When we call `add(element)`:

- It first checks if there's room in the array
- If yes -> Just put it at the next available index
- If no -> Resize the array, then add

So, `add()` is usually $O(1)$ time, but occasionally it can be slower when resizing happens.

5. Accessing Elements

We can access any element by its index using `.get(index)`

```
list.get(2);
```

This is super-fast; $O(1)$ time

6. Inserting or Deleting in the Middle

Here is the catch:

- If we insert or remove from the middle, it has to shift elements to keep the order.
- That takes time; $O(n)$ time

```
list.add(2, "hello"); // shifts everything from index 2 onward
```

```
list.remove(2); // also shifts everything back
```

All the possible Questions on Array List:

[ArrayList Interview Questions in Java - Scientech Easy](#)

69 How HashMap works internally?

I would suggest you go through the below videos before going to the theory.

Reference: [How HashMap Internally Works in Java With Animation | Popular Java Interview QA | Java Techie](#)

[Map and HashMap in Java with Internal Working- Interview Question](#)

HashMap works by using a hash function to map keys to indices in an internal array (buckets).

When a key-value pair is added, the hash code of the key is used to determine which bucket to store it in. If multiple keys hash to the same index, a linked list (or a balanced tree in Java 8+ for larger lists) is used to store these collisions.

1. Hashing:

When we insert a key-value pair (e.g., `put(key, value)`), the `hashCode()` method of the key object is invoked. The result, along with the current size of the `HashMap`, is used to calculate an index within the internal array.

2. Collision Handling:

If multiple keys generate the same hash code (a hash collision), they are stored in the same bucket. In older versions of Java (before 8), this was handled using a linked list. In Java 8 and later, if the number of elements in a bucket exceeds a threshold (usually 8), the linked list is converted to a balanced tree (red-black tree) for better performance.

3. Retrieval:

When we retrieve a value using `get(key)`, the same hashing process is used to determine the correct bucket. If the key is not directly found in the bucket, the linked list or tree (depending on the implementation) is traversed to find the key-value pair.

4. Resizing:

If the `HashMap` becomes too full (exceeds a load factor threshold, typically 0.75), it will resize its internal array to a larger size, typically doubling it. This process, called rehashing, involves recalculating the hash codes and re-distributing the elements into the new array, which can be a costly operation.

Key Concepts:

`hashCode()`: A method that returns an integer representation of an object. It is crucial for determining the bucket index.

`equals()`: Used to compare keys for equality when collisions occur. If `hashCode()` returns the same value for two keys but `equals()` returns false, it signifies a collision.

Load Factor: A threshold that determines when the `HashMap` should resize.

Indirect Questions:

1.What is the load factor in `HashMap`?

`HashMap`'s performance depends on two things first initial capacity and second load factor. whenever we create `HashMap` initial capacity number of the bucket is created initially and load factor is the criteria to decide when we have to increase the size of `HashMap` when it is about to get full. A load factor is a number that controls the resizing of `HashMap`. When a number of elements in the `HashMap`

cross the load factor as if the load factor is 0.75 and when becoming more than 75% full then resizing trigger which involves array copy.

2. How does resize happens in HashMap?

The resizing happens when the map becomes full or when the size of the map crosses the load factor. For example, if the load factor is 0.75 and then becomes more than 75% full, then resizing trigger, which involves an array copy. First, the size of the bucket is doubled, and then old entries are copied into a new bucket.

3. What is the difference between the capacity and size of HashMap in Java?

The capacity denotes how many entries HashMap can store, and size denotes how many mappings or key/value pair is currently present.

4. Apart from String, what else predefined class we can use as Keys in A Map?

Wrapper classes: Integer, Long, Double, Float, Boolean, Character

5. If a poorly designed hash function causes all elements to collide in a HashMap, what would be the performance impact?

Normally, HashMap lookup is O(1) on average. But If all elements collide, every key ends up in the same bucket. But with collisions, it degrades to O(n) because it must scan through a linked list. That is why Java 8 improved this by switching to a balanced tree after many collisions

70 How HashSet works internally?

I would suggest you go through the below video before going to the theory.

Reference: [Core JAVA: How does HashSet work internally? Implementation | Why are its elements unique?](#)

HashSet uses a **HashMap** behind the scenes to store its elements. Here is how it actually works:

1. HashMap is the backbone

When we add something to a HashSet:

- That value is stored as a **key** in an internal HashMap.
- The **value** for that key is just a constant dummy object (called PRESENT).
- HashMap doesn't allow duplicate keys, this is what keeps all HashSet elements unique.

2. How it decides where to store

When we do set.add("apple"):

- Java calculates the **hashCode** of "apple".
- That hash is used to pick a **bucket** inside the HashMap.
- If the bucket already has something, Java checks using equals() to see if "apple" is already there.
 - If yes, it won't add it again.
 - If no, it stores it alongside the others (usually using a list or tree).

3. What happens with duplicates

If we try to add an element that is already in the set:

- The underlying HashMap.put() will return the old dummy value.
- That tells HashSet the item was already there.
- So, add() returns false.

4. Why it matters

- HashSet depends completely on hashCode() and equals() of the objects we add.
- These two methods determine where the object is placed and whether it is considered a duplicate.

71 How LinkedHashMap and LinkedHashSet works internally?

I would suggest you go through the below videos.

LinkedHashMap and LinkedHashSet: [LinkedHashMap and LinkedHashSet in Java | Internal Working](#)

LinkedHashMap : [How does LinkedHashMap work internally? | VS Hashmap internals? | LRU cache | Is it synchronized?](#)

72 How TreeMap works internally?

I would suggest you go through the below video.

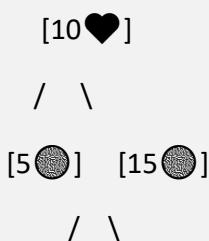
 [How TreeMap Works Internally in java | #programmingkt #java #javatutorial #javatutorialforbeginners](#)

The java.util.TreeMap class in Java's Collections Framework is an implementation of a Red-Black Tree. It provides a sorted map where elements are ordered by their natural ordering of keys, or by a custom Comparator.

When we don't provide a Comparator, TreeMap uses the natural ordering of keys. That means the key class must implement Comparable interface.

Visual of a Red-Black Tree

Imagine this TreeMap with numbers as keys:



[12♥] [20♥]

Legend:

-  = Red node
-  = Black node
- Each node = key-value pair
- Always follows Red-Black tree rules (Below are the rules good to know)
 - **Node color:** Every node is either red or black.
 - **Root rule:** The root is always black.
 - **Leaf rule:** All leaves (null children, considered "NIL" nodes) are black.
 - **Red rule:** If a node is red, both its children must be black (no two reds in a row).
 - **Black-height rule:** Every path from a node to its descendant NIL leaves must have the same number of black nodes.

How TreeMap Maintains Sorted Order?

Natural Ordering

If we just write:

```
TreeMap<Integer, String> map = new TreeMap<>();
```

The keys will be sorted in **increasing order**: 1, 2, 3, 4...

Custom Ordering

We can also pass our own logic:

```
TreeMap<String, String> map = new TreeMap<>(Comparator.reverseOrder());
```

Now keys will be sorted in **reverse**.

Operations:

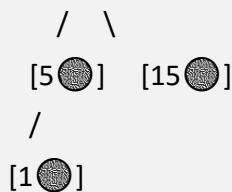
Insertion: put(key, value)

Let's say this is current tree:

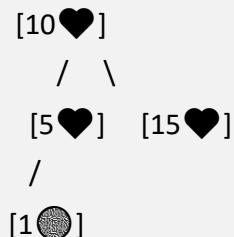


Now if we insert 1 It goes left of 5

[10♥]



But now 5 and 1 are both red. That **breaks the red-red rule**, so the tree **recolors and rotates**:



It stays balanced

Search: get(key)

To find a key, TreeMap walks down the tree:

To find 12 in this tree:

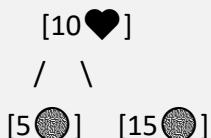


- $12 > 10$, so go right
- $12 < 15$, so go left
- Found it

Only takes a few steps thanks to the balanced structure.

Deletion: remove(key)

If we remove a node, like 10:



TreeMap finds the in-order successor (like 12 or 15), replaces 10 with it, then **re-balances** the tree by rotating/recolouring.

Indirect Questions:

1. Will TreeMap allow key or value?

- TreeMap don't allow null as key but allow null as value

2. Why TreeMap wont allow null key and what will happen if we add null as a key?

- TreeMap maintains its keys in sorted order. To maintain this sorted order, TreeMap needs to compare keys with each other. A null value cannot be compared with a non-null value using these comparison methods. Attempting to do so results in a NullPointerException

- Same with TreeSet as well

73 How TreeSet works internally?

A **TreeSet** in Java stores elements in **sorted order**.

Behind the scenes, it actually uses a **TreeMap** to do all the work. When we create a TreeSet, Java internally creates a TreeMap, and every element we add to the set is stored as a **key** in that TreeMap.

TreeSet implements two important interfaces:

- SortedSet: keeps elements sorted
- NavigableSet: lets us navigate the set (like getting higher, lower, ceiling, floor values)

So even though it behaves like a HashSet (no duplicates), the difference is:

HashSet uses a HashMap (no order).

TreeSet uses a TreeMap (sorted order).

Reference: [TreeSet internal working & treeSet internal implementation in java-JavaGoal](#)

74 How ConcurrentHashMap works internally?

Video 1 : [Internal working of ConcurrentHashMap & Interview Questions - JAVA | Concurrent Collections](#)

Video 2: [ConcurrentHashMap internal working in java | ConcurrentHashMap internal implementation in java](#)

Reference: [How ConcurrentHashMap Works Internally After Java 8?](#)

It is a special kind of Map in Java that allows multiple threads to read and write at the same time without messing up the data.

Before Java 8: Segment-Based

- The map was divided into smaller pieces called segments.
- Each Segment can contain number of buckets.
- Each segment had its own lock.
- If two threads worked on different segments, they could proceed without waiting.
- But still, locking was needed per segment, which made things a bit complex.

. From Java 8 Onwards: Bucket-Based Locking

- Segments were removed.
- Now the map is divided into buckets (like in HashMap).
- Each bucket holds entries (either a linked list or tree).
- Instead of locking the whole map, it only locks the specific bucket being updated.
- This means more threads can work in parallel.

CAS (Compare-And-Swap)

- For small updates like adding or changing a value, it may skip locking altogether.
- It uses CAS, a technique to check and change a value in one atomic step.
- This reduces thread fighting and improves speed.

Internal Operations

Adding or Updating an Entry

- Calculate the hash of the key to find which bucket it belongs to.
- If that bucket is empty, just insert the new entry , so no lock needed.
- If the bucket is not empty:
 - Lock only that bucket.
 - Check if the key exists:
 - If yes, update the value.
 - If not, add the new entry.
 - Then release the lock.
 - In simple cases, CAS may be used instead of locking.

Reading an Entry

- No locking is needed for reads.
- Thanks to special memory rules (volatile), the reading thread will see the most up-to-date value.

Removing an Entry

1. Find the right bucket using the key's hash.
2. Lock that specific bucket.
3. Remove the entry.
4. Release the lock.

Indirect Question:

1. Why ConcurrentHashMap better than HashMap + synchronized?

Because it doesn't lock the entire map. so threads don't block each other . That means better speed and performance.

2. How to make a List as Read Only List?

An ArrayList can be made read-only easily with the help of **Collections.unmodifiableList()** method. This method takes the modifiable ArrayList as a parameter and returns the read-only unmodifiable view of this ArrayList

75 Iterator vs List Iterator?

Both Iterator and ListIterator are used to traverse collections in Java.

Supported Collections:

- Iterator can be used with any collection like List, Set, Queue, or Map.
- ListIterator works only with List collections like ArrayList, LinkedList

Direction:

- Iterator allows only forward traversal of a collection.
- ListIterator supports both forward and backward traversal.

Real-World Example:

Imagine scrolling through a photo gallery.

- With Iterator, we can only hit “Next”.
- With ListIterator, we get both “Next” and “Previous”.

```
//Iterator example - one direction only
List<String> names = new ArrayList<>();
names.add("Dhoni");names.add("Virat");
Iterator<String> iterator = names.iterator();

while(iterator.hasNext()) {
    System.out.println(iterator.next());
}

//ListIterator example - both directions
ListIterator<String> listIterator = names.listIterator();

while(listIterator.hasNext()) {
    System.out.println("Forward: " + listIterator.next());
}

while(listIterator.hasPrevious()) {
    System.out.println("Backward: " + listIterator.previous());
}
```

Modification:

- Iterator can remove elements during iteration.
- ListIterator can add, remove, or modify elements during iteration.

Starting Point:

- Iterator always starts at the beginning of the collection.
- ListIterator can start at any specific index of a list.

Features:

- **Iterator:** Simpler, with basic methods like `hasNext()`, `next()`, and `remove()`.
- **ListIterator:** Provides extra methods like `hasPrevious()`, `previous()`, `add()`, and `set()`.

When to Use:

Iterator: Use for simple traversal of any collection.

ListIterator: Use for advanced list traversal when we need both directions or to modify the list.

Indirect Questions:**1. What is the best way to traverse a Set, and why doesn't ListIterator work for it?**

Use Iterator because Set doesn't maintain index order, and ListIterator needs positional access which only Lists provide.

2. How do you safely remove elements while looping through a collection?

Use the Iterator's `remove()` method right after `next()` to avoid `ConcurrentModificationException`.

3. What would you use to insert an element while iterating?

Use `ListIterator.add()` which inserts right after the current position in a List.

4. How do you iterate backward through a list?

Use ListIterator and loop with `hasPrevious()` and `previous()` methods.

5. What happens if you try to modify a collection during iteration without using the right tool?

We'll get a `ConcurrentModificationException` because the collection detects structural changes mid-iteration.

6. What collection types restrict you from using a ListIterator, and why?

Set, Queue, and Map don't support ListIterator because they don't guarantee indexed order.

7. When should you use Iterator over ListIterator in real-world Java applications?

Use Iterator when working with Sets or non-indexed collections; it is simple and universal.

76 Fail-Fast vs Fail-Safe?

Definition:

Fail-Fast throws a `ConcurrentModificationException` if a collection is modified during iteration (other than via the iterator).

Fail-Safe does not throw exceptions during iteration, as it works on a copy of the collection or uses special mechanisms to handle changes.

Examples:

Fail-Fast: `ArrayList`, `HashMap`, `HashSet`, `LinkedList` (use their standard iterators).

Fail-Safe: `CopyOnWriteArrayList`, `ConcurrentHashMap`.

How They Work:

Fail-Fast detects structural modifications like adding or removing elements using a **modCount** variable, which is checked during iteration.

Fail-Safe operates on a copy of the original collection, so modifications do not affect the iterator.

Performance:

Fail-Fast is Faster, as it directly iterates over the original collection.

Fail-Safe is Slower, as it creates a separate copy or snapshot of the collection.

Use Cases:

Use **Fail-Fast** when thread safety is not required, and we want errors to be caught immediately (example, single-threaded applications).

Use **Fail-Safe** in multi-threaded environments where collections may be modified concurrently.

77 How FAIL-FAST iterator works? why it throws concurrent modification exception?

Fail-fast iterators directly **work the current state of the collection**.

Fail-fast iterators check if the collection has been modified while iterating. If it detects a change, it immediately throws a `ConcurrentModificationException`.

How it works:

1. The collection keeps a `modCount` (modification count) that increases with every change (`add`, `remove`, etc.).
2. When an iterator is created, it saves the current `modCount` as `expectedModCount`.
3. Before each operation , the iterator checks if `modCount == expectedModCount`.
4. If they don't match, it means the collection was modified, and the iterator throws an exception.

Example:

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");

for (String s : list) {
    list.add("C"); // Throws ConcurrentModificationException
    System.out.println(s);
}
```

- o when the iterator is created , `modCount = 2`.
- o Inside the loop, `list.add("C")` changes `modCount` to 3.
- o The iterator checks and sees `modCount != expectedModCount`
- o `ConcurrentModificationException`

Indirect Questions:

1. Which types of iterators consume more memory; fail-fast or fail-safe?

Fail-safe iterators consume more memory. They work on a copy of the collection, so any changes to the original don't affect iteration. Fail-fast iterators, on the other hand, directly iterate the collection and throw `ConcurrentModificationException` on modification, so they use less memory.

78 How FAIL-SAFE iterator works? why it does NOT throw concurrent modification exception?

Fail-safe iterators don't throw `ConcurrentModificationException` because they don't operate on the actual collection. Instead, they work on a copy of the collection's data, taken at the moment the iterator was created.

How it works:

- When we call `.iterator()` on a fail-safe collection (like `CopyOnWriteArrayList` or `ConcurrentHashMap`), it creates a snapshot (copy) of the data.
- The iterator reads from this snapshot, not the live collection.
- That is why modifications to the original collection during iteration don't throw `ConcurrentModificationException`.

```
CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
list.add("A");
list.add("B");

for (String s : list) {
    list.add("C"); // No ConcurrentModificationException
    System.out.println(s); // Prints only A, B
}
```

Here, `list.add("C")` modifies the original list, but the iterator only prints "A" and "B", since it works on copy.

79 Comparable and comparator, explain their differences?

In Java, when we want to **sort a list of objects**, Java needs to know **how to compare** those objects.

Primitive types like `int`, `double`, Java knows how to compare.

But if we have our own class, like `Student`, Java doesn't know if we want to sort by `marks`, `name`, or `age`.

To solve this, Java gave us two interfaces:

- `Comparable`
- `Comparator`

Comparable is an interface in Java that enables **comparing an object with other objects of the same type**. Comparable interface is provided by the `java.lang` package. Several built-in classes in Java like `Integer`, `Double`, `String`, etc., implement the Comparable interface.

Comparable is used for sorting objects by **natural or default ordering**

```
public class Student implements Comparable<Student> {
    int marks;

    public int compareTo(Student s) {
        return this.marks - s.marks; // ascending order
    }
}
```

The **Comparator** interface is used to define multiple ways of sorting objects. It is found in the `java.util` package and has two primary methods: `compare()` and `equals()`, although `equals()` is rarely overridden.

```
class NameComparator implements Comparator<Student> {
    public int compare(Student a, Student b) {
        return a.name.compareTo(b.name);
    }
}
```

Interview Tip:

Use Comparable for **default sorting**. Use Comparator when we need **custom sorting**. Also mention, Java uses them in sorting and ordering, like in `Collections.sort()`, `TreeMap`, or `TreeSet`. Without them, Java won't know how to compare objects. `TreeMap` in Java relies on either the Comparable interface or a Comparator

Reference: [Difference between Comparable and Comparator in Java - Scaler Topics](#)

Comparable: [JavaSamples/ComparableExample.java at main · CodingLyf-Fullstack/JavaSamples](#)

Comparator: [JavaSamples/ComparatorExample.java at main · CodingLyf-Fullstack/JavaSamples](#)

Indirect Question:

1. If a class defines its own natural ordering using Comparable, but I also provide a custom Comparator while sorting, which ordering will be applied?

If a class implements Comparable, that defines its natural ordering (say by ID or name). But if we pass a Comparator explicitly to a sort method (`Collections.sort(list, comparator)` or `stream.sorted(comparator)`), the Comparator takes precedence.

So, natural ordering is used only when no Comparator is given.

80 List.of() vs Collections.unmodifiableList?

List.of() (Java 9+)

- Creates a **truly immutable** list (cannot be modified in any way).
- If we try to modify it (add, remove, set), it throws an `UnsupportedOperationException`.

- **Nulls are not allowed** (throws NullPointerException if we try to add null).
- More memory-efficient (optimized for small lists).

```
List<String> immutableList = List.of("A", "B", "C");
immutableList.add("D"); // Throws UnsupportedOperationException
```

Collections.unmodifiableList()

- Wraps an existing list and makes it **unmodifiable** (but the original list can still change).
- If the original list changes, the unmodifiable view also reflects those changes.
- **Allows** null (if the original list had null).

```
List<String> originalList = new ArrayList<>(List.of("A", "B", "C"));
List<String> unmodifiableList = Collections.unmodifiableList(originalList);

unmodifiableList.add("D"); // Throws UnsupportedOperationException
originalList.add("D"); // This works! Now unmodifiableList = ["A", "B", "C", "D"]
```

81 What do you know about generics in Java?

When Java first introduced collections, they were powerful but also dangerous. we could put anything inside a List or Map, and the compiler wouldn't complain.

```
List students = new ArrayList();
students.add("Dhoni");
students.add("Kholi");
students.add(123); //Inserting number here
```

But when we tried to take things out and cast them to the right type, the program would crash at runtime with a ClassCastException.

```
String name = (String) students.get(2);
// Runtime error: ClassCastException
```

Generics were introduced in Java 5 to solve this exact problem. Generics in Java provide a way to write classes, interfaces, and methods that operate on objects of various types while maintaining type safety.

How Generics Work

Type Parameters

Generics introduce placeholders for types, usually written as single capital letters like T for Type, E for Element, K for Key, V for Value. These placeholders sit inside angle brackets (<>).

Generic Classes

We can define a class that works with any type. For example, a simple Box class:

```
class Box<T> {
    private T value;
```

```

void set(T value) {
    this.value = value;
}

T get() {
    return value;
}
}

```

Now, when we create a Box, we specify what type it holds:

```

Box<String> stringBox = new Box<>();
stringBox.set("Hello Generics");
System.out.println(stringBox.get()); // Hello Generics

Box<Integer> intBox = new Box<>();
intBox.set(42);
System.out.println(intBox.get()); // 42

```

Generic Methods

We can also make methods generic. The type parameter goes before the return type:

```

public <T> T printAndReturn(T data) {
    System.out.println(data);
    return data;
}

```

Now the method works with any type:

```

printAndReturn("Java"); // prints Java
printAndReturn(123); // prints 123

```

Bounded Type Parameters:

We can restrict the types that can be used for a type parameter using the extends keyword.

```

public <T extends Number> void process(T num) {
    System.out.println(num.doubleValue());
}

```

Now process() will only accept Number or its subclasses (Integer, Double, etc.).

Wildcards

What if we don't know the exact type but still want flexibility? That is where wildcards (?) come in

```
List<? extends Number> numbers;
```

This means the list can hold Integer, Double, or any subclass of Number.

Important Thing to Know

- **Primitives don't work:** We can't use int or char directly with generics. Instead, we use wrapper classes like Integer and Character.

82 What is Garbage Collection?

Garbage collection in Java means Java automatically removes objects from memory when they're no longer needed.

If the program doesn't use an object anymore and nothing refers to it, Java's Garbage Collector will clean it up to free memory. We don't have to delete it manually. Java takes care of it in the background.

How It Works:

- **Identifies Garbage:** The GC scans memory and marks objects that are unreachable (no longer referenced by any part of the program).
- **Removes Garbage:** It deletes these unused objects, reclaiming memory.
- **Compacts Memory:** Some GC algorithms also rearrange remaining objects to reduce fragmentation.

83 What are the different types of Garbage Collectors?

1. Serial Garbage Collector

This is the most basic type of GC.

It uses a single thread for garbage collection

2. Parallel Garbage Collector (also called the Throughput Collector)

This GC uses multiple threads to perform garbage collection.

It focuses on high overall performance (high throughput), even if that means longer pause times.

3. Concurrent Mark-Sweep (CMS) Collector

This GC tries to reduce pause times by doing most of its work concurrently with the application.

It is designed for applications that need quick responses, like web servers or GUI applications.

Important Note:

CMS is deprecated starting from Java 9 and removed in Java 14. It was popular for low-latency applications before G1 GC became the default.

4. G1 Garbage Collector (Garbage First)

G1 is the default collector in Java from version 9 onwards.

It divides the heap into small regions and prioritizes collecting regions with the most garbage first.

5. ZGC (Z Garbage Collector)

These are modern, advanced GCs designed for very large heap sizes (often 10 GB or more).

They are capable of ultra-low pause times (often under 10 milliseconds), even as memory usage grows.

84 How Garbage Collections works internally?

Java Garbage Collectors implement a *generational garbage collection strategy* that categorizes objects by age

Step by Step Process from Object creation to object deletion from memory

1. Heap: When we create an object in Java using new keyword, it goes to the heap. The heap memory in the JVM is divided into generations.

Young Generation and Old Generation.

2. Young Generation again divided into 3 section : Eden Space, FromSpace (S0) and ToSpace (S1)

Eden space - all new objects start here, and initial memory is allocated to them

Survivor spaces (FromSpace and ToSpace) - objects are moved here from Eden after surviving one garbage collection cycle. When objects are garbage collected from the Young Generation, it is a minor garbage collection(Minor GC) event.

3. Old Generation: Objects that are long-lived are eventually moved from the Young Generation to the Old Generation. This is also known as Tenured Generation, and contains objects that have remained in the survivor spaces for a long time. Here Major GC event occurs to clear the objects. Major GC uses Marks and Sweep Algorithm.

Mark: The garbage collector "marks" all reachable objects, starting from root nodes.

Sweep: After marking, its "sweeps" through memory to delete objects that were not marked as reachable, freeing up space.

Reference: [Garbage Collection in Java – What is GC and How it Works in the JVM](#)

<https://medium.com/@rakeshrdy8/garbage-collection-in-java-explained-bbebdc5f75ac>

[9. Java Memory Management and Garbage Collection in Depth](#) (Video)

Indirect Questions:

1. Where are the objects created, in heap or stack?

Objects are always created in the heap. Stack holds method calls and references, not the actual objects.

2. Which part of the memory is involved in Garbage Collection, Stack or Heap?

Garbage Collection only works on the heap. The stack is managed separately by the JVM.

3. Who manages the Garbage Collector?

The JVM handles Garbage Collection internally. Developers don't control it directly.

4. How to request Garbage Collection explicitly?

We can call `System.gc()` or `Runtime.getRuntime().gc()`, but JVM may ignore the request.

5. When does an object become eligible for Garbage Collection?

When there are no more references pointing to it. Basically, it becomes unreachable.

6. Different ways to make an object eligible for Garbage Collection?

Set its reference to null, reassign it, or let it go out of scope.

1. Setting reference to null

```
String s = new String("Hello");
s = null; // Now "Hello" is eligible for GC
```

2. Reassigning the reference

```
String s = new String("Java");
s = new String("Python"); // "Java" becomes unreachable
```

7. What is generation strategy?

It is a JVM optimization that divides the heap into young, old, and sometimes permanent spaces. Most objects die young, so it collects young generation more often.

8. What is HotSpot?

HotSpot is the JVM implementation from Oracle. It uses Just-In-Time (JIT) compilation and smart Garbage Collection strategies for better performance.

9. How does Metaspace is different from PermGen?

PremGen is part of Heap memory, It contains in meta data of the class. It is fixed in size. From Java 8 Metaspace is introduced which is not fixed in size.

85 Difference between WeakReference and SoftReference?

WeakReference: The garbage collector clears the object as soon as there are no strong references to it, even if there's plenty of memory available. It is like telling the GC, "If nobody else is holding it, just clear it from memory."

SoftReference: The garbage collector keeps the object around as long as there's enough memory. It is only cleared when memory starts running low. Think of it as saying, "Keep it if you can, but drop it when space gets tight."

Code link: [JavaSamples/ReferenceDemo.java at main · CodingLyf-Fullstack/JavaSamples](#)

Example: **WeakHashMap** uses weak references for its keys, so if a key is no longer strongly referenced elsewhere, the entry will be removed automatically during GC.

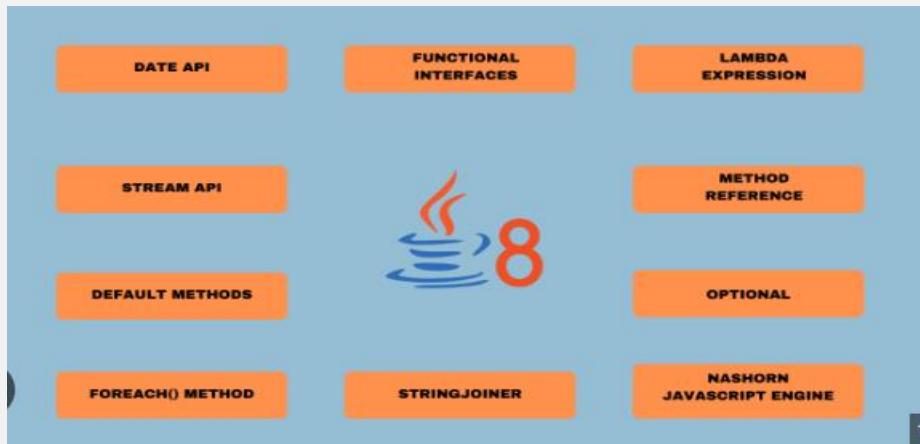
```
public class WeakHashMapExample {  
    public static void main(String[] args) {  
        Map<Object, String> map = new WeakHashMap<>();  
  
        Object key1 = new Object();  
        Object key2 = new Object();  
  
        map.put(key1, "Value for key1");  
        map.put(key2, "Value for key2");  
  
        System.out.println("Before GC: " + map);  
  
        // Remove strong reference to key1  
        key1 = null;  
  
        System.gc(); // Suggest GC  
        System.runFinalization();  
  
        System.out.println("After GC: " + map);  
    }  
}  
Output:  
Before GC: {java.lang.Object@372f7a8d=Value for key1, java.lang.Object@2f92e0f4=Value for key2}  
After GC: {java.lang.Object@2f92e0f4=Value for key2}
```

What happened here

- **key1** had NO strong reference after `key1 = null`, so during GC the weak reference in `WeakHashMap` was cleared, and its entry disappeared automatically.
- **key2** was still strongly referenced, so its entry stayed.

86 Can you list out the Java 8 features?

Below image contains Java 8 features. These features are very important to prepare for the interviews. We will go through one by one.



Interview Tip: No need to mention all the features Start with 3–4 important ones, explain how they help in real projects. The major ones are lambdas, streams, functional interfaces, and default methods. We used lambdas and streams heavily when working on data transformations and in business logics.

87 What are default methods and why are they introduced?

A default method allows to add a new method to an interface without breaking the classes that already implement it.

Before Java 8, interfaces can only have abstract methods. So, if we add a new method to an interface, every class that implemented it has to override that method, or we will get a compile-time error. That was a huge headache, especially for large codebases or libraries where interfaces were implemented in hundreds of places.

To fix this, Java introduced default methods. Now, when we add a new method to an interface, we can also provide a default implementation. In this way, existing classes won't break, they can either use the default or choose to override it.

The main goal here is backward compatibility.

Example:

The stream() method was added as a **default method** in the java.lang.Iterable interface in Java 8:

```
default Stream<E> stream() {
    return StreamSupport.stream(splitter(), false);
}
```

Now any class that implements Iterable, like List, Set, or Queue, automatically got the ability to use streams:

```
List<String> names = Arrays.asList("Coding", "Lyf");
names.stream()
    .filter(name -> name.length() > 4)
```

```
.forEach(System.out::println);
```

We don't have to go back and modify ArrayList, LinkedList, or any collection class.

Indirect Questions:

1. Is it necessary to override default methods of interface in Java 8?

No, they contain built in implementation. We override only if we want custom behaviour.

2. Is the default keyword one of the access modifiers?

No, it is not. It is a special keyword for method implementation in interfaces, not for setting access levels like public or private.

3. How to override default methods?

We can override like a like any normal method. override it in the class using @Override, and provide own body.

4. If two interfaces have the same default method, and you implement both, what will happen?

If a class implements two interfaces that each have the same default method, a compilation error will occur because the compiler cannot determine which default method to inherit.

To resolve this, the class must explicitly override the default method and provide its own implementation, or explicitly call the desired interface's default method using InterfaceName.super.methodName().

5. Can you call a default method from a class that implements Interface?

Yes, we can call a default method from a class that implements the interface, but only within that class or its subclasses, and only using InterfaceName.super.methodName().

Interview Tip: Try to tell with real world example like, Default methods were introduced so interfaces can be good choice without breaking existing implementations. For example, if an interface is used in 100 places, adding a new method would normally break them. With default, we can add it with a basic body and avoid rewriting everywhere.

88 Why Java 8 introduced static methods to the Interfaces?

Before Java 8, utility methods related to interfaces were in separate classes. Think about how we always used Collections.sort() or Math.max() , those were just static helpers sitting outside the actual interface logic.

So, in Java 8, They allowed interfaces to group their own utility methods?

That is how static **methods in interfaces** came to picture. They belong to the interface, not to the implementing class. So, we can call them like InterfaceName.methodName(), not through an object.

JDK Example:

In Java 8, the Comparator interface added static methods like:

```
public static <T extends Comparable<? super T>> Comparator<T> naturalOrder() {
    return (a, b) -> a.compareTo(b);
}
```

Now we can use like:

```
List<Integer> values = Arrays.asList(212, 324, 435, 566, 133, 100, 121);
// naturalOrder is a static method
values.sort(Comparator.naturalOrder());
```

Indirect Question:

1. Can an interface in Java have utility methods?

Yes, starting from Java 8. We can now add static methods to interfaces to hold related utilities.

2. What's the difference between a static method in a class and a static method in an interface?

A static method in a class can be called from its child class if the child doesn't define the same method. But a static method in an interface can only be called using the interface name. it doesn't get passed to the class that implements it.

3. Why do we still have utility classes like Collections or Arrays if interfaces can now have static methods?

Because they were created before Java 8, and changing them now would break backward compatibility across millions of projects.

4. If a class implements an interface, can it access the static method inside the interface via object reference?

Nope. Static methods in interfaces must be called using the interface name not via the object.

5. Can we override the static method of an interface?

No. Static methods in interfaces can't be overridden, they belong only to the interface, not to the implementing class.

89 What is optional and it uses?

Let's say we are calling a method and we are not sure if it'll return a value or null. Before Java 8, We had to check whether the value is null or not if not, there is high chance of getting **NullPointerException**.

Optional is a container that may or may not hold a non-null value. It is used to clearly show that a method might not return a result. Instead of returning null, the method returns an Optional to help avoid NullPointerException and make the code safer and easier to understand.

Example: In the below example, if user is null, user.getName() will throw NullPointerException.

```
String getName(User user) {
    return user.getName();
}
```

Now from Java 8, instead of returning String we will return **Optional<String>** which tells that , this may contain null and should be handled.

```
Optional<String> getName(User user) {
```

```

    return Optional.ofNullable(user.getName());
}

String name = getName(user).orElse("Anonymous"); //If user is null it returns 'No User'

```

Differences of each method in optional:

Optional.of() vs Optional.ofNullable():

Optional.of(value): Use it when the value is guaranteed to be non-null. If value is null, it throws a NullPointerException.

Optional.ofNullable(value): Use it when the value might be null. If value is non-null, it returns an Optional containing the value; if value is null, it returns an empty Optional (Optional.empty()).

Optional.empty() vs null:

Optional.empty(): Represents the explicit absence of a value within an Optional container. It is an object that clearly signals "no value present."

null: Represents the absence of a reference to an object. It can lead to NullPointerExceptions if not handled carefully and doesn't explicitly convey the intent of "no value." Optional aims to reduce the need for null checks.

Optional.get() vs Optional.orElse() vs orElseGet():

Optional.get():

Returns the contained value if present; otherwise, it throws a NoSuchElementException.

This method should only be used when isPresent() has confirmed the presence of a value.

Optional.orElse(defaultValue)

Returns the value if present. If not, returns the defaultValue we give.

But defaultValue is always be executed, even if it is not needed.

Example:

```
String name = optionalName.orElse(createDefaultName());
```

Here, createDefaultName() will always be executed even optionalName is not null.

Optional.orElseGet(supplier)

Returns the value if present. If not, it *calls* the supplier to get the default.

The supplier runs only if needed, so it is better for heavy or costly operations.

Example:

```
String name = optionalName.orElseGet(() -> createDefaultName());
```

Optional.isPresent() + get() vs ifPresent():

Optional.isPresent() + get()

We first check if the value is present using isPresent(), and then use get() to retrieve it.

```
if (optionalName.isPresent()) {
    System.out.println(optionalName.get());
}
```

This works, but it is a bit long and not the cleanest way to write code.

Optional.ifPresent()

This is a shorter and cleaner way. If a value is present, it runs the code. No need to check manually.

```
optionalName.ifPresent(name -> System.out.println(name));
```

This avoids NullPointerException and looks neater.

Interview Tip: Mention a real-world scenario like, we can use Optional for API responses, if a user was not found, we returned Optional.empty() instead of null. That avoided null pointer bugs and we can use orElse to return default value if no user found.

90 What is functional interface and why they are introduced in Java?

First let's see what is functional programming

Java was always an object-oriented language. But as apps grew bigger and data processing became more complex, writing clean and reusable code became harder. Developers wanted a way to **focus more on "what to do" than "how to do it"**, especially when working with **collections, filtering, mapping, reducing, etc.**

That is where Functional Programming came into picture. Java adopted it in Java 8 with features like:

- Lambda expressions - short way to write functions
- Functional interfaces - interfaces with just one abstract method
- Streams API - to process collections in a chain-style

These made code shorter, cleaner.

Functional Programming:

Functional programming is a way of writing code where we build programs using functions, not objects or changing variables. It focuses on “what to do, not “how to do” it.

Example:

Let's say we have a list of names and want only names that start with "S".

Before Java 8:

```
List<String> names = Arrays.asList("Sam", "Ravi", "Sneha", "John");
List<String> result = new ArrayList<>();
```

```

for (String name : names) {
    if (name.startsWith("S")) {
        result.add(name);
    }
}

System.out.println(result); // Output: [Sam, Sneha]

```

Here we wrote:

- A loop
- An if condition
- Manually added items to a new list

From Java 8 (Functional Way):

```

List<String> names = Arrays.asList("Sam", "Ravi", "Sneha", "John");

List<String> result = names.stream()
    .filter(name -> name.startsWith("S"))
    .collect(Collectors.toList());

System.out.println(result); // Output: [Sam, Sneha]

```

Here: We don't need write loops, no manual adding.

Reference: [Functional Programming in Java with Examples - GeeksforGeeks](#)

Functional interface: A functional interface is an interface that contains exactly one abstract method, although it can also contain multiple default or static methods. The presence of a single abstract method allows instances of functional interfaces to be created with lambda expressions.

```

@FunctionalInterface
public interface MyFunctionalInterface {
    void execute();
}

```

This interface has only one abstract method, execute, making it a functional interface.

Functional interfaces are used extensively with lambda expressions. A lambda expression provides a clear and concise way to implement the single abstract method of a functional interface. Here is how we can implement the MyFunctionalInterface using a lambda expression

```

MyFunctionalInterface myFunc = () -> System.out.println("Executing...");
```

myFunc.execute();

Reference: [Functional Interfaces in Java - Scaler Topics](#)

Tricky Questions:

1. Can we include default and static methods in functional Interface?

Yes, A functional interface can have exactly **one abstract method** and any number of default and static methods.

2. Can a functional interface extend another functional interface?

Yes, but with a condition

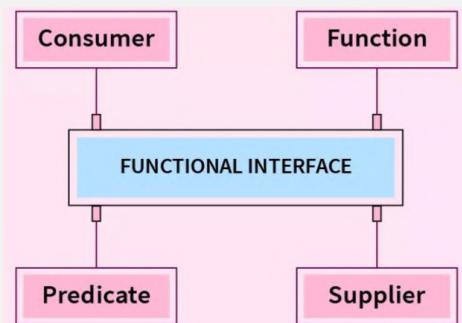
- If the parent functional interface has one abstract method, and the child does not add another abstract method, then the child is still functional.
- If the child adds another abstract method (other than what it inherits), it is no longer a functional interface.

3. Can a functional interface implement another interface?

An interface can extend another interface, but it cannot implement one. Implementation is only for classes.

91 Explain different types of functional interfaces?

Java SE 8 included 4 main kinds of functional interface.



Consumer: This interface represents an operation that accepts a single input argument and returns no result. It is used when an action needs to be performed on an object without producing a return value. Example: `forEach()`

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

Supplier: This interface represents a supplier of results. It takes no arguments and returns a value of a specified type. It is used when a value needs to be generated or retrieved. Example: `Optional.orElseGet`

```
@FunctionalInterface
public interface Supplier<T> {
```

```
T get();
}
```

Predicate: A Predicate is a functional interface that represents an operation that takes an input of type `T` and returns a boolean result. Example: filter()

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Function: This interface represents a function that accepts one argument and produces a result. It is used for transforming an input value into an output value of a potentially different type. Example: map()

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Indirect Questions:

1. What interface would you use to check if an item matches a criteria?

Use Predicate<T>. It returns a boolean and is perfect for filtering data in streams.

```
Predicate<Integer> isEven = num -> num % 2 == 0;
```

2. What interface allows converting one data type into another in streams?

Function<T, R> : it maps one type to another, like converting String to Integer.

```
Function<String, Integer> toInteger = str -> Integer.parseInt(str);
```

3. How do you perform logging or save to DB while processing a stream?

Use Consumer<T>. It handles side effects like printing, logging, or saving records.

```
Consumer<String> logger = msg -> System.out.println("Logging: " + msg);
```

4. How can you generate random numbers or timestamps in a stream?

Use Supplier<T>. It takes no input and returns something like new Date() or UUID.

```
Supplier<Double> randomGenerator = () -> Math.random();
```

5. How do you lazily provide a default value in case of null using Optional?

Use Optional.orElseGet(Supplier). It'll only call the Supplier if the value is missing.

```
Optional<String> name = Optional.ofNullable(null);
```

```
String result = name.orElseGet(() -> "Default Name");
```

6. How do you think the Supplier interface can be useful when writing lazy initialization code?

Supplier can delay object creation until it is actually needed, it is great for saving memory or resources.

7. Which functional interfaces existed in Java before Java 8?

Runnable and Callable were considered functional interfaces before Java 8 as they had a single abstract method.

92 Explain about Lambda expression?

Before Java 8, if we want to do something simple, filter a list, or run some code in a new thread we had to write a lot of code using anonymous inner classes.

Java 8 changed this by introducing the idea that **functions can be first-class citizens**. That means functions can be treated just like variables, we can pass them around, store them, and return them from methods.

Before Java 8 (Anonymous Inner Class): if we want to create a thread that prints "Hello".

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello");
    }
});
```

After Java 8 (Lambda Expression): Same behaviour, written in a much cleaner way:

```
Thread thread = new Thread(() -> System.out.println("Hello"));
thread.start();
```

Now we are passing behaviour (a function) like data, that is what we mean by functions becoming **first-class citizens**.

Code Link: [JavaSamples/LambdaExample.java at main · CodingLyf-Fullstack/JavaSamples](#)

93 Difference between lambda expression and anonymous class?

Anonymous Classes An anonymous class is like a temporary, nameless class that we create and instantiate on the spot. We can use it to implement an interface or extend an abstract class.

Before Java 8, if we wanted to create a new Runnable to run in a thread, we had to write it as an anonymous class:

```
Runnable r = new Runnable() {
```

```

@Override
public void run() {
    System.out.println("Running a task!");
}
};
```

Lambda Expressions Introduced in Java 8, a lambda expression provides a much more concise way to represent an implementation of an interface with a single abstract method (known as a functional interface).

Using a lambda expression, the same Runnable task can be written in a single line:

```
Runnable r = () -> System.out.println("Running a task!");
```

Anonymous Inner Class

It is a class without name.

It can extend abstract and concrete class.

It can implement an interface that contains any number of abstract methods.

Inside this we can declare instance variables.

Anonymous inner class can be instantiated.

Inside Anonymous inner class, "this" always refers to current anonymous inner class object but not to outer object.

It is the best choice if we want to handle multiple methods.

At the time of compilation, a separate .class file will be generated.

Memory allocation is on demand, whenever we are creating an object.

Lambda Expression

It is a method without name (anonymous function).

It can't extend abstract and concrete class.

It can implement an interface which contains a single abstract method.

It does not allow declaration of instance variables; variables declared act as local variables.

Lambda Expression can't be instantiated.

Inside Lambda Expression, "this" always refers to current outer class object (enclosing class).

It is the best choice if we want to handle interface.

At the time of compilation, no separate .class file will be generated; it is converted into a private method of the outer class.

It resides in a permanent memory of JVM.

94 Explain about Method references and uses?

Method references are like shortcuts. Instead of writing a full lambda expression, we can just point to an existing method.

Lambda style: `list.forEach(s -> System.out.println(s));`

It works fine. What we are doing is, taking "s" and passing it to `System.out.println`.

Now with method reference: `list.forEach(System.out::println);`

Types of Method References:

Static Method Reference: Refers to a static method of a class.

```
// Without method reference
Arrays.sort(numbers, (a, b) -> Integer.compare(a, b));

// With method reference - much cleaner!
Arrays.sort(numbers, Integer::compare);
```

Instance Method Reference of a Particular Object: Refers to an instance method of a specific object.

```
class Printer {
    public void print(String message) {
        System.out.println(message);
    }
}

Printer myPrinter = new Printer();

// Without method reference
messages.forEach(msg -> myPrinter.print(msg));

// With method reference
messages.forEach(myPrinter::print);
```

Constructor Reference: Refers to a constructor of a class

```
// Without method reference
Supplier<List<String>> supplier = () -> new ArrayList<>();

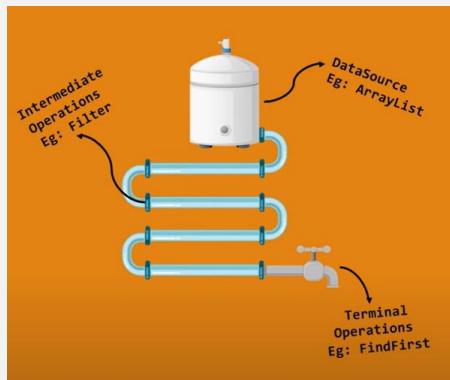
// With method reference
Supplier<List<String>> supplier = ArrayList::new;
```

95 Explain about Streams?

Before Java 8, we used to write long for-loops to filter, map, or collect data.

Streams are introduced in Java 8. Streams make the code shorter, readable, and more declarative; they provide a functional and efficient way to process collections by creating a pipeline of operations. They enable declarative programming, lazy evaluation, and parallel processing without modifying the original data source.

Think of Java Streams like water flowing through pipes.



Lazy Evaluation -> Water doesn't flow until the tap is opened. Similarly, Stream operations don't execute until a terminal operation like collect or forEach is called.

Parallel Processing -> Just like multiple pipes can distribute water to different locations simultaneously, parallel streams split data into chunks and process them across multiple threads for better performance.

Immutable Flow -> Water in the pipeline doesn't affect the water tank. Similarly, a Java Stream doesn't modify the original collection; instead, it produces a transformed result.

It provides various operators to process the data. They can be segregated like

Source Operators

This is where the stream starts. We can turn a list, array, or even lines from a file into a stream.

Example: `List.of(1,2,3).stream()` or `Files.lines(path)`

Intermediate Operators

These are the steps in the middle. They transform the stream but don't run until a terminal operation is called (this is called lazy evolution).

Example: `.filter(n -> n % 2 == 0).map(n -> n * 2)`

Short-Circuit Operators

These stop the stream early when a condition is met. Useful when we don't need to process the entire stream.

Example: `.limit(5), .anyMatch(x -> x > 100)`

Terminal Operators

These are the end of the stream. Once we call a terminal operation, everything runs and gives us a result like a list, count, sum, etc.

Example: `.collect(toList()), .count(), .forEach(System.out::println)`

Indirect Questions:

1. What is lazy evaluation in streams?

Streams don't do anything until a terminal operation is called. That means nothing runs during filter() or map() until we finally say something like collect() or forEach().

2. Common pitfalls to avoid with Streams?

Reusing the same stream twice doesn't work it throws an error.

3. How stream operations are optimized internally?

Java combines operations like filter() and map() into a single loop behind the scenes , so we don't pay for each one separately. It is called loop fusion.

4. What is loop fusion?

Loop fusion in Java Streams is an internal optimization technique where multiple intermediate stream operations are combined and executed in a single pass over the data source. Instead of processing each element through one operation, then passing the result to the next, and so on, loop fusion merges these operations into a single, optimized loop.

```
list.stream()
    .filter(x -> x % 2 == 0)
    .map(x -> x * x)
    .forEach(System.out::println);
```

This looks like multiple steps, but under the hood it is one loop.

How Fusion Happens

- Each intermediate op (filter, map) doesn't create a new list.
- Instead, it wraps its logic into a chain of operations.
- When the terminal op (forEach) pulls elements, the pipeline executes all stages in sequence for each element.

So for element x = 4:

- Check filter ($4 \% 2 == 0 > \text{true}$)
- Apply map ($4 * 4 > 16$)
- Send to forEach (print 16)

Only then does it move to the next element.

This is why it is called loop fusion: multiple logical loops (filter > map > forEach) are fused into a single physical loop.

5. Are Streams always faster than loops? Why or why not?

No, Streams are NOT always faster than loops.

Overhead: Streams introduce additional abstraction and object creation, which can make them slower than plain for-loops, especially for small or simple tasks.

Performance: For basic iteration (e.g., summing an array), a classic loop is often faster due to lower overhead.

Reference: [Is Java Stream Really Faster Than For-Loop? 🔎 | Java Interview Question](#)

96 Can you list of streams operators you have used or you know?

1. Source Operators: Create a stream from a collection, array, or I/O source.

- `stream()` – Converts a collection into a sequential stream.
- `parallelStream()` – Creates a parallel stream for multi-threading.
- `Stream.of()` – Generates a stream from given values.
- `Arrays.stream()` – Converts an array into a stream.
- `IntStream.range()` – Creates a stream of numbers within a range.
- `BufferedReader.lines()` – Converts lines of a file into a stream.

2. Intermediate Operators: Transform or filter data without executing immediately (lazy evaluation).

- `filter()` – Selects elements that match a given condition.
- `map()` – Transforms each element using a function.
- `flatMap()` – Flattens nested structures into a single stream.
- `distinct()` – Removes duplicate elements from the stream.
- `sorted()` – Sorts elements in natural or custom order.
- `limit(n)` – Restricts the stream to the first n elements.
- `skip(n)` – Skips the first n elements in the stream.
- `peek()` – Used for debugging without modifying elements.

3. Terminal Operators: Trigger execution and produce a final result, consuming the stream.

- `collect()` – Converts a stream into a List, Set, or Map.

- `forEach()` – Performs an action for each element.
- `count()` – Returns the number of elements in the stream.
- `findFirst()` – Retrieves the first element of the stream.
- `findAny()` – Retrieves any element from the stream.
- `reduce()` – Aggregates elements into a single result.
- `toArray()` – Converts stream elements into an array.

4. Short-Circuiting Operators: Stop execution early when a condition is met for better performance.

- `anyMatch()` – Returns true if any element matches a condition.
- `allMatch()` – Returns true if all elements match a condition.
- `noneMatch()` – Returns true if no elements match a condition.
- `limit(n)` – Stops processing after selecting n elements.
- `findFirst()` – Retrieves the first element and stops further operations.

Many indirect or Scenario based questions can be asked on these operators. Example

1. How do you debug the stream operational flow?

`peek()` is an intermediate operation that allows us to inspect elements as they flow through the stream pipeline . it is typically used for debugging, to observe the elements as they flow through the pipeline.

2. Can we access outside variable in streams?

In Java streams (or lambdas), we can access variables from outside the stream/lambda body if they are effectively final. That means the variable isn't reassigned after its initialization.

```
int factor = 2; // effectively final
List<Integer> numbers = List.of(1, 2, 3);

numbers.stream()
    .map(n -> n * factor) // accessing outside variable
    .forEach(System.out::println);
```

3. Can we change the outside variable in streams?

No, if we try to modify an outside variable inside the lambda/stream, we will get a compilation error.

```
int sum = 0;
List<Integer> numbers = List.of(1, 2, 3);

numbers.forEach(n -> sum += n); //Compilation Error
Error: Local variable sum defined in an enclosing scope must be final or effectively final
```

4. Is there any way to update outside variables in streams?

If we really need to update state from inside a stream, we can use Mutable containers like AtomicInteger or a list.

```
AtomicInteger sum = new AtomicInteger(0);
numbers.forEach(n -> sum.addAndGet(n));
System.out.println(sum.get());
```

5. If you don't use the terminal operation in Stream pipeline, will the intermediate operations be executed, why or why not?

- Streams in Java are lazy. Intermediate operations (map, filter, sorted, etc.) are only recorded, not run immediately.
- They get executed only when a terminal operation (forEach, collect, reduce, etc.) is called.
- Without a terminal operation, the pipeline never triggers, so nothing runs.

97 Difference between Streams and Collections?

Collections, like List, Set, and Map are used hold the data. It is like a container. We can add the items, retrieve the items and update them. They will be saved in memory.

Streams are different from collections. Streams don't store anything. They just describe how data should be processed. We don't add or remove things from a Stream. We just say, 'take this list, filter out the even numbers, map each number to its square, and collect the result.'

The important difference is, Streams are lazy. They don't do anything until we tell them to finish, we usually do with a terminal operation like .collect() or .forEach().

Let's say we have a list of 1000 numbers. With a Collection, we already have all those numbers in memory. With a Stream, nothing really happens until we start processing them. The data flows through like water in a pipe, step-by-step.

So, the core difference is, **Collections are about storing data. Streams are about processing data.**

98 Difference between map and flatMap?

The **map()** method transforms each element of a stream into another object. It takes a Function as an argument, which defines the transformation logic. This function is applied to each stream element, and the result is collected into a new stream.

The **flatMap()** method takes each element in the stream, applies a function to it, and that function returns **another stream** (could have 0, 1, or many elements). Then, flatMap() takes all those small streams and joins them into one big stream.

Reference: [Exploring the Differences: Java map vs. flatMap | by Reetesh Kumar | Medium](#)

Indirect Question:

1. List of lists and need to be converted it into a single list of all elements, how you do it?

Use flatMap

99 How reduce works?

reduce() is a terminal operation in Java Streams that takes a sequence of elements and combines them into a **single result** using a function , like adding, multiplying, or finding the max.

Example:

Lets say, We have a box of numbers:

[5, 10, 15]

We want to **add** them together. Normally, we say:

$5 + 10 = 15$, then $15 + 15 = 30$

That is exactly what reduce() does. It takes two elements at a time, applies an operation (like sum), and continue further.

```
List<Integer> numbers = Arrays.asList(5, 10, 15);

int sum = numbers.stream()
    .reduce(0, (a, b) -> a + b); // 0 is the starting value

System.out.println(sum); // Output: 30
```

100 forEach vs collect()?**collect()**

- It collects the processed stream into a data structure (like a List, Set, or Map)
- It returns that data structure
- We use it when we want to store or reuse the results

forEach()

- It goes through each item and performs an action (like printing or logging)
- It returns nothing (void)
- We use it for side effects, not for building data

Indirect Question:

1. What happens if you call terminal operation without any intermediate operations?

It works. The stream directly applies the terminal operation over the source data.

For example: count()

```
List<String> list = Arrays.asList("a", "b", "c");
```

```
long count = list.stream().count();
```

```
System.out.println(count); // Output: 3
```

101 How can you optimize a Stream pipeline for better performance?

- Apply filtering and short-circuiting operations (like filter, limit, findFirst) early in the pipeline to reduce the number of elements processed downstream.
- Use primitive specialized streams (IntStream, LongStream, DoubleStream) to avoid unnecessary boxing/unboxing.
- Use parallel streams only when working with large datasets and the workload is CPU-bound.

102 Difference between sequential stream and parallel stream?

Sequential Streams

By default, all streams in Java are sequential. This means the data is processed one item at a time, in order, using a single thread, usually the main thread or the one we are currently in.

Example:

```
List<String> names = List.of("Rohit", "Gill", "Hardik");  
  
names.stream()  
    .map(String::toUpperCase)  
    .forEach(System.out::println);
```

This will process "Rohit", then "Gill", then "Hardik", one after the other.

Parallel Streams

A parallel stream splits the data into chunks and processes them in parallel using multiple threads behind the scenes (from the common fork-join pool). The goal is better performance, especially with large datasets.

Just change .stream() to .parallelStream():

```
List<String> names = List.of("Rohit", "Gill", "Hardik");  
  
names.parallelStream ()  
    .map(String::toUpperCase)  
    .forEach(System.out::println);
```

Now "Rohit", "Gill", and "Hardik" might be processed *out of order* and *at the same time* by different threads.

When .parallel() is called, Spliterator **splits the work** into chunks that can be processed in parallel by ForkJoinPool.

Indirect Questions:

1. When to use Sequential Streams?

We will use sequential streams when the order of operations matters or when we have a small amount of data.

Ex: Read data in sequential manner and processing it.

2. When to use Parallel Streams?

Use parallel streams for large, CPU-intensive tasks where the order of elements doesn't matter. Example. If we have a million numbers and we need to calculate the square of each one; a parallel stream can split the work and do it much faster

3. When not to use Parallel Streams?

Avoid parallel streams when order is must, the operations are fast and the dataset is small, as the overhead of managing threads will hit the performance.

4. What is a Spliterator?

Spliterator stands for Split + Iterator. It is a special iterator introduced in Java 8 that supports:

1. Sequential traversal (like Iterator).
2. Efficient splitting of data into chunks for parallel processing.

It is the backbone of how the Stream API works under the hood.

103 How do you handle exceptions in streams?

Let's divide this into checked and unchecked exceptions.

Unchecked Exceptions:

We can catch these unchecked exceptions while processing the data. We can handle unchecked exceptions with two approaches

1. Handle It Inside
2. Filter Out the Bad Ones

Example: Suppose we have a list and we would like to convert strings to number. But it will throw NumberFormatException because of "four"

```
List<String> numbersAsString = Arrays.asList("1", "2", "3", "four", "5");
```

In "**Handle it Inside**" approach, we catch the error and return some fallback value. In the below we have handled the exception and return -1 as a fallback value.

```
List<String> numbersAsString = Arrays.asList("1", "2", "3", "four", "5");

numbersAsString.stream()
    .map(s -> {
        try {
            return Integer.parseInt(s); // throw NumberFormatException
        }
```

```

    } catch (NumberFormatException e) {
        return -1; // fallback value
    }
})
.forEach(System.out::println);

```

In “**Filter out the bad ones**” approach we will filter them using filter method. Here we are returning null when exception occurs and filtered using Objects:nonNull.

```

List<String> numbersAsString = Arrays.asList("1", "2", "3", "four", "5");

numbersAsString.stream()
    .map(s -> {
        try {
            return Integer.parseInt(s);
        } catch (NumberFormatException e) {
            // We caught the exception! Let's return a default value, like -1.
            System.err.println("Could not parse: " + s);
            return null;
        }
    })
    .filter(Objects::nonNull)
    .forEach(System.out::println);

```

We can also use Wrapper to handle exceptions to write more cleaner code. We will see that in next section.

Code Link: [JavaSamples/LambdaUnCheckedException.java at main · CodingLyf-Fullstack/JavaSamples](#)

Checked Exceptions:

Java streams don't work well with checked exceptions because the functional interfaces used in stream operations (like Function, Predicate, etc.) don't declare any checked exceptions. Let's see three approaches

1. Wrap in RuntimeException

The simplest approach is to catch the checked exception and wrap it in a runtime exception. In the below we are catching Checked Exception and throwing Runtime Exception.

```

list.stream()
    .map(item -> {
        try {
            return someMethodThatThrowsCheckedException(item);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    })
    .forEach(System.out::println);

```

2. Create a Wrapper Function

I would suggest to watch the video and code link to understand better.

Video link: [Java 8 Streams | Exception Handling Mechanism | lambda | JavaTechie](#)

Code: [JavaSamples/LambdaCheckedException.java at main · CodingLyf-Fullstack/JavaSamples](#)

3. Propagate Exception

If we need to propagate the checked exception, we need to collect results and handle exceptions after processing:

```
List<String> results = new ArrayList<>();
List<Exception> exceptions = new ArrayList<>();

list.forEach(item -> {
    try {
        results.add(someMethodThatThrowsCheckedException(item));
    } catch (CheckedException e) {
        exceptions.add(e);
    }
});

if (!exceptions.isEmpty()) {
    // Handle exceptions
}
```

104 What are disadvantages or when not to Use Java Streams?

Java Streams are great for writing clean and short code when working with collections. But they're not always the best tool. Sometimes, using a regular for or while loop is easier, faster, or just makes more sense. Here is when we should avoid using streams:

1. When we Need to Change the Original List

Streams don't change the original data. They work on a copy and return a new result. If we need to add, remove, or update items directly in the original list, use a loop. It is clearer and usually faster.

2. When we Want to Exit Early with Complex Logic

Streams have methods like `findFirst()` or `anyMatch()` for early exit. But if our logic is more complicated, like we want to stop after checking several conditions, a loop with `break` is easier to follow and debug.

3. When we are Dealing with Checked Exceptions

Streams don't handle checked exceptions well inside functions like map() or filter(). We need to wrap the code in try-catch blocks, which makes it messy and hard to read. With loops, you can just handle exceptions normally.

4. When the Code Has Side Effects

Streams are meant for pure functions, can not change the outside variables. If we are updating variables outside the stream (like counters or flags), it can lead to bugs, especially if the stream is parallel. Loops are safer and more predictable in such cases.

5. When We Need to Use Indexes

Streams don't give the easy access to element positions. If the logic depends on knowing the index (like comparing the current and next item), a regular loop is much easier to write and understand.

105 What's the difference between "this" in a lambda vs. an anonymous?

In Java, this just means "**the current object**"

But when we write code inside lambdas or anonymous classes, this behaves differently

In a Lambda, this refers to the outer class , the class where the lambda is written.

Example:

```
public class MyClass {
    private String name = "Coding Lyf";

    public void greet() {
        Runnable r = () -> {
            System.out.println("Hello, " + this.name); // accessing instance variable
using `this` 
        };
        r.run();
    }

    public static void main(String[] args) {
        new MyClass().greet();
    }
}
```

In above example, this.name prints "Coding Lyf" , in Lambdas this refers to the class where it is written. But In an anonymous class, this refers to the anonymous instance NOT to outer class

Example : Accessing instance variable in **anonymous class**

```
public class MyClass {
    private String name = "CodingLyf";

    public void greet() {
        Runnable r = new Runnable() {
            String name = "Inside anonymous";
        };
        r.run();
    }
}
```

```

public void run() {
    // `this.name` doesn't work , 'this' refers to anonymous class, not
MyClass
    System.out.println("Hello, " + this.name);
    System.out.println("Hello, " + MyClass.this.name); // have to use
MyClass.this
}
};

r.run();
}

public static void main(String[] args) {
    new MyClass().greet();
}
}

```

Here this.name prints '`Inside anonymous`'. MyClass.this.name prints '`CodingLyf`'.

Indirect Questions:

1. Can you use this inside a Stream lambda?

Yes. Lambdas have access to the enclosing class's this.

2. Does this behave the same inside a lambda and an anonymous class?

No. In a lambda, this refers to the enclosing class. In an anonymous class, this refers to the anonymous instance

3. Can you access instance methods or fields inside Stream lambdas?

Yes, because this is accessible.

106 Features of different versions?

Interview Tip: Generally, we will be asked the latest version we are using and features of it. Let's see all major versions and important features. As we have seen Java 8 features already, let's see from Java 11 – Java 23

Java 9:

Java 9 introduced convenience static factory methods on the List, Set, and Map interfaces to easily create immutable collections. These methods provide a concise way to create collections whose state cannot be changed after construction. Attempting to modify such collections will result in an UnsupportedOperationException.

`List.of()`, `Set.of()`, `Map.of()`, `Map.ofEntries()`

```

// Empty immutable list
List<String> emptyList = List.of();
System.out.println("Empty List: " + emptyList);

```

```
// Immutable list with elements
List<String> fruits = List.of("Apple", "Banana", "Orange");
System.out.println("Fruits List: " + fruits);

// Attempting to modify will throw UnsupportedOperationException
// fruits.add("Grape");
```

Java 10:

- Local variable reference
- Optional API: orElseThrow

Local variable reference : Local Variable Type Inference , allows to declare the local variables without explicitly stating their type. Java looks at the value we assign and automatically guesses the correct type for the variable. This feature utilizes the reserved type name **var**

```
public class VarExample {
    public static void main(String[] args) {
        var message = "Hello, Java 10"; // inferred as String
        var number = 42; // inferred as int
        var list = List.of("a", "b", "c"); // inferred as List<String>

        System.out.println(message);
        System.out.println(number);
        System.out.println(list);
    }
}
```

Java 11 (Long Term Support)

1. var keyword in lambda expressions

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.forEach((var number) -> {
    System.out.println(number);
});
```

2. HTTP Client: Java 11 includes a new HTTP client API that provides a more modern and efficient way to send HTTP requests and receive responses. The new HTTP client API supports both HTTP/1.1 and HTTP/2 protocols and includes features like HTTP/2 push, server push, and WebSocket. The new API is also asynchronous and non-blocking, making it more scalable and performant than the previous HttpURLConnection API.

3. String methods: isBlank(), strip(), stripLeading(), stripTrailing(), lines(), repeat(int)

Java 12:

Collectors.teeing(): The Collectors.teeing() method in Java was introduced in Java 12 as part of the Stream API. It allows us to collect the same stream using two different collectors at once, and then combine their results using a merging function (BiFunction).

Reference: [Mastering Collectors.teeing\(\) in Java Streams \(With Examples\) | by A cup of JAVA coffee with NeeSri | Medium](#)

File.mismatch() Method: A new method mismatch(Path, Path) was added to the Files class, allowing for easy comparison of two files and returning the index of the first differing byte or -1 if they are identical.

String API Enhancements: New methods like indent() and transform() were added to the String class for easier string manipulation.

String.indent(int n): To add or remove indentation (leading spaces) from each line of a multiline string.

String.transform(): To apply a custom transformation to a string

```
String result = "hello"
    .transform(str -> str.toUpperCase())
    .transform(str -> "Prefix_" + str);

System.out.println(result);
```

Java 14:

Switch Expressions: In Java 14, switch expressions were finalized Standard feature. This allowed switch to return values and eliminated the need for explicit break statements

```
int day = 2;
String dayName = switch (day) {
    case 1, 7 -> "Weekend";
    case 2, 3, 4, 5, 6 -> "Weekday";
    default -> "Invalid day";
};
System.out.println(dayName);
```

- The switch statement can now be used as an expression that returns a value.
- Arrow (->) syntax introduced for cleaner case expressions.
- The break statement is no longer needed in this context.

Yield: the yield keyword is used within switch expressions to return a value from a case block.

```
String dayOfWeek = "Monday";
String activity = switch (dayOfWeek) {
    case "Monday" -> {
        System.out.println("Starting the week!");
        yield "Go to work";
    }
}
```

```

        case "Saturday", "Sunday" -> "Relax";
        default -> "Study";
    };
    System.out.println("Today's activity: " + activity);
}

```

Java 15

Text Blocks: A Text Block is a multi-line string literal enclosed in triple double quotes (""""). It simplifies the creation of strings that span multiple lines, such as HTML, JSON, SQL queries, or other structured text.

```

String htmlContent = """
    <html>
        <body>
            <h1>Welcome!</h1>
        </body>
    </html>
""";

```

Java 16:

Record: In Java, a record is a special type of class declaration aimed at reducing the boilerplate code. Java records were introduced with the intention to be used as a fast way to create data carrier classes, i.e. the classes whose objective is to simply contain data and carry it between modules, also known as POJOs (Plain Old Java Objects) and DTOs (Data Transfer Objects)

```
public record Person(String name, int age) {}
```

This one line gives:

- A constructor
- getName() and getAge() methods
- A readable toString()
- Proper equals() and hashCode()

Note: All fields are immutable.

Pattern matching for instanceof in Java simplifies type checking and casting by combining them into a single operation.

Before pattern matching, checking the type of an object and then casting it to access its specific members required two separate steps:

```

Object obj = "Hello World";
if (obj instanceof String) {
    String s = (String) obj; // Explicit cast
    System.out.println(s.toUpperCase());
}

```

With pattern matching, as a standard feature in Java 16, the explicit casting is no longer needed

```
Object obj = "Hello World";
if (obj instanceof String s) { // 's' is the pattern variable
    System.out.println(s.toUpperCase());
}
```

Java 17:

Sealed classes provide a mechanism to restrict the inheritance hierarchy of a class or interface. This means we can explicitly control which classes are allowed to extend a sealed class or implement a sealed interface.

```
public sealed class Shape permits Circle, Square, Triangle {
    // ...
}
```

A class or interface is declared as sealed using the `sealed` modifier.

The `permits` clause explicitly lists the classes or interfaces that are allowed to extend or implement the sealed type.

When we use `permits` in a sealed class, the classes or interfaces we list must be marked with one of these:

- **final**: This means the class can't be extended by any other class. It is the end of the chain.
- **sealed**: This class is also sealed, so it controls who can extend it, just like the parent.
- **non-sealed**: This class removes the restriction. Any class can extend it now, breaking the sealed rule from this point on.

Java 21:

Pattern matching for switch

In older versions of Java, `switch` could only work with fixed values like numbers, strings, or enums. But from Java 21, with **pattern matching for switch**, we can now use `switch` to check both the **type** of an object and do something based on that type

This means we don't need to write extra `if` or `instanceof` checks. Java handles that for you right inside the `switch`.

```
static String handle(Object obj) {
    return switch (obj) {
        case String s -> "It is a string: " + s;
        case Integer i -> "It is a number: " + i;
        case null -> "It is null";
        default -> "Something else";
    };
}
```

So, if we pass a String, it will go into the String s case. If it is an Integer, it goes into the Integer i case.

Record Patterns:

Records are a simple way to create classes that just hold data (introduced in Java 14, made stable in Java 16).

Now, starting with Java 21, With **Record Patterns** we can easily extract the values from a record using pattern matching, without manually calling getters.

Before record patterns, if we had a record “**record Person(String name, int age) {}**”, we read like

```
Person p = new Person("Macha", 25);
String name = p.name();
int age = p.age();
```

With **record patterns**, Java do this in a more readable way, especially inside if or switch:

```
if (p instanceof Person(String name, int age)) {
    System.out.println("Name: " + name + ", Age: " + age);
}
```

Here, Java is checking if p is a Person, and at the same time, it pulls out name and age, no need to call .name() or .age() manually.

Switch:

```
switch (p) {
case Person(String name, int age) ->
    System.out.println(name + " is " + age + " years old");
}
```

- Record patterns make it easier to **extract data from records**.
- We can use them with instanceof or switch.

Sequenced Interfaces:

Java introduced 3 new interfaces in the collection hierarchy.

- SequencedCollections

- SequencedMap
- SequencedSet

SequencedCollection works with ArrayList and LinkedList, it gives new methods like

- getFirst() – gives the first item
- getLast() – gives the last item
- addFirst() – adds an item at the beginning
- addLast() – adds at the end
- reversed() – gives a reversed view of the collection

```
SequencedCollection<String> sc = new ArrayList<>();
sc.addLast("Apple");
sc.addLast("Banana");
sc.addFirst("Mango");

System.out.println(sc); // [Mango, Apple, Banana]
System.out.println(sc.getFirst()); // Mango
System.out.println(sc.getLast()); // Banana
System.out.println(sc.reversed()); // [Banana, Apple, Mango]
```

SequencedSet and SequencedMap works in LinkedHashSet and LinkedHashMap respectively.

Output Based Questions

107 Output-Based 1: How many String objects will be created in memory from below code?

```
String s1 = new String("hello");
String s2 = "hello";
```

Answer and Explanation: 2 String objects

1. String s1 = new String("hello");

- This creates a new object in heap memory and String "hello" in String pool.
- So now there are two strings:
 - One in the string pool ("hello")
 - One in the heap (s1)

2. String s2 = "hello";

- This points directly to the string pool version of "hello".
- So now:
 - s1 points to heap object
 - s2 points to pool object

108 Output-Based 2: How many String objects will be created in memory from below code?

```
String s1 = new String("hello");
String s2 = "hello";
String s3 = s1.intern();
We have added 3rd line s1.intern();
```

intern(): When invoked on a String object, it checks if an identical string already exists in the pool. If found, it returns a reference to the existing string. If not, it adds the string to the pool and returns a reference to the newly added string.

Since "hello" is already existed in the string pool, It returns same object. So s1.intern() wont create new string object.

Answer is 2 String objects.

109 Output-Based 3: String ==?

```
String a = "CodingLyf";
String b = "CodingLyf";

System.out.println(a == b);
System.out.println(a.equals(b));
```

Explanation:

a == b is true

- Both a and b are string literals with the same content.
- Java stores string literals in the String pool, so both point to the same memory location.
- Same reference -> == is true.

a.equals(b) is true

- .equals() checks if the contents are the same.
- "CodingLyf" equals "CodingLyf" , so it returns true.

110 Output-Based 4: String == and equals?

```
String a = "hello";
String b = "hello";
String c = new String("hello");

System.out.println(a == b);
System.out.println(a == c);
System.out.println(a.equals(c));
```

Explanation:

- `a == b`: true, because both refer to the same string literal in the String pool.
- `a == c`: false, because `new String()` creates a new object in heap memory.
- `a.equals(c)`: true, because `equals()` compares values, not references.

111 Output-Based 5: String concatenation?

```
String x = "codinglyf";
String y = "coding";
String z = y + "lyf";

System.out.println(x == z);
System.out.println(x.equals(z));
```

Explanation:

- `z = y + "lyf"` is built at runtime using a variable, so it is a new object in memory.
- Even though `x` and `z` look the same, they are not the same object, so `==` gives false.
- `.equals()` checks the actual text inside both strings, which is the same, so it returns true.

112 Output-Based 6: valueOf()?

```
String a = "hello";
String b = String.valueOf(a);
System.out.println(a == b);
```

Explanation:

- `a` is "hello"
- The method `String.valueOf(Object obj)` does this internally: `return (obj == null) ? "null" : obj.toString();`
- `a.toString()` for a `String` just returns itself (not a new object)..
- So `b` ends up pointing to the **same object** as `a`. Output is true

113 Output-Based 7: Substring()?

```
String a = "hello";
String b = a.substring(0);
System.out.println(a == b);
```

Explanation:

- "hello" is stored in the String Pool, "a" points to it.
- `a.substring(0)` returns the same object when the whole string is taken.

- So `a == b` is true because both references point to the same string.

114 Output-Based 8: String == , equals and intern()?

```
String s1 = new String("hello");
String s2 = "hello";
String s3 = s1.intern();
System.out.println(s1 == s3)
System.out.println(s3 == s2);
```

Explanation:

`s1` is created using `new`, so it is a separate object in heap, and creates string literal "hello" in the pool.
`s2` points to the string pool version of "hello".

`s3` = Since "hello" is already there in string pool, `s1.intern()` won't create new `String`, it just returns the pool version of "hello", so `s3 == s2` is true, but `s1 == s3` is false

115 Output-Based 9: On StringBuilder.

```
StringBuilder sb1 = new StringBuilder("abc");
StringBuilder sb2 = new StringBuilder("abc");;

System.out.println(sb1 == sb2);
System.out.println(sb1.equals(sb2));
```

Explanation: Both print false

`sb1 == sb2`

- This checks reference equality.
- `sb1` and `sb2` point to two different objects in memory, even though the contents are the same.
- So this returns false.

`sb1.equals(sb2)`

- We might expect this to compare contents, like `String.equals()` does.
- But `StringBuilder` does not override `equals()` from `Object`.
- So, it behaves like `==`, it still checks reference equality.
- Again, false.

To compare the content in `StringBuilder`, we need to use

```
System.out.println(sb1.toString().equals(sb2.toString()));
```

116 Output-Based 10: On StringBuilder, Checking the reference and content.

```
StringBuilder sb1 = new StringBuilder("abc");
```

```
StringBuilder sb2 = sb1;
sb1.append("d");
System.out.println(sb1 == sb2);
System.out.println(sb1.equals(sb2));
```

Explanation: Both print true**sb1 == sb2**

- We are assigning sb2 = sb1, so both references point to the same object in memory.
- So == returns true.

sb1.append("d")

- We have modified the content of the object both sb1 and sb2 point to.
- Since S1 and S2 refer to the same object, the change is visible to both.

sb1.equals(sb2)

- Again, StringBuilder doesn't override .equals(), so it uses Object.equals, so it checks ==.
- And since both refer to the same object, it returns true.

117 Output-Based 11: Array reference comparison vs. content comparison.

```
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};

System.out.println(a == b);
System.out.println(Arrays.equals(a, b));
```

Explanation: false and true**a == b**

- This compares the references (memory addresses).
- a and b are two separate array objects, even though their contents are the same.
- So, this prints false.

Arrays.equals(a, b)

- This compares the contents of the arrays, element by element.
- Both arrays contain the same values in the same order: {1, 2, 3}.
- So, this prints true.

118 Output-Based 12: Pass by Value.

```
class Main {
    public static void change(int x) {
        x = 100;
```

```

    }

    public static void main(String[] args) {
        int a = 50;
        change(a);
        System.out.println(a);
    }
}

```

Explanation:

- In Java, **all arguments are passed by value**, even primitives like int.
- So, when change(a) is called, a **copy of a's value (50)** is passed to x.
- Inside change() method, x = 100 modifies the **copy**, only local variable will be change, not the original variable.
- The original 'a' in main() remains unchanged, so the output is 50.

119 Output-Based 13: Passing the reference.

```

public class Test {
    public static void change(StringBuilder sb) {
        sb.append(" Lyf");
    }

    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Coding");
        change(sb);
        System.out.println(sb);
    }
}

```

Explanation:

- In Java, everything is passed by value, including object references.
- sb in main() holds a reference to the StringBuilder object.
- When passed to change(), a copy of that reference is passed, now both sb in main() and sb in change() point to the same object in memory.
- sb.append(" Lyf") changes the actual object in memory, not the reference variable. so the change is visible even after the method ends.
- So, System.out.println(sb) prints Coding Lyf.

Point to remember:

Java passes object references by value.

We are not passing the object, we are passing a copy of its reference, but both refer to the same underlying object. So, changes to the object are reflected even outside the method.

120 Output-Based 14: Passing the reference and reassigning.

```
public class Test {
    public static void reassign(StringBuilder sb) {
        sb = new StringBuilder("New");
    }

    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Original");
        reassign(sb);
        System.out.println(sb);
    }
}
```

Explanation:

- In main(), sb points to a StringBuilder object with the content "Original".
- When you call reassign(sb), a copy of the reference is passed into the method.
- Inside reassign(), sb = new StringBuilder("New") just reassigns the local copy to a new object. Now it is completely new object.
- So, the original sb in main() still points to "Original", so output is "Original".

121 Output-Based 15: Passing reference of the array.

```
public class Test {
    public static void modify(int[] arr) {
        arr[0] = 99;
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        modify(nums);
        System.out.println(Arrays.toString(nums));
    }
}
```

Explanation:

- In main(), nums is an array with values {1, 2, 3}.

- When passed to modify(), a copy of the reference to the array is passed.
- Java passes object references by value, so both arr and nums refer to the same array object.
- arr[0] = 99 changes the content of the original array in memory, since both arr and nums point to the same object.
- So, when we print nums, the change is visible.

122 Output-Based 16: Passing reference of the array's copy.

```
public class Test {
    public static void modify(int[] arr) {
        arr[0] = 99;
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        modify(Arrays.copyOf(nums, nums.length));
        System.out.println(Arrays.toString(nums));
    }
}
```

Explanation:

- Arrays.copyOf(nums, nums.length) creates a new array object with the same contents as nums. So changes made inside the method do not affect the original.
- This new array is passed to modify(), where arr[0] = 99 modifies only the copy, not the original.
- The original nums array in main() stays unchanged.
- So System.out.println(nums) prints [1, 2, 3].

123 Output-Based 17: Final variable.

```
final int x = 10;
x = 20;
System.out.println(x);
```

Explanation:

- final primitive = value can't be changed.
- Compilation error: cannot assign a value to final variable 'x',

124 Output-Based 18: Final array.

```
final int[] arr = {1, 2, 3};
arr[0] = 99;
```

```
System.out.println(Arrays.toString(arr));
```

Explanation:

- final means, we can't reassign the value to 'arr' but we can change the content.
- So arr[0] = 99 is allowed and updates the original array.
- Output is [99, 2, 3]

125 Output-Based 19: Final Object.

```
class Person {
    String name = "Alex";
}

public class Test {
    public static void main(String[] args) {
        final Person p = new Person();
        p.name = "Sam";
        System.out.println(p.name);
    }
}
```

Explanation:

- final means, we can't reassign the value to Person object 'p' but we can change the state.

```
final Person p = new Person();
p = new Person(); //It throws compilation error as we are reassigning
```

- So p.name updates the name.
- Output is "Sam"

126 Output-Based 20: Multiple Exceptions.

```
try {
    // code
} catch (Exception e) {
    // handle
} catch (ArithmetricException ae) {
    // handle
}
```

Explanation:

- More specific exceptions must come before general exceptions.
- It throws Compile Time Error

127 Output-Based 21: Try with return and finally.

```
try {
```

```

        return;
} finally {
    System.out.println("In finally");
}

```

Explanation:

- Even try return the value, finally still executes before method returns.
- So, Output is “In finally”.

128 Output-Based 22: Try and Finally with returns.

```

public static int test1() {
    try {
        return 1;
    } finally {
        return 2;
    }
}

```

Explanation:

- Output is 2, because finally overrides the return of try.

129 Output-Based 23: Execution flow with static

```

public class Test {

    static int x = Init();

    static {
        System.out.println("Static Block");
    }

    private static int Init() {
        System.out.println("Static variable initialization");
        return 1;
    }

    public static void main(String[] args) {
    }
}

```

Explanation:

When the class is loaded (before main() runs), Java does these things in order for static:

- All static variables are initialized first in the order they are defined.
- Static blocks are executed in the order they appear in the code.

- Here in this class static variable is defined at first so the output is
 1. Static variable initialization
 2. Static block
 3. Main method (Does thing in above code)

Points to remember:

Static blocks & static variables run once when the class is loaded.

Instance variables and instance blocks run before the constructor for each new object.

130 Output-Based 24: Execution flow with instance.

```
public class Test {
    int x = instanceInit();

    {
        System.out.println("Instance Block");
    }

    Test() {
        System.out.println("Constructor");
    }

    private int instanceInit() {
        System.out.println("Instance Variable Initialization");
        return 10;
    }

    public static void main(String[] args) {
        new Test();
    }
}
```

Explanation:

- When new Test() is called, object creation starts.
- First, instance variable x is initialized, which calls instanceInit() and prints: Instance Variable Initialization.
- Then the instance block runs and prints: Instance Block.
- After that, the constructor runs and prints: Constructor.
- Concept: For every object, Java runs instance variable initializations (depends on order) > instance blocks (depends on order) > constructor

Answer is:

Instance Variable Initialization

Instance Block

Constructor

131 Output-Based 25: Overriding: Method overriding basics.

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

Explanation:

- A obj = new B(); creates a reference of type A but an object of type B , this is upcasting.
- At runtime, method overriding comes into picture , we call it as dynamic method dispatch.
- Since show() is overridden in class B, the method from B gets called, not A.
- Output is B. It demonstrates the runtime polymorphism.

What is Dynamic method dispatch?

Dynamic method dispatch is a concept in Java where a call to an overridden method is resolved at runtime rather than compile time.

It allows a parent class reference to refer to a child class object, and when a method is called, the version in the actual object's class runs, not the reference type. This is key for polymorphism. For example, if Animal a = new Dog(); and both have a sound() method, calling a.sound() will run Dog's sound(), not Animal's. It makes code flexible, letting one reference handle different object types safely and efficiently.

132 Output-Based 26: Overriding: Variable hiding.

```
class A {
    int x = 10;
}
class B extends A {
    int x = 20;
}
public class Test {
```

```

public static void main(String[] args) {
    A obj = new B();
    System.out.println(obj.x);
}
}

```

Explanation:

- A obj = new B(); it is upcasting , means reference is of type A but object is of type B.
- int x = 10 in A and int x = 20 in B . This is variable hiding, not overriding.
- **Point to remember:** Variable resolution happens at compile-time based on reference type, not object type.
- So, obj.x access the A's variable, Output: 10.

133 Output-Based 27: Overriding: Constructor Execution Order.

```

class A {
    A() {
        System.out.println("A constructor");
    }
}
class B extends A {
    B() {
        System.out.println("B constructor");
    }
}
public class Test {
    public static void main(String[] args) {
        new B();
    }
}

```

Explanation:

- When new B() is called, B's constructor runs.
- But since B extends A, A's constructor is called first which is implicit super() call.
- This shows constructor chaining in inheritance.
- So, output is:

A constructor

B constructor

134 Output-Based 28: Overriding: Static method hiding.

```

class A {
    static void show() {
        System.out.println("A static");
    }
}
class B extends A {
}

```

```

    static void show() {
        System.out.println("B static");
    }
}

public class Test {
    public static void main(String[] args) {
        A obj = new B();
        obj.show();
    }
}

```

Explanation:

- show() is a static method, which means it is not overridden but it is hidden, a concept called method hiding.
- A obj = new B(); is upcasting, but with static methods, only the reference type matters, not the object.
- Since obj is of type A, obj.show() calls A's version, not B's.
- So the output is: A static.

135 Output-Based 29: Overriding: Accessing Child-Specific Method.

```

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

class B extends A {
    void display() {
        System.out.println("B display");
    }
}

public class Test {
    public static void main(String[] args) {
        A obj = new B();
        obj.display();
    }
}

```

Explanation:

- A obj = new B(); here reference is of type A, object is of type B.
- We are trying to call obj.display();, but display() is not defined in class A.
- Even though the object is of type B, the compiler checks in class A for available methods because the reference type is A.
- Since A has no display() method, this results in a compile-time error

How you fix this compile time error?

Answer: Cast 'obj' to B class.

```
B b = (B) obj;
b.display();
```

136 Output-Based 30: String Concatenation.

```
class Test {
    public static void main(String[] args) {
        String s1 = "S" + 1 + 2;
        String s2 = "S" + (1 + 2);
        String s3 = 1 + 2 + "S";

        System.out.println(s1); //S12
        System.out.println(s2); //S3
        System.out.println(s3); //3S
    }
}
```

Explanation:

In `s1 = "S" + 1 + 2`, Java processes expressions left to right. Since "S" is a string, 1 is converted to "1" and concatenated, resulting in "S1". Then "S1" concatenates with 2 (converted to "2"), so "S12".

For `s2 = "S" + (1 + 2)`, the parentheses force the addition `1 + 2` to happen first as integer arithmetic, giving 3. Then "S" concatenates with "3" to form "S3".

In `s3 = 1 + 2 + "S"`, the addition `1 + 2` happens first because both are integers, resulting in 3. Then 3 is concatenated with "S" to get "3S".

137 Output-Based 31: Overriding: Type Casting 1.

```
class A {}
class B extends A {}
class C extends B {}
public class Test {
    public static void main(String[] args) {
        A a = new C();
        B b = (B) a;
        System.out.println("Cast successful");
    }
}
```

Explanation:

- C extends B, which extends A, so C is indirectly a subclass of A.
- `A a = new C();` is valid because we are assigning a subclass (C) to a superclass reference (A), that is **upcasting**.

- B b = (B) a; is **downcasting**, means are converting ‘a’ reference to class B. It is safe here because ‘a’ is actually pointing to a C object and C is a subclass of B.
- Since the cast is valid at runtime and the output is: Cast successful.

138 Output-Based 32: Overriding: Type Casting 2.

```
class A {}
class B extends A {}
class C extends B {}
public class Test {
    public static void main(String[] args) {
        A a = new B();
        C c = (C) a;
        System.out.println("Cast successful");
    }
}
```

Explanation:

- A a = new B(); we are storing a B object in a reference of type A. This is **upcasting** and perfectly safe.
- Then we write: C c = (C) a; this is **downcasting**. We are telling Java, “Treat this object as a C.”
- But here is the problem: the actual object in memory is a B, **not a C**. So, the cast is **logically wrong**, the compiler allows it because C is related to A.
- At runtime, Java checks the object and throws: ClassCastException

139 Output-Based 33: Overriding: Field Access.

```
class A {
    int val = 1;
}
class B extends A {
    int val = 2;
}
public class Test {
    public static void main(String[] args) {
        A obj = new B();
        System.out.println(obj.val);
    }
}
```

Explanation:

- Both class A and class B have a field named val. This is **variable hiding**, not overriding.
- A obj = new B(); we are creating a B object but storing it in an A reference.

- When we access obj.val, Java looks at the reference type, not the object type and since obj is of type A, it fetches A's val.
- So the output is 1.
- **Point to remember:** fields are resolved at compile-time by reference type, but methods resolve at run time which use runtime polymorphism.

140 Output-Based 34: Overriding: Private Method in Parent.

```
class A {
    private void show() {
        System.out.println("A");
    }
}
class B extends A {
    public void show() {
        System.out.println("B");
    }
}
public class Test {
    public static void main(String[] args) {
        A obj = new B();
        obj.show();
    }
}
```

Explanation:

- In class A, show() is marked private, which means it is not inherited by class B.
- In class B, we define a new public show() method, but it doesn't override A's method.
- A doesn't have a visible show() method (it is private), so we can't call it through an A reference..
- So obj.show(); causes a **compile-time error**

Point to remember: Private methods are not inherited, so no overriding happens.

New B().show() will work, it prints B. It treats show() as normal function

141 Output-Based 35: Overriding: Protected Method Access

```
class A {
    protected void test() {
        System.out.println("A test");
    }
}
class B extends A {
    void test() {
```

```

        System.out.println("D test");
    }
}
public class Test {
    public static void main(String[] args) {
        new B().test();
    }
}

```

Point to remember: In Java, when we override a method in a subclass, the overridden method (the method in the subclass) must have the same or more accessible (less restrictive) access modifier than the overriding method (the method in the parent class). **Keep same visibility or increase it, but never reduce it**

Explanation:

- A protected method can be accessed within the same package or by subclasses (even in different packages).
- A default (package-private) method is less accessible than protected because it is only accessible within the same package.
- test() in class A is protected, means visible to subclasses and classes in the same package.
- test() in class B is default/package-private (since we didn't specify any modifier).
- So, we are reducing visibility, which is not allowed, even if the classes are in the same package.
- That is why the compiler throws error: **Cannot reduce the visibility of the inherited method from A**

Solution: Either make the method protected or public in class B:

142 Output-Based 36: Overriding: Parent public, child private

```

class A {
    public void test() {
        System.out.println("A test");
    }
}
class B extends A {
    private void test() {
        System.out.println("B test");
    }
}
public class Test {
    public static void main(String[] args) {
        new B().test();
    }
}

```

Point to remember: In Java, when we override a method in a subclass, the overridden method (the method in the subclass) must have the same or more accessible (less restrictive) access modifier than the method (the method in the parent class). **Keep same visibility or increase it, but never reduce it**

Explanation:

- Class A has a public method test() that prints "A test".
- Class B extends A and attempts to override test() but declares it as private.
- In Java, a subclass cannot reduce the visibility of an inherited method. This is a compilation error because private is more restrictive than public

Solution: make the method public in class B:

143 Output-Based 37: Overriding: Checked Exceptions

```
class Parent {
    void readFile() throws IOException {
        System.out.println("Reading file");
    }
}

class Child extends Parent {
    @Override
    void readFile() throws Exception {
        System.out.println("Reading file in child");
    }
}
```

Point to Remember: In method overriding, the child class cannot throw broader or new checked exceptions than the parent method. It can throw the same or narrower checked exceptions.

Explanation:

- Parent.readFile() declares throws IOException, which is a checked exception.
- In Child, the overridden method declares throws Exception, which is more general than IOException.
- In Java, an overridden method can only throw the same or narrower checked exceptions , not a broader one.
- Since Exception is broader than IOException, the compiler throws an error

144 Output-Based 38: Overriding: Un-Checked Exceptions

```
class Parent {
    void connect() throws Exception{
        System.out.println("Connecting to server");
    }
}

class Child extends Parent {
```

```

@Override
void connect() throws NullPointerException {
    System.out.println("Connecting to child server");
}
}

```

Explanation:

- Parent.connect() throws Exception, a checked exception.
- Child.connect() throws NullPointerException, which is an unchecked (runtime) exception.
- **Point to Remember:** In Java, Child class methods can throw unchecked exceptions freely, regardless of what the Parent declares

145 Output-Based 40: Overriding: Method Call in Constructor

```

class A {
    A() {
        show();
    }
    void show() {
        System.out.println("A show");
    }
}
class B extends A {
    int x = 10;
    void show() {
        System.out.println("B show: " + x);
    }
}
public class Test {
    public static void main(String[] args) {
        new B();
    }
}

```

Explanation:

- new B(); starts object creation, which first calls A's constructor (because B extends A).
- Inside A() constructor, it calls show().
- Since show() is overridden in B, Java calls B's version , this is runtime polymorphism.
- But here is the catch: when A() is running, the B part of the object isn't fully initialized yet.
- So even though control goes to B.show(), the field 'x' in B hasn't been assigned 10 yet, 'x' holds the default value 0.
- That is why the output is: B show: 0.

146 Output-Based 41: Overriding: Method Call in Constructor with static

```

class A {
    A() {
        show();
    }
    static void show() {
        System.out.println("A show");
    }
}
class B extends A {
    static int x = 10;
    static void show() {
        System.out.println("B show: " + x);
    }
}
public class Test {
    public static void main(String[] args) {
        new B().show();
    }
}

```

Explanation:

- new B() triggers object creation, which first calls the constructor of A, since B extends A.
- Inside A() constructor, it calls the static method show().
- Static methods are not overridden, they are hidden, so the version that gets called is based on the reference type at compile-time, not runtime.
- In this case, since show() is called from A's constructor, the method A.show() gets executed, not B.show().
- So, the line System.out.println("A show"); runs.
- After the object is fully created, new B().show() is executed in main(). Now the reference is of type B, so B.show() runs, and by this point, static variable x = 10 is initialized.
- Output is A show and B show: 10

147 Output-Based 42: Interface Default Method vs Class Method

```

class A {
    public void msg() {
        System.out.println("Class");
    }
}
interface B {
    default void msg() {
        System.out.println("Interface");
    }
}

```

```
class Test extends A implements B {
    public static void main(String[] args) {
        Test m = new Test();
        m.msg();
    }
}
```

Explanation:

- Class A has a regular instance method msg().
- Interface B has a default method msg().
- Class Test extends A and implements B.
- **Point to remember:** When msg() is called, Java gives priority to the class method over the default method from the interface.
- So, the output is: Class.

148 Output-Based 43: Overriding: Diamond Problem (Interface)

```
interface A {
    default void show() {
        System.out.println("A");
    }
}
interface B {
    default void show() {
        System.out.println("B");
    }
}
class C implements A, B {
    public void show() {
        A.super.show();
        B.super.show();
        System.out.println("C");
    }
}
public class Test {
    public static void main(String[] args) {
        new C().show();
    }
}
```

Explanation:

- Both Interfaces A and B have a default method with **same signature**: void show().
- When class C implements both A and B, and both interfaces have a default method with the same name, Java doesn't know which one to inherit, this is called Diamond problem.

- Class C implements both A and B, so Java forces it to override show() to resolve the conflict.
- Inside C's show() method, A.super.show() calls A's version and B.super.show() calls B's version
- The class can still access specific interface methods using InterfaceName.super.methodName()
- After calling both interface methods, System.out.println("C") prints "C".
- Final output is: A, B, C

149 Output-Based 44: Abstract Class: Main Method

```
abstract class Test {
    public static void main(String[] args) {
        System.out.println("Yes, abstract class can have main");
    }
}
```

Explanation:

- An abstract class can have a main method and run like any other class. Abstract class only prevents instantiation, not execution.

150 Output-Based 45: Abstract Class: Private Method

```
abstract class A {
    abstract private void run();
}
```

Explanation:

- An abstract method can't be a private method, it causes compile time error.

151 Output-Based 46: Abstract Class: Multiple abstract classes in chain

```
abstract class A {
    abstract void run();
}
abstract class B extends A {}
class C extends B {
    void run() {
        System.out.println("Running from C");
    }
}
public class Test {
    public static void main(String[] args) {
        A obj = new C();
        obj.run();
    }
}
```

Explanation:

- Class A defines an abstract method run(), and class B extends it without implementing run() so B is still abstract.
- Class C extends B and provides the actual implementation of run().
- In main(), A obj = new C(); it is a valid upcasting, and obj.run() calls C's version due to runtime polymorphism.
- Output is: Running from C.

152 Output-Based 47: Overloading: Type Promotion

```
class Test {
    void show(int a, long b) {
        System.out.println("int, long");
    }

    void show(long a, int b) {
        System.out.println("long, int");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.show(10, 20);
    }
}
```

Explanation:

- There are two overloaded methods: show(int, long) and show(long, int).
- We call t.show(10, 20), where both 10 and 20 are int.
- Java doesn't know which method to use, should it convert the first int to long or long to int?
- This causes compile time error.

153 Output-Based 48: Overloading: With Varargs vs Exact Match

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

    void show(String... s) {
        System.out.println("Varargs");
    }

    public static void main(String[] args) {
        Test t = new Test();
    }
}
```

```

        t.show("hello");
    }
}

```

Explanation:

- There are two overloaded show() methods: one takes a single String, the other takes a varargs (String...).
- We call t.show("hello") with a single String argument.
- Java gives **priority to the exact match** over varargs.
- So, show(String s) is chosen instead of show(String... s).
- Output: String.

154 Output-Based 49: Overloading: Autoboxing vs Widening

```

class Test {
    void show(Integer i) {
        System.out.println("Integer");
    }

    void show(long l) {
        System.out.println("long");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.show(10);
    }
}

```

Explanation:

- Two overloaded show() methods: one takes Integer, the other takes long.
- The call t.show(10) passes an int.
- **Point to Remember:** Java prefers **widening** over **boxing** during overload resolution.
- int to long is widening and int to Integer is boxing.
- So, java choose, long.

155 Output-Based 50: Overloading: Primitive Overloading

```

class Test {
    void show(String s) {
        System.out.println("String");
    }

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

```

```

    }

    public static void main(String[] args) {
        Test t = new Test();
        t.show(null);
        t.show("CodingLyf");
    }
}

```

Explanation:

- Two overloaded show() methods: one takes String, the other takes Object.
- **Point to remember:** Overloading prefers more specific types when multiple matches are possible.
- t.show(null) passes null, which matches both, but String is more specific than Object
- "CodingLyf" is a String, so again show(String) is called.
- Output: String, String

156 Output-Based 51: Overloading and Inheritance

```

class A {}
class B extends A {}

class Test {
    void show(A a) {
        System.out.println("A");
    }

    void show(B b) {
        System.out.println("B");
    }

    public static void main(String[] args) {
        A obj = new B();
        Test t = new Test();
        t.show(obj);
    }
}

```

Explanation:

- ‘obj’ is declared as type A but instantiated with new B().
- **Point to Remember:** Method overloading is resolved at compile-time based on reference type, not object type. Overloading depends on reference type, not runtime object type.
- Since obj is of type A, show(A a) is selected.
- Output: A.

157 Output-Based 52: Overloading: Reference vs Object Type

```
class Test {
    void show(Object o) {
        System.out.println("Object");
    }

    void show(String s) {
        System.out.println("String");
    }

    public static void main(String[] args) {
        Object obj = "Hello";
        Test t = new Test();
        t.show(obj);
    }
}
```

This is different from Question No.50. Here we are passing Object.

Explanation:

- Two overloaded show() methods: one takes Object, one takes String.
- ‘obj’ is declared as Object but holds a String value.
- **Point to remember:** Overloading is based on reference type, not runtime value type.
- So, show(Object o) is chosen at compile-time.
- Output: Object

158 Output-Based 53: Super and This: Constructor

```
class A {
    A(String s) {
        System.out.println("A");
    }
}

class B extends A {
    B() {
        System.out.println("B Default");
    }
}

public class Test {
    public static void main(String[] args) {
        new B();
    }
}
```

Explanation:

- Class A has only a parameterized constructor A(String s), it has no default constructor.

- Class B extends A and its constructor must call super(...) explicitly if no default constructor exists in A.
- **Point to remember:** If the superclass does not contain a no-arg constructor, the subclass must explicitly call one of its available constructors.
- B() doesn't call super(...), so the compiler tries to insert super() but no-arg constructor doesn't exist in A.
- Result: Compile-time error (**Implicit super constructor A() is undefined. Must explicitly invoke another constructor**)

159 Output-Based 54: Super and This: Constructor

```
class A {
    A() {
        System.out.println("A");
    }
}
class B extends A {
    B() {
        System.out.println("B");
    }
}
public class Test {
    public static void main(String[] args) {
        new B();
    }
}
```

Explanation:

Class A has a no-arg constructor that prints A.

Class B extends A and also has a no-arg constructor that prints B.

Point to remember: When creating a subclass object, the superclass constructor runs first.

In this example, Since A has no-arg constructor, when we call new B(), it first implicitly invoke the A's constructor.

Output: A, B

160 Output-Based 55: Super and This: Constructor Chaining

```
class A {
    A(int a) {
        System.out.println("A "+a);
    }
}
class B extends A {
    B() {
        this(10);
    }
}
```

```

        System.out.println("B Default");
    }
    B(int b) {
        super(b);
        System.out.println("B int");
    }
}
public class Test {
    public static void main(String[] args) {
        new B();
    }
}

```

Explanation:

- Class A only has a parameterized constructor A(int) that prints "A <value>".
- In B's constructor B(), the first statement is this(10), so it jumps to B(int) before executing "B Default".
- **Point to remember:** this() is used for constructor chaining within the same class, and it must be the first statement.
- Inside B(int), super(b) calls A's constructor A(int), which runs first because superclass constructors always execute before subclass constructors.
- Execution flow: A(int) -> print "A 10", back to B(int) -> print "B int", back to B() -> print "B Default".
- Output: A 10, B, B Default.

161 Output-Based 56: Super and This: Variable Access

```

class A {
    int x = 10;
}
class B extends A {
    void display() {
        x = 20;
        System.out.println(x);
        System.out.println(super.x);
    }
}
public class Test {
    public static void main(String[] args) {
        new B().display();
    }
}

```

Explanation:

new B().display();

Creates an object of B. Since B extends A, it also contains A's x variable.

Inside display()

- `x = 20;` changes the inherited `x` from 10 to 20.
- No data type is written here because we're not declaring a new variable, we're modifying an existing one inherited from A.
- **Note:** If we write `int x = 20;` that will create a new local variable inside `display()` and shadow the inherited one.

`System.out.println(x);` - Prints the `x` from the current object : 20.

`System.out.println(super.x);` - `super.x` also refers to the same `x` variable in A (since it wasn't shadowed in B). The value is now 20 because we modified it earlier.

162 Output-Based 57: Functional Interface: Can a functional interface have default methods?

```
@FunctionalInterface
interface MyFunc {
    void test();
    default void show() {
        System.out.println("Default");
    }
}

public class Test {
    public static void main(String[] args) {
        MyFunc f = () -> System.out.println("Lambda");
        f.test();
        f.show();
    }
}
```

Answer: Yes, a functional interface can have any number of default or static methods but it must have exactly one abstract method.

Explanation:

1. `@FunctionalInterface` ensures the interface has only **one abstract method**.
2. `MyFunc` has one abstract method `test()` and one default method `show()`.
3. **Pointer to remember:** Default methods don't break the functional interface rule because they have a body. Same rule applies even to static methods
4. A lambda expression is used to implement the single abstract method `test()`.
5. Output: Lambda, Default

163 Output-Based 58: Functional Interface: Is Child class is a functional interface?

```
@FunctionalInterface
```

```

interface Parent {
    void baseMethod();
}
@interface Child extends Parent {
    // No new abstract methods allowed except override
}
class Test {
    public static void main(String[] args) {
        Child c = () -> System.out.println("Child");
        c.baseMethod();
    }
}

```

Answer: Yes, Child class is a functional interface.

Explanation:

- Base is a functional interface with one abstract method baseMethod().
- Child extends Base but doesn't declare any **new** abstract methods , it just inherits baseMethod().
- **Point to remember:** A sub-interface remains a functional interface if it only inherits the single abstract method from its parent and doesn't declare any additional abstract methods.
- This means Child still has exactly one abstract method (baseMethod()), so it meets the functional interface rule.
- The lambda expression provides the implementation for that single abstract method, producing output: Child

164 Output-Based 59: Functional Interface: Will the code below compile?

```

@FunctionalInterface
interface MyFunc {
    void test();
}

public class Test {
    public static void main(String[] args) {
        int x = 10;
        MyFunc f = () -> System.out.println(x);
        x = 20;
        f.test();
    }
}

```

Answer: No, It won't compile.

Explanation:

- MyFunc is a functional interface with one abstract method test().

- In the lambda, x is used, so variables must be **final or effectively final**.
- **Point to remember:** Local variables captured in a lambda cannot be modified after they are used inside it.
- Since x is reassigned to 20, it is no longer effectively final.
- Result: Compile-time error

165 Output-Based 60: Functional Interface: Is this a valid Functional Interface?

```
@FunctionalInterface
interface MyInterface {
    int calculate(int x);
    String toString();
}

class Demo {
    public static void main(String[] args) {
        MyInterface m = (a) -> a * a;
        System.out.println(m.calculate(5));
    }
}
```

Answer: Yes, it is a valid Functional Interface.

Explanation:

MyInterface declares one abstract method calculate(int) and overrides toString() from Object.

Methods from Object class are do **not** count as the abstract methods in a functional interface.

Point to remember: A functional interface can still have only one abstract method even if it overrides Object methods.

The lambda (a) -> a * a implements calculate(int).

Output: 25

166 Output-Based 61: Thread: Starting the thread twice?

```
public class Test implements Runnable {
    public void run() {
        System.out.printf("CodingLyf");
    }
    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new Thread(new Test());
        thread1.start();
        thread1.start();
        System.out.println(thread1.getState());
    }
}
```

Answer: Exception, IllegalThreadStateException.

Explanation:

- A Thread object can be started only once , once it is started, it moves from NEW to other states.
- `thread1.start()` the first time works fine and runs `run()`.
- The second `thread1.start()` call throws `IllegalThreadStateException` because the thread is no longer in NEW state.
- **Point to remember:** Once a thread has been started, it cannot be restarted; we must create a new Thread object.
- Output: First "CodingLyf" may print, then a runtime exception is thrown before printing

167 Output-Based 62: Thread: Call run() and start()?

```
public class Test extends Thread implements Runnable {
    public void run() {
        System.out.printf("CodingLyf ");
    }
    public static void main(String[] args) throws InterruptedException {
        Test obj = new Test();
        obj.run();
        obj.start();
    }
}
```

Explanation:

- `Test` extends `Thread` and implements `Runnable`, but `Thread` already implements `Runnable`, so the `implements Runnable` here is redundant.
- Calling `obj.run()` directly just executes it like a normal method in the main thread (no new thread is created).
- Calling `obj.start()` creates a new thread, which then calls `run()`.
- **Point to remember:** `run()` is just a method; `start()` is what actually starts a new thread and then calls `run()` internally.
- Output: CodingLyf CodingLyf

168 Output-Based 63: Thread: Join ?

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> System.out.println("One"));

        Thread t2 = new Thread(() -> {
            try {
                t1.join();
            }
```

```
        System.out.println("Two");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});

t2.start();
t1.start();

t2.join();
System.out.println("Three");
}
```

Explanation:

Creating t1: A Thread is created with a lambda that simply prints "One". At this stage, the thread is in the **NEW** state and hasn't started yet.

Creating t2: Another Thread is created whose job is to first call `t1.join()` (meaning it will wait until `t1` finishes) and then print "Two". It is also in the **NEW** state.

Starting t2 first: The main thread calls `t2.start()`. This moves `t2` to the **RUNNABLE** state, and eventually the JVM schedules it. Inside `t2`'s code, it reaches `t1.join()`. Since `t1` hasn't started yet, `t2` just waits for `t1` to terminate in the future.

Starting t1: The main thread then calls `t1.start()`. Now `t1` moves to **RUNNABLE**, the JVM picks it up, and it executes `System.out.println("One")`. Once it finishes, `t1` moves to the **TERMINATED** state.

Unblocking t2: Because t1 is now terminated, t1.join() in t2 returns immediately. t2 resumes execution and prints "Two", then terminates.

Main thread waiting for t2: The main thread calls `t2.join()`. If `t2` hasn't finished yet, the main thread will block until it does. In this case, it waits just until "Two" is printed.

Printing "Three": Once `t2` has terminated, the main thread continues and prints "Three".

169 Output-Based 64: Thread: Create a deadlock

```
public class Test {  
    public static void main(String[] args) throws InterruptedException {  
        Thread mainThread = Thread.currentThread();  
  
        Thread t = new Thread(() -> {  
            System.out.println("A");  
            try {  
                mainThread.join();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }).start();  
        t.join();  
    }  
}
```

```

        }
        System.out.println("B");
    });

    t.start();
    t.join();
    System.out.println("C");
}
}

```

Explanation:

Thread.currentThread() returns the main thread object, stored in mainThread.

A new thread "t" is created with a task that prints "A", then tries mainThread.join() (waiting for the main thread to finish), and finally would print "B".

The main thread calls t.start(). This moves t to the **RUNNABLE** state, and the JVM schedules it.

As soon as t runs, it prints "A", then calls mainThread.join(). Now "t" is blocked, waiting for the main thread to finish execution.

Right after starting t, the main thread calls t.join(), which means the main thread is now blocked waiting for "t" to finish.

Deadlock: The main thread is waiting for t to finish, but "t" is waiting for the main thread to finish , neither can proceed. This is a **classic deadlock**.

Result: The only output is "A", after which the program hangs forever. "B" and "C" never print.

170 Output-Based 65: Instance vs Static ?

```

class Test {
    int a = 10;
    static int b = 20;

public static void main(String[] args) {
    Test t1 = new Test();
    t1.a = 100;
    t1.b = 200;

    Test t2 = new Test();

    System.out.println("t1.a =" + t1.a + " t1.b =" + t1.b);
    System.out.println("t2.a =" + t2.a + " t2.b =" + t2.b);
}
}

```

Answer:

t1.a =100 t1.b =200

t2.a =10 t2.b =200

Explanation:

- Instance variable a belongs to each object separately, so changing t1.a to 100 doesn't affect t2.a so it stays 10.
- Static variable b is shared by the class itself, not individual objects, so updating t1.b to 200 changes b for all instances.
- So, both t1.b and t2.b reflect the updated value 200.

171 Output-Based 66: Static variable declaration?

```
class Test {
    static int i = 1;

    public static void main(String[] args) {
        static int i = 1;
    }
}
```

Answer: Compile-time Error

Explanation:

We cannot declare a local variable "i" as static inside a method; static is only allowed for class-level members.

172 Output-Based 67: Tricky question on runnable

```
class CodingLyf implements Runnable {
    public void run() {
        System.out.println("Run");
    }
}

class Test {
    public static void main(String[] args) {
        CodingLyf t = new CodingLyf();
        t.start();
        System.out.println("Main");
    }
}
```

Answer: Compile-time Error

Explanation:

- CodingLyf implements Runnable, which means it only has the run() method.
- Runnable does not have a start() method.
- The start() method belongs to the Thread class.

Fix: Wrap the Runnable instance inside a Thread and call start() on that Thread:

```
CodingLyf t = new CodingLyf();
Thread thread = new Thread(t);
thread.start();
System.out.println("Main");
```

Java – Scenario Based and Custom Implementations

173 Java Scenario 1: If – Else Conditions or Switch.

If a customer is a "Premium" member and purchases more than 5 items, give 20% discount. If not premium but buys more than 10 items, give 10% discount. Otherwise, no discount.

```
public class DiscountCalculator {
    public static void main(String[] args) {
        String membershipType = "Premium"; // Change to "Regular" to test
        int itemsPurchased = 7;           // Change to test different cases

        int discount = switch (membershipType) {
            case "Premium" -> (itemsPurchased > 5) ? 20 : 0;
            default          -> (itemsPurchased > 10) ? 10 : 0;
        };

        System.out.println("Membership: " + membershipType);
        System.out.println("Items purchased: " + itemsPurchased);
        System.out.println("Discount: " + discount + "%");
    }
}
```



Explanation:

- We can do with using if-else or switch. Above code is using switch expression from Java 17+.
- The switch directly evaluates membershipType.
- For "Premium", it checks if itemsPurchased > 5, else 0.
- For anything else (non-premium), it checks itemsPurchased > 10.

174 Java Scenario 2: ATM Operations - Switch.

Write logic to handle ATM operations:

- Ask user for operation type (withdraw, deposit, check_balance).
- If withdraw, check if the balance is sufficient; if yes, deduct amount, else show "Insufficient funds".
- If deposit, add amount to balance.
- If check_balance, display balance.

Hint: Use a switch for operation type and if-else for balance validations.

Code link: [JavaSamples/Scenario2_ATMOOperations.java at main · CodingLyf-Fullstack/JavaSamples](#)

175 Java Scenario 3: Traffic Light Action - Switch.

Given the current lightColor (RED, YELLOW, GREEN), determine the action:

- RED > “Stop”
- YELLOW > if pedestrianWaiting == true, print “Stop, pedestrian crossing”; else “Slow down”.
- GREEN > if emergencyVehicle == true, print “Give way to emergency vehicle”; else “Go”.

Hint: Use switch for light color, if-else inside each case for conditions.

Code link: [JavaSamples/Scenario3_TrafficLightActions.java at main · CodingLyf-Fullstack/JavaSamples](#)

176 Java Scenario 4: E-commerce Cart Total Calculation (For loop).

Given a list of cart items with price and quantity, loop through and:

- Calculate total bill
- Apply a 10% discount if total exceeds 5000
- Stop processing further items if total exceeds 10,000 (simulate “max cart limit”)

Hint:

- Loops through each CartItem and adds price × quantity to the total.
- Stops the loop early if total > 10,000.
- Applies a 10% discount if the total is above 5,000

Code link: [JavaSamples/Scenario4_CartBilling.java at main · CodingLyf-Fullstack/JavaSamples](#)

177 Java Scenario 5: Password Strength Checker.

Given a string password, loop through characters and:

- Count uppercase, lowercase, digits, and special characters
- If all categories are present and length ≥ 8 , print “Strong password” else “Weak password”

Hint:

- Loops through each character.
- Uses Character methods for uppercase, lowercase, and digit detection.
- Treats anything else as a special character.
- Strength check: at least one from each category and length ≥ 8 .

Code link: [JavaSamples/Scenario5_PasswordChecker.java at main · CodingLyf-Fullstack/JavaSamples](#)

178 Java Scenario 6: Printing Even and Odd Numbers with Two Threads

Two threads should print numbers from 1 to 20. One prints even numbers, the other prints odd numbers.

Hint:

- Use synchronization to ensure correct order

Code link: [JavaSamples/Scenario6_OddEvenPrinter.java at main · CodingLyf-Fullstack/JavaSamples](#)

Explanation:

1. Two threads start with different responsibilities

- Odd thread calls `printOdd(1, 3, 5...)`
- Even thread calls `printEven(2, 4, 6...)`
- Both share the same `Printer` object, so synchronization works on its monitor.

2. The `isOdd` flag controls whose turn it is

- Initially `isOdd` is false (meaning it is Odd thread turn).
- Odd thread should run first, Even thread must wait until `isOdd` becomes true.

3. Even thread hits `printEven()` and blocks

`while (!isOdd) { wait(); }`

- Since `isOdd` is false initially, this condition is true.
- `wait()` is called, which releases the lock on `Printer` and puts Even thread into *waiting state*.
- It stays parked until someone calls `notify()` on the same monitor.

4. Odd thread gets the lock and runs `printOdd()`

`while (isOdd) { wait(); }`

- Here, `isOdd` is false, so it skips waiting.
- Prints odd number, sets `isOdd = true`, calls `notify()` to wake up the Even thread.

5. Key point about `wait()` here

- Odd waits if it is not its turn (`isOdd == true`).
- Even waits if it is not its turn (`isOdd == false`).
- `wait()` is the mechanism that *releases the lock and pauses the thread until notify() signals that it is safe to proceed*.

Reference: [Print Even and Odd Numbers Using 2 Threads | Baeldung](#)

179 Java Scenario 7: Sequential Execution of Two Threads Printing Numbers (1,2,3...) and Letters(A,B,C,..) in Java. So that it prints (A,1B,2...)**Hint:**

- Use synchronization to ensure correct order

Code link: [JavaSamples/Scenario7_AplhaNumericSequence.java at main · CodingLyf-Fullstack/JavaSamples](https://github.com/CodingLyf-Fullstack/JavaSamples/blob/main/src/main/java/com/codinglyf/threading/Scenario7_AplhaNumericSequence.java)**Explanation:****Check the code, it contains all necessary explanation in the comments.**

- We create two threads; one for letters, one for numbers.
- we create a shared lock “monitor” .
- We also keep a flag called isLetter. If it is false, it is the letter thread’s turn. If it is true, it is the number thread’s turn.
- Each thread checks this flag in a while loop. If it is not their turn, they go to sleep using monitor.wait() until the other thread wakes them up.
- When a thread does print its value, it flips the flag to give the turn to the other thread, then calls monitor.notify() to wake them up.
- We start with isLetter = false, so the letter thread goes first, prints “A”, and hands the turn to the number thread.
- This back-and-forth continues until we’ve gone through all 26 letters and numbers in sequence.

180 Java Scenario 8: Producer Consumer Problem

One thread produces integers into a shared queue, another consumes them. Implement using wait() and notify() for thread coordination.

Solution:

This can be done in two ways

1. Using Synchronized and inter process communication(wait() and notify())
2. Using BlockingQueue

Explanation:**a. Synchronized(wait() and notify()):**

1. Create two threads t1 and t2, t1 is produce the value and t2 is to consume the value.

2. Create a ShareQueue class, In which we take a Queue or LinkedList. We need to know how much capacity this Queue should be.
3. SharedQueue class contains two synchronized methods produce() and consume(). These methods are invoked by the two threads.
4. If Queue is full, make producer to wait, until consumer consumes the queue.
5. If Queue is empty, make consumer to wait, until producer pushes the values to the queue.
6. In this sample, we are producing 20 values and consuming till queue is empty. Once queue is empty thread stops execution.
7. Here is the code link:

b. Using BlockingQueue

A BlockingQueue is just a queue with built-in thread safety and blocking behavior. It is in `java.util.concurrent` and is mostly used when we have producer-consumer kind of situations.

How it works?

When we try to put something in and the queue is full -> the thread will wait until space is available. When we try to take something out and the queue is empty -> the thread will wait until there's something to take.

This removes the headache of manually handling `wait()` and `notify()`; the queue does it for us.

Blocking Operations:

- `put(E e)`: Inserts the specified element into this queue, waiting if necessary for space to become available.
- `take()`: Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

Implementations:

- `ArrayBlockingQueue`: A bounded blocking queue backed by an array.
- `LinkedBlockingQueue`: An optionally bounded blocking queue backed by a linked list.
- `PriorityBlockingQueue`: An unbounded blocking queue that orders elements according to their natural ordering or a custom Comparator.

Reference for producer consumer problem: [Java BlockingQueue Example | DigitalOcean](#)

Code link: [JavaSamples/Scenario8_ProducerConsumer.java at main · CodingLyf-Fullstack/JavaSamples](#)

181 Java Scenario 9: Simulating the deadlock

Create two threads, each trying to lock two resources in opposite order, causing a deadlock. Then show how to avoid it.

Code links:

Deadlock Example: [JavaSamples/Scenario9_Deadlock.java at main · CodingLyf-Fullstack/JavaSamples](#)

Explanation: In this example, thread1 acquires resource1 and then tries to acquire resource2.

Simultaneously, thread2 acquires resource2 and then tries to acquire resource1.

This creates a circular dependency where thread1 holds resource1 and waits for resource2 (held by thread2), and thread2 holds resource2 and waits for resource1 (held by thread1), leading to a deadlock

Deadlock Prevention Example: [JavaSamples/Scenario9_DeadPrevention.java at main · CodingLyf-Fullstack/JavaSamples](#)

In the corrected example, both thread1 and thread2 acquire resource1 before attempting to acquire resource2.

This consistent ordering prevents the circular wait condition, thereby avoiding deadlock.

If thread1 acquires resource1 first, thread2 will wait for resource1 to be released. Once thread1 releases both locks, thread2 can then acquire them in the correct order and proceed.

182 Java Scenario 10: Increment the shared counter with threads

Create 10 threads, each updating a shared counter. The goal is for the final count to be 10,000.

Solution: Use AtomicInteger for thread-safe increments.

We'll look at two examples:

1. Using a normal int (shows an incorrect result due to race conditions).
2. Using AtomicInteger (produces the correct result).

Code links:

With Integer: [JavaSamples/Scenario10_SharedCounter_Int.java at main · CodingLyf-Fullstack/JavaSamples](#)

AtomicInteger: [JavaSamples/Scenario10_SharedCounter_Atomic.java at main · CodingLyf-Fullstack/JavaSamples](#)

183 Java Scenario 11: Bank Account Withdrawal

Multiple threads withdraw from the same account. How do you make sure to avoid race conditions. What techniques you would use?

Solution: We should use synchronized blocks. Here is the complete sample.

Code link: [JavaSamples/Scenario11_BankAccount.java at main · CodingLyf-Fullstack/JavaSamples](#)

184 Java Scenario 12: CompletableFuture Scenario

Imagine you are calling two external services:

1. Fetch user details from Service A.
2. Fetch user's recent orders from Service B.

We don't want to wait for one to finish before starting the other. How will you implement it?

Solution:

Using CompletableFuture. It allows us to run tasks asynchronously and combine results when ready. It is Great for parallel tasks like API calls, DB queries, or expensive computations.

```
public class CompletableFutureExample {  
  
    public static void main(String[] args) throws ExecutionException,  
InterruptedException {  
  
        // Async task 1  
        CompletableFuture<String> userFuture = CompletableFuture.supplyAsync(() -> {  
            simulateDelay(1000);  
            return "User: CodingLyf";  
        });  
  
        // Async task 2  
        CompletableFuture<String> ordersFuture = CompletableFuture.supplyAsync(() -> {  
            simulateDelay(1500);  
            return "Orders: 5";  
        });  
  
        // Combine both results  
        CompletableFuture<String> combinedFuture = userFuture.thenCombine(ordersFuture,  
(user, orders) -> {  
            return user + " | " + orders;  
        });  
    }  
}
```

```

});  

System.out.println(combinedFuture.get()); // Waits for both tasks to complete  

}  

private static void simulateDelay(int millis) {  

    try { Thread.sleep(millis); } catch (InterruptedException e) {  

        e.printStackTrace();  

    }
}

```

CompletableFuture.supplyAsync() runs a task in a separate thread, thenCombine() helps to combine the results of two independent futures, and the execution remains non-blocking until we explicitly call .get() or .join().

185 Java Scenario 13: Payment Gateway Integration

You need to build a payment API that must support multiple payment methods (UPI, Card, Wallet) and allow third parties to add new methods in the future.

What would you choose? Interface, Abstract Class, or Static Method?

Solution:

Static methods won't work here because the code is fixed. We will have to change it every time we add a new payment method, which makes it hard to grow in the future.

Abstract classes can give some default behavior, but they only allow inherit from one class. That can limit third parties if they already have another base class.

Interfaces are the best option. We can add new payment methods without touching the existing code.

```

//Payment method contract
interface PaymentMethod {  

    void pay(double amount);  

    String getName();  

}  

//Existing implementations
class Upipayment implements PaymentMethod {  

    //Implement the methods  

}  

class CardPayment implements PaymentMethod {  

    //Implement the methods  

}

```

Code link: [JavaSamples/Scenario13_PaymentIntegration.java at main · CodingLyf-Fullstack/JavaSamples](#)

186 Java Scenario 14: Notification Service

Different channels like Email, SMS, Push Notification must follow the same contract (sendMessage), but each channel is completely independent.

The app needs to send notifications through Email, SMS, and Push Notifications.

- Create a Notifier interface with sendNotification(String message, String to).
- Implement each type separately.
- Show how dependency injection can let you swap implementations.

Solution:

Interfaces are the best choice because they let us add new payment methods without changing any existing code, keeping the design flexible and future-proof.

```
interface Notifier {
    void sendNotification(String message, String to);
}

class EmailNotifier implements Notifier {
    @Override
    public void sendNotification(String message, String to) {
        System.out.println("Sending EMAIL to " + to + ": " + message);
    }
}

class SmsNotifier implements Notifier {
    @Override
    public void sendNotification(String message, String to) {
        System.out.println("Sending SMS to " + to + ": " + message);
    }
}
```

Code link: [JavaSamples/Scenario14_NotificationService.java at main · CodingLyf-Fullstack/JavaSamples](#)

187 Java Scenario 15: File Export System

You need to build a reporting system that can export data to PDF, Excel, or CSV.

- Create an Exporter interface with export(List<String> data, String fileName).
- Implement the interface for each format and allow users to choose export type at runtime.

Hint: Use Interfaces

188 Java Scenario 16: Authentication Strategies

Your app should allow Username/Password login, OAuth login, and Biometric login.

- Create an interface `Authenticator` with boolean authenticate(String user, String credential).

- Implement separate classes for each authentication type.

Hint: Use Interfaces.

189 Java Scenario 17: Employee Payroll System

All employees have name, salary, and a method calculateBonus(), but bonus calculation differs for FullTimeEmployee and ContractEmployee..

How do you design this, Using interface or abstracts class?

Solution:

An abstract class fits better here because:

- All employees have common fields (name, salary) ; abstract classes can store these, interfaces can't.
- We can keep shared code in one place and only make subclasses implement the different part (calculateBonus()).
- calculateBonus() can be an abstract method so each type of employee defines its own formula.

In short: use an abstract class when all types share fields and some common methods, but each has its own version of part of the logic.

```
//Abstract class with common fields and behavior
abstract class Employee {
    protected String name;
    protected double salary;

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    // Abstract method: subclasses must implement their own bonus logic
    public abstract double calculateBonus();

    public void displayInfo() {
        System.out.println("Name: " + name + ", Salary: " + salary);
    }
}
```

Code link: [JavaSamples/Scenario17_EmployeePayroll.java at main · CodingLyf-Fullstack/JavaSamples](#)

190 Java Scenario 18: Bank Account Types

In a banking app, savings and current accounts share logic like deposit() and calculateInterest() but have different interest rules.

Hint:

- Create an abstract class BankAccount with deposit() and abstract calculateInterest() method.
- Implement for SavingsAccount and CurrentAccount.

191 Java Scenario 19: Custom Exceptions

When creating a custom exception, how do you decide whether to extend Exception or RuntimeException?

Explanation:

When creating a custom exception in Java, the choice between extending Exception or RuntimeException depends on whether we want to create a checked exception or an unchecked exception.

1. Extend Exception for Checked Exceptions:

If the exception represents a condition that a client (calling code) can handle, we should create a checked exception.

Example 1: InsufficientBalanceException

Imagine a bank withdrawal method. If someone tries to take out more than they have, we throw this exception. The ATM or mobile app can catch it and display, “Not enough funds.” The program continues running happily.

Example 2: FileFormatException

The program only accepts .csv files for importing data. If someone uploads a .pdf, we throw this. The caller can say, “Wrong file format. Please upload a CSV.”

Example 3: OrderNotFoundException

In an e-commerce site, a customer checks the status of an order ID that doesn’t exist. We throw this so the UI can show “Order not found” instead of a system crash.

2. Extend RuntimeException for Unchecked Exceptions:

If the exception represents a programming error, a fundamental flaw in the application logic, or a condition that cannot be recovered by the client, we should create an unchecked exception.

Example 1: PaymentProcessingFailedException

The payment service has a serious error, maybe wrong credentials or the provider is down. The app can’t fix this during runtime.

Example 2: DatabaseConnectionFailedException

The app can't connect to the database at all. Without that, nothing works, so it should fail immediately.

192 Java Scenario 20: Closing resources

How do you make resources like Inputstream or files or DB connections close after you done with your work?

Explanation:

In Java, We need to use **try-with-resources** to close the resources like InputStream, Reader, or File handles.

```
import java.io.*;

public class FileReadExample {
    public static void main(String[] args) {
        // try-with-resources automatically closes the stream
        try (InputStream input = new FileInputStream("data.txt")) {
            int data;
            while ((data = input.read()) != -1) {
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

How this works?

- Any class that implements **AutoCloseable** (like InputStream, OutputStream, Reader, Writer) will be closed automatically when the try block ends , even if an exception happens.
- This removes the need for a finally block.

Reference: [Java's AutoCloseable Interface Explained | Medium](#)

193 Java Scenario 21: Creating Custom Exception

You have an Interface EmailService with sendEmail(String to, String subject, String body) that throws InvalidEmailException or EmailServerDownException.

Implementations are SMTPEmailService and SendGridEmailService.

How do you design the custom exceptions?

Explanation:

We should create 2 Exception classes that extends Exception.

```
//Checked exception for invalid email addresses
class InvalidEmailException extends Exception {
    public InvalidEmailException(String message) {
        super(message);
    }
}

//Checked exception for server issues
class EmailServerDownException extends Exception {
    public EmailServerDownException(String message) {
        super(message);
    }
}
```

Code link: [JavaSamples/Scenario21_CustomException.java at main · CodingLyf-Fullstack/JavaSamples](#)

194 Java Scenario 22: Global Math Operations

You need add, subtract, multiply, divide methods that same for the whole application and how do you design these methods?

Explanation:

We should create them as static methods as they won't depend on any object state.

```
public final class MathUtils {

    // Private constructor prevents instantiation
    private MathUtils() {
    }

    public static int add(int a, int b) {
        return a + b;
    }

    public static int subtract(int a, int b) {
        return a - b;
    }

    public static int multiply(int a, int b) {
        return a * b;
    }

    public static double divide(double a, double b) {
        if (b == 0) {
            throw new ArithmeticException("Division by zero");
        }
    }
}
```

```

        return a / b;
    }
}

```

195 Java Scenario 23: Logging

Your system needs a log(message) function that can be called from anywhere without creating a logger object?

Hint:

Use Static.

196 Java Scenario 24: Creating Custom Functional Interface

How do you create and use a functional interface? Write a simple interface (Calculator) and demonstrate how to implement it using a lambda expression.

Explanation:

We should create them as static methods as they won't depend on any object state.

```

@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}

public class Test {
    public static void main(String[] args) {
        Calculator add = (a, b) -> a + b;
        Calculator multiply = (a, b) -> a * b;

        System.out.println(add.calculate(5, 3));      // 8
        System.out.println(multiply.calculate(5, 3)); // 15
    }
}

```

197 Java Scenario 25: Collections - 1

You have a list of recently visited URLs. The order of visits matters, but a URL should only appear once. Which collection will you use and why?

Solution: Use a **LinkedHashSet** ; it preserves insertion order and prevents duplicates.

198 Java Scenario 26: Collections - 2

You are building a leaderboard where player names are keys and scores are values. You need constant-time lookup by name and to iterate over them in sorted order of names. Which collection type works here?

Solution: Use a **TreeMap<String, Integer>**; it keeps keys sorted in natural or custom order while allowing O(log n) operations for insert, lookup, and remove.

199 Java Scenario 27: Collections - 3

You have a **List<Integer>** with 1 million numbers. You need to remove all numbers less than 100 while iterating.

What's the correct way to do this without hitting **ConcurrentModificationException**?

Solution: Use an **Iterator** and its **remove()** method:

```
Iterator<Integer> it = list.iterator();
while (it.hasNext()) {
    if (it.next() < 100) {
        it.remove();
    }
}
```

This avoids **ConcurrentModificationException** because **remove()** updates the iterator's internal state safely.

200 Java Scenario 28: Collections - 4

You have a **Map<String, Integer>** with employee names and their performance scores. You need the top 3 performers in descending order of score. How will you achieve this?

Solution: We can achieve this by **Collections.sort** and also we can use Streams.

```
public class Test {
    public static void main(String[] args) {
        Map<String, Integer> scores = new HashMap<>();
        scores.put("Mahesh", 85);
        scores.put("NTR", 95);
        scores.put("Charan", 92);
        scores.put("Prabhas", 88);

        List<Map.Entry<String, Integer>> list = new ArrayList<>(scores.entrySet());

        // Sort by value in descending order
        Collections.sort(list, (e1, e2) -> e2.getValue().compareTo(e1.getValue()));

        for (int i = 0; i < Math.min(3, list.size()); i++) {
            Map.Entry<String, Integer> e = list.get(i);
```

```
        System.out.println(e.getKey() + " -> " + e.getValue());  
    }  
}  
}
```

201 Java Scenario 29: Collections - 5

You have two lists of customer emails. Merge them into one, preserving order of first appearance, without duplicates. Which collection will you use?

Solution: Use a LinkedHashSet, it removes duplicates while keeping the insertion order.
Add all emails from the first list, then from the second list.

202 Java Scenario 30: Collections - 6

You have a string and need to find the first character that appears only once. Which collection will help track occurrence and maintain original order efficiently?

Solution: Use a `LinkedHashMap<Character, Integer>`; it preserves insertion order and allows us to store character counts.

After populating it, iterate over the entries and return the first key with a count of 1.

```
public class Test {
    public static void main(String[] args) {
        String str = "swiss";

        Map<Character, Integer> map = new LinkedHashMap<>();

        // Count occurrences
        for (char c : str.toCharArray()) {
            map.put(c, map.getOrDefault(c, 0) + 1);
        }

        // Find first character with count 1
        for (Map.Entry<Character, Integer> entry : map.entrySet()) {
            if (entry.getValue() == 1) {
                System.out.println("First non-repeating character: " +
entry.getKey());
                return;
            }
        }

        System.out.println("No unique character found");
    }
}
```

Note: We can also use Streams to achieve this

203 Java Scenario 31: Collections - 7

You are building a real-time logging system that stores only the last 100 log entries in memory. Old entries should automatically be removed as new ones are added. Which collection will you choose?

Solution: Use an **ArrayDeque** (or **LinkedList**) and enforce the size manually.

On each insert, if size exceeds 100, remove from the head (**poll()**), then add the new log to the tail (**offer()**).

```
public class Test {
    private static final int MAX_SIZE = 5;
    private Queue<String> logs = new ArrayDeque<>();

    public void addLog(String log) {
        if (logs.size() == MAX_SIZE) {
            logs.poll(); // remove oldest
        }
        logs.offer(log); // add newest
    }

    public void printLogs() {
        logs.forEach(System.out::println);
    }

    public static void main(String[] args) {
        Test logger = new Test();

        for (int i = 1; i <= 10; i++) {
            logger.addLog("Log " + i);
        }

        logger.printLogs(); // prints last 100 logs
    }
}
```

Note: In the sample, we have taken Max Size as 5, when we try to insert 10 items, first 5 will be deleted.

204 Java Scenario 32: Collections – 8

Given a list of product IDs sold in a day, you want to find the product sold the most. Which collection will you use, and what will be your approach?

Solution: Use a **HashMap<Integer, Integer>** to store each product ID and its sale count.

Iterate through the list, update counts with **getOrDefault()**, then scan the map to find the entry with the highest value.

```
public class Main {
    public static void main(String[] args) {
        List<Integer> sales = Arrays.asList(101, 102, 101, 103, 101, 102, 103, 103,
103);
```

```

Map<Integer, Integer> countMap = new HashMap<>();

// Count occurrences
for (int id : sales) {
    countMap.put(id, countMap.getOrDefault(id, 0) + 1);
}

// Find product with max sales
int maxProduct = -1, maxCount = 0;
for (Map.Entry<Integer, Integer> entry : countMap.entrySet()) {
    if (entry.getValue() > maxCount) {
        maxProduct = entry.getKey();
        maxCount = entry.getValue();
    }
}
System.out.println("Most sold product: " + maxProduct + " (" + maxCount + " sales)");
}
}

```

205 Java Scenario 33: LRU Cache Implementation

You are implementing a cache where the most recently used item should stay, and the least recently used should be evicted once capacity is reached. Which Java collection can do this with minimal custom code?

Solution: Use a **LinkedHashMap** with `accessOrder = true`, it maintains entries in the order they were last accessed.

Override `removeEldestEntry()` to automatically evict the least recently used when capacity is exceeded.

Code Link: [JavaSamples/Scenario33_LRUCache_Implementation.java at main · CodingLyf-Fullstack/JavaSamples](#)

206 Java Scenario 34: Task Scheduler by Priority

You need to schedule jobs where each job has a priority number. Highest priority should be executed first, lowest last.

Solution: Use **PriorityQueue**

```

// Max-heap style: highest priority first
PriorityQueue<Job> queue = new PriorityQueue<>(
    (j1, j2) -> Integer.compare(j2.priority, j1.priority)
);

queue.add(new Job("Email Report", 2));
queue.add(new Job("Database Backup", 5));

```

```

queue.add(new Job("UI Refresh", 3));

while (!queue.isEmpty()) {
    Job job = queue.poll();
    System.out.println("Executing: " + job);
}

```

PriorityQueue is normally a min-heap (smallest element first), so we provide a custom comparator to reverse it.

The loop executes jobs from highest priority to lowest.

Code Link: [JavaSamples/Scenario34_TaskScheduler.java at main · CodingLyf-Fullstack/JavaSamples](#)

207 Java Scenario 35: Detecting Duplicate Usernames

Given a stream of usernames during registration, detect immediately if the username is already taken without scanning the entire list. How will you use HashSet here?.

Solution:

Use a **HashSet<String>** to store all registered usernames.

Before adding a new one, check `set.contains(username)`.

This gives **O(1)** average-time lookup without scanning the list.

```

public boolean register(String username) {
    if (usernames.contains(username)) {
        System.out.println("Username already taken: " + username);
        return false;
    }
    usernames.add(username);
    System.out.println("Registered: " + username);
    return true;
}

```

208 Java Scenario 36: Auto-Suggest in Search

You have a collection of product names. As the user types a prefix, show suggestions in sorted order starting from that prefix. Which collection do you use?

Solution:

We can use a **TreeSet<String>** since it stores elements in sorted order and supports range queries using `subSet()`.

```

public class Test {
    public static void main(String[] args) {
        TreeSet<String> products = new TreeSet<>();

```

```

        products.add("apple");
        products.add("apricot");
        products.add("banana");
        products.add("blueberry");
        products.add("blackberry");
        products.add("cherry");

        String prefix = "bl";
        String end = prefix + Character.MAX_VALUE; // ensure all starting with prefix

        NavigableSet<String> suggestions = products.subSet(prefix, true, end, true);
        System.out.println("Suggestions for '" + prefix + "' : " + suggestions);
    }
}

```

How it works:

- TreeSet keeps items in natural sorted order.
- subSet(prefix, true, end, true) efficiently fetches only items starting with that prefix , no full scan needed.

209 Java Scenario 37: Stack – Parentheses Balancer

You have a collection of product names. As the user types a prefix, show suggestions in sorted order starting from that prefix. Which collection do you use?

Solution:

We can use a **Stack<Character>**:

- **Push** opening brackets ((), {}, []) onto the stack using push().
- **Pop** when we encounter a closing bracket and check if it matches the top.

```

public class Main {
    public static void main(String[] args) {

        System.err.println(isBalanced("{}"));
        System.err.println(isBalanced("{}"));
    }

    static boolean isBalanced(String str) {
        Stack<Character> stack = new Stack<>();
        for (Character ch : str.toCharArray()) {
            if (ch == '(' || ch == '{' || ch == '[') {
                stack.push(ch);
            } else if (ch == ')' || ch == '}' || ch == ']') {
                if (stack.isEmpty())
                    return false;
                Character top = stack.pop();
                if (ch == ')' && top != '(' ||
                    ch == '}' && top != '{' ||
                    ch == ']' && top != '[')
                    return false;
            }
        }
        return stack.isEmpty();
    }
}

```

```
        ch == ']' && top != '[') {
            return false;
        }
    }
}
return stack.isEmpty();
}
```

210 Java Scenario 38: Thread safe list

Multiple threads are adding elements to a list at the same time. Which collection will you choose?

Solution:

- Use **CopyOnWriteArrayList**.
 - it allows concurrent reads without locking because it works on a snapshot of the array.
 - Writes (add, remove) create a new copy of the array, so reads never block.

211 Java Scenario 39: Counting API Requests Per User

Multiple threads are recording the API hits for different users in real time. Which collection you would use to keep counts without explicit synchronization?

Solution:

Use ConcurrentHashMap<String, AtomicInteger> so increments are thread-safe without explicit locking.

```
public class ApiRequestCounter {  
    private ConcurrentHashMap<String, AtomicInteger> counts = new ConcurrentHashMap<>();  
  
    public void recordRequest(String user) {  
        counts.computeIfAbsent(user, k -> new AtomicInteger()).incrementAndGet();  
    }  
  
    public int getCount(String user) {  
        return counts.getOrDefault(user, new AtomicInteger()).get();  
    }  
  
    public static void main(String[] args) {  
        ApiRequestCounter counter = new ApiRequestCounter();  
        counter.recordRequest("Dhoni");  
        counter.recordRequest("Dhoni");  
        counter.recordRequest("Virat");  
  
        System.out.println("Dhoni: " + counter.getCount("Dhoni")); // 2
```

```

        System.out.println("Virat: " + counter.getCount("Virat"));    // 1
    }
}

```

212 Java Scenario 40: ArrayList vs LinkedList

You are storing the last 1 million stock price updates in memory.

- Every new price gets added **at the end**.
- Once you reach 1 million, you remove the **oldest price** (front of the list).
- You also frequently check the price at random positions in the list.

Which list implementation would you choose, and why?

Solution:

Use ArrayList for fast random access and for quick insertions at the end.

LinkedList: better for frequent middle insertions/deletions since no shifting of elements, just pointer updates.

ArrayList: better for fast random access and adding at end because it uses an array under the hood

213 Java Scenario 41: Large file handling

How would you read a large file efficiently without running out of memory?

Solution:

When dealing with large files, the key is to read the file in chunks rather than loading the entire file into memory. This can be done efficiently using BufferedReader or FileInputStream in Java. By processing the file line-by-line or in smaller byte chunks.

```

public class LargeFileReader {
    public void readFile(String filePath) {
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
                // Process each line
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

214 Java Scenario 42: Memory Leaks

How would you detect and prevent memory leaks that cause OutOfMemory in a Java application??

Common causes of memory leaks

- Static fields holding onto large objects for the entire life of the program.
- Long-lived collections (like List or Map) that keep growing because items are never removed.
- Listeners, callbacks, or threads that are registered but never unregistered.
- Unclosed resources like streams, sockets, or database connections.

We would need to 2 steps to follow.

1. **Identify the memory leaks**
2. **Fix to avoid the leaks**

To detect memory leaks, I'd use profiling tools like VisualVM, YourKit, or JProfiler. These tools allow me to monitor heap usage and identify objects that are not being garbage collected.

Preventing memory leaks often involves practices like:

- Avoiding static references: Ensure that static fields don't hold onto objects longer than necessary.
- Properly closing resources:
- Use try-with-resources to ensure resources like streams and connections are closed automatically.
- Weak References: Use weak references for cache implementations to allow garbage collection when memory is needed.

Reference: [Java Memory Leaks: Detection & Fix | Medium](#)

215 Java Scenario 43: Optimize the GC

How would you optimize an application to reduce the impact of garbage collection?

To reduce the impact of garbage collection (GC) in an application, follow these steps:

1. **Reduce Object Creation:** Avoid creating unnecessary objects. Reuse objects when possible instead of creating new ones.
2. **Use Primitive Types:** Prefer int, double, etc., over their object versions (Integer, Double) to avoid extra memory usage.
3. **Limit Large Collections:** Clear or reuse large lists, maps, or arrays instead of letting them grow endlessly.

4. **Avoid Memory Leaks:** Remove references to objects (like event listeners or caches) when they're no longer needed.
5. **Tune GC Settings:** Adjust JVM flags (like -Xmx, -Xms) to optimize heap size and GC behavior based on the app's needs.
6. **Use Object Pools:** For frequently created/destroyed objects (like database connections), use pooling (e.g., ArrayList pooling).
7. **Profile & Monitor:** Use tools (like VisualVM) to find memory hotspots and optimize them.

216 Java Scenario 44: Best coding practices

What are the best practices that you follow while developing java application?

1. Write Clean, Readable Code

- Follow consistent naming conventions (camelCase for variables, PascalCase for classes).
- Keep methods small and focused ; one responsibility per method.
- Use meaningful names (instead of calc() write calculateTax()).

2. Follow SOLID Principles

- Single Responsibility: Each class does one thing well.
- Open/Closed: Open for extension, closed for modification.
- Liskov Substitution: Derived classes should be usable as their base type.
- Interface Segregation: Keep interfaces simple.
- Dependency Inversion: Depend on abstractions, not concrete classes.

3. Handle Exceptions Properly

- Don't swallow exceptions silently (catch (Exception e) {} is bad).
- Use custom exceptions where it adds clarity.

4. Optimize Collections Usage

- Choose the right type (ArrayList vs LinkedList, HashMap vs TreeMap).
- Avoid unbounded collections for data that grows indefinitely.
- Prefer Collections.unmodifiableList() when returning read-only data.

5. Manage Resources Safely

- Use try-with-resources for streams, files, sockets, DB connections.
- Close resources in the same scope we open them.

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    // work
}
```

6. Keep Concurrency Safe

- Use concurrent collections (ConcurrentHashMap, CopyOnWriteArrayList) when needed.
- Avoid unnecessary synchronization , it hurts performance.

- Use ExecutorService instead of creating raw threads.

7. Avoid Hardcoding

- Move constants to config files or environment variables.
- Use .properties or .yaml for application settings.

8. Write Unit & Integration Tests

- Test edge cases, not just happy paths.
- Use frameworks like JUnit, Mockito.

9. Memory Usage

- Avoid creating unnecessary objects in loops.
- Reuse immutable objects like String carefully.
- Watch out for memory leaks from static references or listeners.

10. Follow Java Language Features Wisely

- Prefer StringBuilder for concatenation in loops.
- Use Optional to avoid null checks when it makes sense.
- Use record for DTOs (Java 16+).

11. Use Logging Properly

- Prefer a logging framework (SLF4J, Logback) over System.out.println.
- Log only meaningful info.
- Keep sensitive data out of logs.

12. Keep Build & Dependencies Clean

- Avoid unused dependencies.
- Keep dependencies up to date.
- Use dependency management tools like Maven or Gradle consistently.

217 Java Scenario 45: Performance Optimization

What do you optimize the performance of a Java application?

1. **Profiling and Monitoring:** We can use profiling tools like JProfiler to identify performance bottleneck and to monitor the CPU usage, memory consumption, and response times.
2. **Database Optimization:** Data is very important part of every application, so we need to optimize the SQL queries. Optimize queries by adding appropriate indexes, rewriting complex joins, and reducing the number of queries by implementing batch processing.
3. **Code Refactoring:** Optimized loops and algorithms to reduce time complexity. I also removed unnecessary synchronization to improve thread performance (Check above question for coding practices).
4. **Caching:** Implemented caching strategies using tools like Ehcache to reduce the load on the database and improve response times for frequently accessed data.

5. **Garbage Collection Tuning:** Adjusted JVM garbage collection settings to reduce pause times and improve memory management.

218 Java Scenario 46: Implementing the Retry

How do you implement retry using Java.

Explanation:

We can implement retry using

1. Thread – A simple retry
2. Callable
3. Custom Interface

1. Thread – A simple retry

This code tries to run performOperation() up to 3 times, waiting 1 second between each retry if it fails.

It simulates random failure with a 70% chance, and stops early if the operation succeeds.

If all retries fail, it prints "Retries exhausted".

Code Link: [JavaSamples/Scenario46_SimpleRetry.java at main · CodingLyf-Fullstack/JavaSamples](#)

2. Callable

This code defines a generic executeWithRetry method that runs a Callable and retries it up to a given number of times with a delay between attempts.

It stops and returns immediately if the operation succeeds, otherwise it keeps retrying until the max retries are reached.

If all attempts fail, it throws the last caught exception to the caller.

Code Link: [JavaSamples/Scenario46_Retry_Callable.java at main · CodingLyf-Fullstack/JavaSamples](#)

3. Custom Interface

[Java — Retry Pattern. Ah, the joys of development! Dealing... | by Lucas Amado | Medium](#)

219 Java Scenario 47: Implementing stack using arrays

How do you implement stack using arrays in java.

Explanation:

A stack is a fundamental data structure in computer science that follows the last-in, first-out (LIFO) principle. Imagine a stack of plates: the last plate we add is the first one we can remove. In programming, stacks are used to manage function calls, track undo/redo actions, and more.

Key Concepts:

- LIFO (Last-In, First-Out): The last element added to the stack is the first one to be removed.
- Push: Adds an element to the top of the stack.
- Pop: Removes and returns the top element from the stack.
- Peek: Returns the top element without removing it.
- isEmpty: Checks if the stack is empty.
- Size: Returns the number of elements in the stack.

Code Link: [JavaSamples/Scenario47_StackWithArray.java at main · CodingLyf-Fullstack/JavaSamples](#)

220 Java Scenario 48: Implementing stack using 2 Queues

How do you implement stack using queues in java.

Explanation:

In Java, a Queue is an interface within the Java Collections Framework that represents a collection designed for holding elements prior to processing, typically in a First-In, First-Out (FIFO) manner. This means the element added first to the queue is the first one to be removed.

Key Characteristics:

- **FIFO Ordering:** Elements are processed in the order they were inserted.
- **No Random Access:** Unlike lists, elements cannot be accessed directly by index.
- **Implementations:** Common implementations include LinkedList, PriorityQueue, ArrayDeque, and ArrayBlockingQueue.

Code Link: [JavaSamples/Scenario48_Stack_Using_Queues.java at main · CodingLyf-Fullstack/JavaSamples](#)

Code Explanation:

q1 (Main Queue): This queue always stores the elements in the order required for stack operations (LIFO). The front of q1 will always contain the element that should be popped next.

q2 (Helper Queue): This queue is used temporarily during the push operation to maintain the correct order in q1.

push(int x):

The new element x is first added to q2.

All elements from q1 are then moved to q2. This ensures that the new element x is now at the front of q2, followed by all previous elements in their original order.

Finally, the references of q1 and q2 are swapped. This effectively makes q2 (which now holds the correctly ordered elements) the new q1, and the old q1 becomes the new q2 (which is now empty).

pop(): Since q1 always has the most recently added element at its front, a simple poll() operation on q1 removes and returns the top element.

peek(): Similarly, peek() on q1 returns the top element without removing it.

isEmpty() and **size()**: These operations directly leverage the corresponding methods of q1.

Code Steps:

Initial state: q1: [] q2: []

Step 1: push(A):

- q2.offer(A); // q2: [A]
- while (!q1.isEmpty()) { ... } // skipped, q1 is empty
- swap q1 and q2
- After swap: q1: [A] q2: []

Step 2: push(B):

- q2.offer(B); // q2: [B]
- while (!q1.isEmpty()) {
 - q2.offer(q1.poll()); // move A from q1 -> q2
}
- Now q2: [B, A], q1: []
- swap q1 and q2
- After swap: q1: [B, A] q2: []

Final view: Top of stack -> B A

221 Java Scenario 49: Implement Queue using Array

A simple circular array-based queue.

In a **circular array queue**, the idea is that the array behaves like a circle rather than a straight line. When we reach the end of the array, we wrap around to the beginning instead of stopping. This allows us to **reuse empty spaces** at the front of the array once elements are dequeued.

How it works in code:

rear = (rear + 1) % capacity;

1. rear + 1 moves the rear pointer to the next position in the array.
2. % capacity ensures that if rear reaches the last index (capacity - 1), it comes back to 0.

Code link: [JavaSamples/Scenario49_ArrayQueue.java at main · CodingLyf-Fullstack/JavaSamples](#)

Explanation is there in the code itself.

222 Java Scenario 50: Implement Custom HashMap

Code Link: [JavaSamples/Scenario50_CustomHashMap.java at main · CodingLyf-Fullstack/JavaSamples](#)

Explanation there itself in the code: Check the logs to understand it better

Streams – Programming and Scenarios

223 Streams 1: Remove duplicates without distinct()

```
public class StreamsSample {
    public static void main(String[] args) {
        List<Integer> duplicateNumbers = Arrays.asList(1, 2, 3, 2, 4, 3, 5, 1);

        HashSet<Integer> seen = new HashSet<>();
        List<Integer> uniqueNumbers =
duplicateNumbers.stream().filter(seen::add).collect(Collectors.toList());
        System.out.println("Unique Numbers: " + uniqueNumbers);

    }
}
```

Explanation:

1. filter(seen::add) means: for every number, try adding it into the HashSet seen.
2. HashSet.add(x) returns **true** the first time a number is inserted, and **false** if that number was already in the set.
3. Because filter keeps only the elements where the condition is **true**, the stream passes the first time each number appears and drops later duplicates.

224 Streams 2: Get duplicates

```
public class StreamsSample {
    public static void main(String[] args) {
        List<Integer> duplicateNumbers = Arrays.asList(1, 2, 3, 2, 4, 3, 5, 1);

        HashSet<Integer> seen = new HashSet<>();
        List<Integer> uniqueNumbers = duplicateNumbers
            .stream()
            .filter(i -> !seen.add(i))
```

```

        .collect(Collectors.toList());
System.out.println("Unique Numbers: " + uniqueNumbers);

    }
}

```

Explanation:

- `filter(i -> !seen.add(i))` means: for every number, try adding it into the HashSet `seen`.
- `HashSet.add(x)` returns **true** the first time a number is inserted, and **false** if that number was already in the set.
- Because we use `!seen.add(i)`, the filter keeps only the elements where `add()` returned **false**. That means only duplicates pass through the streams, and first occurrences are dropped.

225 Streams 3: Sort in descending order

```

public class StreamsSample {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(11, 11, 1, 3, 5, 6, 5);
        List<Integer> result = list
            .stream()
            .distinct()
            .sorted(Comparator.reverseOrder()).toList();
        System.out.println("Result: " + result);
    }
}

```

Explanation:

- `distinct()` removes duplicates, leaving only unique numbers.
- `sorted(Comparator.reverseOrder())` then sorts those unique numbers in **descending order** (largest -> smallest).
- By default, `sorted()` uses **natural order** (ascending). When we pass `Comparator.reverseOrder()`, it flips that natural order.
- Behind the scenes, the stream collects elements, then applies the comparator to arrange them before building the final list.

For the input [11, 11, 1, 3, 5, 6, 5]:

After `distinct()` -> [11, 1, 3, 5, 6]

After `sorted(reverseOrder())` -> [11, 6, 5, 3, 1]

226 Streams 4: Practice Questions on Sort

1. Given a list of strings, sort them according to increasing order of their length?

```
List<String> listOfStrings = Arrays.asList("Java", "Python", "C#", "HTML", "Kotlin", "C++", "COBOL", "C");
```

Use: .sorted(Comparator.comparing(String::length))

2. How do you sort the given list of decimals in reverse order?

```
List<Double> decimalList = Arrays.asList(12.45, 23.58, 17.13, 42.89, 33.78, 71.85, 56.98, 21.12);
```

227 Streams 5: Find Max Number in a list

```
public class StreamsSample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 4, 41, 4);
        int maxNumber = numbers.stream()
            .max(Comparator.naturalOrder())
            .orElse(0);
        System.out.println("Max Number: " + maxNumber);
    }
}
```

Explanation:

1. numbers.stream() : creates a stream from the list [1, 2, 4, 41, 4].
2. .max(Comparator.naturalOrder()) : finds the maximum element by comparing numbers in their **natural order** (ascending).
3. .orElse(0) : in case the stream is empty, it safely returns 0 instead of throwing an exception.

Another Variant:

Given a list of strings, find the longest string using Java streams.

```
List<String> strings = Arrays.asList("apple", "banana", "orange", "grape", "kiwi");
```

Use: .max((o1, o2) -> o1.length() - o2.length())

228 Streams 6: Check all numbers even or not

```
public class StreamsSample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(2, 4, 6, 8, 10);
        boolean allEven = numbers
            .stream()
            .allMatch(i -> i % 2 == 0);
        System.out.println("Result: " + allEven);
    }
}
```

Explanation:

1. `allMatch(i -> i % 2 == 0)` -> checks if **every element** in the stream satisfies the condition (even in this case).
2. Since all numbers are even, it returns true.

Other flavours

1. `anyMatch`

Returns true if **at least one element** satisfies the condition.

```
boolean hasEven = numbers.stream().anyMatch(i -> i % 2 == 0);
// true, because even numbers exist
```

2. `noneMatch`

Returns true if **no elements** satisfy the condition.

```
boolean noOdd = numbers.stream().noneMatch(i -> i % 2 != 0);
// true, because there are no odd numbers
```

Similar Question:

1. Check if Any String Starts with 'A'

```
List<String> nameList = Arrays.asList("Banana", "Apple", "Cat", "Andrew");
```

Use - `anyMatch`

229 Streams 7: Numbers starting with 1

```
public class StreamsSample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(10, 12, 20, null, 19, 30);
        List<Integer> startWith1 = numbers
            .stream()
            .filter(i -> String.valueOf(i).startsWith("1"))
            .toList();
        System.out.println(startWith1);
    }
}
```

Explanation:

1. `numbers.stream()` -> creates a stream of [10, 12, 20, null, 19, 30].
2. `.filter(i -> String.valueOf(i).startsWith("1"))`
 - Converts each element into a string.
 - Keeps only those that start with "1".
 - Caution: when i is null, `String.valueOf(null)` returns the literal string "null", so it won't throw an exception (but it also won't match).
3. `.toList()` -> collects results into a list.

230 Streams 8: Find Palindrome Strings

```
public class StreamsSample {
    public static void main(String[] args) {
        List<String> palindromeNames = Arrays.asList("Telugu", "Tamil",
"Malayalam");
        List<String> findPalindromeStrings = palindromeNames
            .stream()
            .filter(s -> {
                return s.toLowerCase()
                    .contentEquals(new
StringBuilder(s.toLowerCase()).reverse());
            }).toList();
        System.out.println(findPalindromeStrings);
    }
}
```

Explanation:

1. numbers.stream() -> creates a stream of [10, 12, 20, null, 19, 30].
- 2 .filter(i -> String.valueOf(i).startsWith("1"))
 - Converts each element into a string.
 - Keeps only those that start with "1".
 - Caution: when i is null, String.valueOf(null) returns the literal string "null", so it won't throw an exception (but it also won't match).
- 3.toList() -> collects results into a list.

231 Streams 9: Find the longest word in a list

```
List<String> words = Arrays.asList("cat", "elephant", "dog", "giraffe", "zebra");

Optional<String> longestWord = words.stream().max(Comparator.comparingInt(String::length));

longestWord.ifPresent(word -> System.out.println("The longest word is: " + word));
```

232 Streams 10: List of the questions on filter (For Practice)

```
List<Integer> list = Arrays.asList(10,15,8,49,25,98,32);
```

1. Find out all the even numbers that exist in the list using Stream functions

Hint: Use .filter

2. Find out the FIRST even numbers that exist in the list using Stream functions

Hint: Use .filter and .findFirst

3. Java 8 program to perform cube on list elements and filter numbers greater than 50.

Hint: Use .map and .filter

233 Streams 11: Merge two unsorted arrays into single sorted array

```
public class StreamsSample {
    public static void main(String[] args) {
        int[] a = new int[] { 4, 2, 7, 1 };
        int[] b = new int[] { 8, 3, 9, 5 };
        int[] c = Stream.concat(
            Arrays.stream(a).boxed(), Arrays.stream(b).boxed())
            .sorted()
            .mapToInt(i -> i)
            .toArray();
        System.out.println(Arrays.toString(c));
    }
}
```

Explanation:

- Stream.concat(...) -> merges two streams (a and b) into one sequence.
- .boxed() -> converts int primitives into Integer objects so they can be handled in a regular Stream.
- .mapToInt(i -> i) -> converts back from Stream<Integer> to IntStream for the final array.

Variants can be asked:

- Distinct merge: First remove duplicates , hint: .distinct().sorted().toArray()
- Reverse order: .boxed().sorted(Comparator.reverseOrder()).mapToInt(i -> i).toArray()
- Find max after merge: .max().orElse(-1)

234 Streams 11: Get top 3 elements from the list

```
public class StreamsSample {
    public static void main(String[] args) {
        List<Integer> listOfIntegers = Arrays.asList(71, 18, 42, 21, 67, 32, 95,
14, 56, 87);

        List<Integer> topThree = listOfIntegers
            .stream()
            .sorted(Comparator.reverseOrder())
            .limit(3)
            .toList();
        System.out.println(topThree);
    }
}
```

```
}
```

Explanation:

sorted(Comparator.reverseOrder()) : arranges the numbers in descending order (largest to smallest).
limit(3) : takes only the first 3 elements from that sorted stream (the top three numbers).

235 Streams 12: Get 3rd highest element from the list

```
public class StreamsSample {
    public static void main(String[] args) {
        List<Integer> listOfIntegers = Arrays.asList(71, 18, 42, 21, 67, 32, 95,
14, 56, 87);

        Integer thirdHighest = listOfIntegers
            .stream()
            .sorted(Comparator.reverseOrder())
            .skip(2)
            .findFirst().orElse(-1);
        System.out.println(thirdHighest);
    }
}
```

Explanation:

skip(2): ignores the first 2 elements in the sorted stream (so it moves past the highest and second highest).

findFirst(): picks the very next element after skipping, which is the third highest here.

orElse(-1): if no element is found (e.g., empty list), it safely returns -1 instead of failing.

236 Streams 13: Check if two strings are anagrams or not

```
public class StreamsSample {
    public static void main(String[] args) {
        String s1 = "RaceCar";
        String s2 = "CarRace";

        s1 = Stream.of(s1.split(""))
            .map(String::toUpperCase)
            .sorted()
            .collect(Collectors.joining());

        s2 = Stream.of(s2.split(""))
            .map(String::toUpperCase)
            .sorted()
            .collect(Collectors.joining());

        if (s1.equals(s2)) {
            System.out.println("Two strings are anagrams");
        } else {
            System.out.println("Two strings are not anagrams");
        }
    }
}
```

An anagram is when two words or phrases use the same letters with the same frequency, just in a different order.

Example: "listen" and "silent" are anagrams.

Explanation:

- `s1.split("")` -> `map(String::toUpperCase)` -> `sorted()` breaks the string into characters, makes them uppercase, and sorts them alphabetically.
- `.collect(Collectors.joining())` puts the sorted characters back into a single string.
- If both sorted strings are equal, then they're anagrams (same letters, same count, just rearranged).

237 Streams 14: Sum of all digits in a number

```
public class StreamsSample {
    public static void main(String[] args) {
        int i = 15623;

        Integer sumOfDigits = Arrays.stream(String.valueOf(i).split(""))
            .collect(Collectors
                .summingInt(Integer::parseInt));

        System.out.println(sumOfDigits);
    }
}
```

238 Streams 15: Common elements between two arrays

```
public class StreamsSample {
    public static void main(String[] args) {
        List<Integer> list1 = Arrays.asList(71, 21, 34, 89, 56, 28);

        List<Integer> list2 = Arrays.asList(12, 56, 17, 21, 94, 34);

        list1.stream().filter(list2::contains).forEach(System.out::println);
    }
}
```

239 Streams 16: Reverse each word of a string

```
public class StreamsSample {
    public static void main(String[] args) {
        String str = "Java Concept Of The Day";

        String reversed = Arrays.stream(str.split(" "))

```

```

        .map(s -> new StringBuilder(s).reverse())
        .collect(Collectors.joining(" "));

    System.out.println(reversed);
}
}

```

Explanation:

.map(s -> new StringBuilder(s).reverse()): reverses each word

.collect(Collectors.joining(" ")): joins them back with spaces.

240 Streams 17: Count the number of occurrences of a given String.

```

public class StreamsSample {
    public static void main(String[] args) {
        List<String> strings = Arrays.asList("java scala ruby", "java react spring
java");
        String word = "java";
        long count = strings.stream()
            .flatMap(s -> Arrays.stream(s.split(" ")))
            .peek(System.out::println)
            .filter(w -> w.equals(word))
            .count();
        System.out.println("Occurrences of \\" + word + "\": " + count);
    }
}

```

Explanation:

flatMap(s -> Arrays.stream(s.split(" ")))

- Each element in strings is a sentence like "java scala ruby".
- s.split(" ") breaks it into words : ["java", "scala", "ruby"].
- Arrays.stream(...) turns that array into a stream of words.
- **flatMap** flattens all those small word-streams into one continuous stream of words across all sentences.
 - So instead of [[{"java", "scala", "ruby"}, {"java", "react", "spring", "java"}]]
 - We get a single stream: [{"java", "scala", "ruby", "java", "react", "spring", "java"}].

241 Streams 18: Convert the list of sentences into unique words.

```

public class StreamsSample {
    public static void main(String[] args) {
        List<String> sentences = List.of("java is cool", "cool code in java");
        Set<String> words = sentences.stream()
            .flatMap(s -> Arrays.stream(s.split(" ")))
            .collect(Collectors.toSet());
    }
}

```

```

        System.out.println(words);
    }
}

```

242 Streams 19: More Questions on flatMap.

1. You have `List<List<Integer>>`. How do you create a single `List<Integer>`?

Input: `List<List<Integer>> nums = List.of(List.of(1, 2), List.of(3, 4));`

Output: [1,2,3,4]

2. How do you get distinct characters from a list of words?

Input: `List<String> words = List.of("java", "scala");`

Output: [j,a,v,s,c,l]

243 Streams 20: Find all the Longest words in a list

```

public class StreamsSample {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "orange",
"pineapple", "blueberry");

        int maxLength = words.stream()
            .max((o1, o2) -> o1.length() - o2.length())
            .get()
            .length();

        List<String> list = words
            .stream()
            .filter(s -> s.length() == maxLength)
            .toList();

        System.out.println("Longest Strings " + list);
    }
}

```

Output:

Longest Strings [pineapple, blueberry]

244 Streams 21: Questions on reduce

1. Sum of numbers `List<Integer> numbers = Arrays.asList(5, 10, 15, 20);`

2. Find Maximum using reduce, `List<Integer> numbers = Arrays.asList(7, 2, 10, 4);`

3. `List<String> words = Arrays.asList("cat", "elephant", "dog", "tiger");`

245 Streams 22: Find the longest common prefix using Java streams:

```
public class StreamsSample {
    public static void main(String[] args) {

        List<String> strings = Arrays.asList("flower", "flow", "flight");
        String longestCommonPrefix = strings.stream().reduce((s1, s2) -> {
            int length = Math.min(s1.length(), s2.length());
            int i = 0;
            while (i < length && s1.charAt(i) == s2.charAt(i)) {
                i++;
            }
            return s1.substring(0, i);
        }).orElse("");
        System.out.println("Longest common prefix: " + longestCommonPrefix);

    }
}
```

.reduce((s1, s2) -> {...})

- reduce takes two elements at a time (s1 and s2), applies a function, and returns a result.
- That result will then be compared with the next element in the stream.
- First compares "flower" and "flow" : gives "flow".
- Then compares "flow" and "flight" : gives "fl".

Inside the lambda:

Find the minimum length of the two strings.

Start comparing characters one by one.

Stop when characters differ or when we reach the end.

s1.substring(0, i) returns the prefix that matched.

Output: Longest common prefix: fl

246 Streams 23: Max Product in a given array

```
public class StreamsSample {
    public static void main(String[] args) {

        int[] array = { 1, 4, 9, 6, 2, 7, 8 };
        Integer maxProduct = IntStream.range(0, array.length)
            .map(i -> IntStream.range(i + 1, array.length)
                .map(j -> array[i] * array[j]))
            .max()
            .orElse(0)
            .max()
            .orElse(0);

        System.out.println(maxProduct);
    }
}
```

```
}
```

The problem is about finding the maximum product that can be formed by multiplying any two different numbers in the array.

- Think of it like trying every pair, calculating their product, and then just picking the largest one.
- For example, in {1, 4, 9, 6, 2, 7, 8}, the biggest product is from $9 \times 8 = 72$.

Step by step

1. `IntStream.range(0, array.length)`
 - This creates a stream of all indices from 0 to `array.length - 1`.
 - Each index `i` represents the first number in a pair.
2. `IntStream.range(i + 1, array.length)`
 - For each `i`, this creates another stream of indices starting from `i+1` to the end.
 - Why `i+1`? To avoid repeating pairs and avoid multiplying a number with itself.
 - So if `i = 2` (say number 9), it pairs 9 with every number after it.
3. `.map(j -> array[i] * array[j])`
 - For each valid pair (`i, j`), it multiplies the two numbers.
 - Example: if `i=2` (9) and `j=6` (8), result is 72.
4. Inner `.max().orElse(0)`
 - Among all products generated for a fixed `i`, this picks the largest one.
 - Example: if `i=2` (9), the products are $9 \times 6 = 54$, $9 \times 2 = 18$, $9 \times 7 = 63$, $9 \times 8 = 72$. The max here is 72.
5. Outer `.max().orElse(0)`
 - Now it compares the “best product” for each `i` and keeps the overall maximum.
 - That gives us the largest product across the entire array.

247 Streams 23: First non-repeating character in a String

```
public class StreamsSample {
    public static void main(String[] args) {

        String input = "aabbcdeffg";
        String firstNonRepeating = Arrays.stream(input.split(""))
            .collect(Collectors
                .groupingBy(
                    Function.identity(),
                    LinkedHashMap::new,
                    Collectors.counting())))
    }
}
```

```

        .entrySet()
        .stream()
        .filter(entry -> entry.getValue() == 1)
        .map(Map.Entry::getKey)
        .findFirst()
        .orElse(null);
    System.out.println("First Non-Repeating Character: " + firstNonRepeating);
}
}

```

248 Streams 24: Most Repeated Number in a given list.

```

public class StreamsSample {
    public static void main(String[] args) {

        List<Integer> numbers = Arrays.asList(10, 20, 5, 8, 30, 25, 30);

        Integer mostRepeated = numbers.stream()
            .collect(Collectors
                .groupingBy(Function.identity(),
                           Collectors.counting()))
            .entrySet()
            .stream()
            .max(Map.Entry.comparingByValue())
            .map(Map.Entry::getKey)
            .orElse(null);
        System.out.println("Most Repeated Number: " + mostRepeated);
    }
}

```

249 Streams 25: Frequency of words or count of each word in a String

```

public class StreamsSample {
    public static void main(String[] args) {
        String sentence = "Java is fun and java is powerful";
        Map<String, Long> frequencySplit = Arrays
            .stream(sentence.toLowerCase().split("\\s"))
            .collect(Collectors
                .groupingBy(Function.identity(),
                           Collectors.counting()));
        System.err.println("Result " + frequencySplit);
    }
}

```

Explanation:

- `sentence.toLowerCase()` : converts everything to lowercase so "Java" and "java" are treated the same.
Result: "java is fun and java is powerful"

- `.split("\\s")` : splits the string by whitespace into words.
Result: ["java", "is", "fun", "and", "java", "is", "powerful"]
- `Arrays.stream(...)` -> turns that array into a stream.
- `.collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))`
`groupingBy(Function.identity())` : groups by the word itself.
`Collectors.counting()` : counts how many times each word appears.
Result {java=2, is=2, fun=1, and=1, powerful=1}

250 Streams 26-35: Scenario based questions on Student Data

We have Student records with Firstname, lastname, city, grade, age and department

```
List<Student> students = Arrays.asList(
    new Student("Rahul", "Sharma", "Hyderabad", 8.38, 19, "Civil"),
    new Student("Amit", "Verma", "Delhi", 8.4, 21, "IT"),
    new Student("Suresh", "Reddy", "Chennai", 7.5, 20, "Civil"),
    new Student("Kiran", "Patel", "Mumbai", 9.1, 20, "IT"),
    new Student("Arjun", "Naidu", "Bengaluru", 7.83, 20, "Civil")
);
```

Questions:

1. Find students from Hyderabad with a grade greater than 8.0
2. Find the student with the highest grade
3. Count the number of students in each department
4. Find the average grade per department
5. List students sorted by age and then by grade
6. Create a comma-separated list of student names
7. Check if all students are above 18
8. Find the department with the most students
9. Divide students into those who have grades above 8.0 and below
10. Find the student with the longest full name

Code link: [StreamsScenarios/StreamsOnStudentData.java at main · CodingLyf-Fullstack/StreamsScenarios](#)

251 Streams 36-45: Scenario based questions on Employee Data

We have Employee data with Name, Department, Age, Gender

```
EmployeeDto employee1 = new EmployeeDto("SRK", "ECE", 31, "Male");
EmployeeDto employee2 = new EmployeeDto("Salman", "CS", 44, "Male");
EmployeeDto employee3 = new EmployeeDto("Katrina", "ECE", 21, "Female");
EmployeeDto employee4 = new EmployeeDto("Kareena", "CS", 34, "Female");
EmployeeDto employee5 = new EmployeeDto("Hrithik", "EEE", 30, "Male");
EmployeeDto employee6 = new EmployeeDto("Aish", "EEE", 25, "Female");
```

Questions:

1. Find the names of all Employees in the CS department, sorted by age in descending order
2. Group Employees by department and count how many Employees are in each department
3. Find the youngest female Employee.
4. Create a map of department -> list of Employee names.
5. Find the average age of Employees in each department.
6. Get a list of unique departments Employees belong to
7. Partition Employees into male and female groups, then list their names.
8. Group employees by department, then within each department find the oldest employee
9. Build a map of gender with average age of employees sorted by average age descending
10. For each department, find the youngest employee, but instead of returning the employee object, return only their name in uppercase.
11. Return a map where keys will be first letter of the name and value will be the set of names starting with that letter, no solution provided, try on your own.

Code link: [StreamsScenarios/StreamsOnEmployeeData.java at main · CodingLyf-Fullstack/StreamsScenarios](#)

252 Stream 46-50: Scenario based questions on City Data

We have City data with name, population.

```
List<City> cities = Arrays.asList(
    new City("Delhi", 12000),
    new City("Mumbai", 800000),
    new City("Bangalore", 450000),
    new City("Hyderabad", 1200000),
    new City("Chennai", 60000)
);
```

Questions:

1. City with the second highest population
2. Group by first character of name, then max population in each group
3. Average population of top 3 most populated cities.
4. Map of population range -> city names.
5. Using reduce: String of cities ordered by population.

Expected output: Hyderabad(1200000) > Mumbai(800000) > Bangalore(450000) > Chennai(60000) > Delhi(12000)

Code link: [StreamsScenarios/StreamsOnCityData.java at main · CodingLyf-Fullstack/StreamsScenarios](#)

Spring

253 Spring 1: What is Spring boot?

Spring Boot is a framework built on top of Spring that makes it faster and easier to develop Java applications. Normally with Spring, we have to do a lot of manual setup, configuration, and XML files. Spring Boot removes that pain by giving auto-configuration, embedded servers, and prebuilt dependencies. So, we can just focus on writing business logic instead of wiring everything.

Indirect Question:

1. What's new in Spring boot 3.x?

Spring Boot 3.x makes some big changes. First, it switches from the old javax.* packages to the new jakarta.* ones because it now uses Jakarta EE 10. It also requires Java 17 or higher, so we get access to modern Java features. Another big addition is support for GraalVM native images, which lets us build apps that start super-fast and use less memory , perfect for cloud and container environments.

2. Can we create a non-web application in Spring boot?

Yes, Spring Boot can be used to build non-web applications. We can exclude the web-starter dependency and use CommandLineRunner or ApplicationRunner to run code at startup. It is often used for batch jobs, schedulers, or standalone services.

254 Spring 2: What are the advantages of Spring boot?

Spring Boot is a framework that creates stand-alone, production grade Spring based applications. So, this framework has so many advantages.

- **Easy to use:** The majority of the boilerplate code required to create a Spring application is reduced by Spring Boot.
- **Rapid Development:** Spring Boot's opinionated approach and auto-configuration enable developers to quickly develop apps without the need for time-consuming setup, cutting down on development time.
- **Scalable:** Spring Boot apps are intended to be scalable. This implies they may be simply scaled up or down to match the application's needs.

- **Production-ready:** Metrics, health checks, and externalized configuration are just a few of the features that Spring Boot includes and are designed for use in production environments.
- **Hot reloading** allows developers to make changes to their code, resources, or configuration files while an application is running, and see those changes immediately reflected without needing to **restart the application**.

255 Spring 3: What are embedded containers / servers in Spring and what is the default one? how to add Jetty for example?

In normal Java web apps, we used to deploy the WAR file into an external server like Tomcat or Jetty. That meant installing and managing the server separately.

Spring Boot ships with an embedded server inside the application.

- The app is packaged as a JAR (not WAR).
- When we run it, the server starts along with the code.
- We don't need to deploy to any external container.

What's the default server?

By default, Spring Boot uses Apache Tomcat as the embedded server.

So, if we don't configure anything, we are already running on Tomcat without even realizing it.

Suppose we prefer Jetty instead of Tomcat. Here is what we do in Maven:

Exclude Tomcat (since it comes by default) and Add Jetty dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

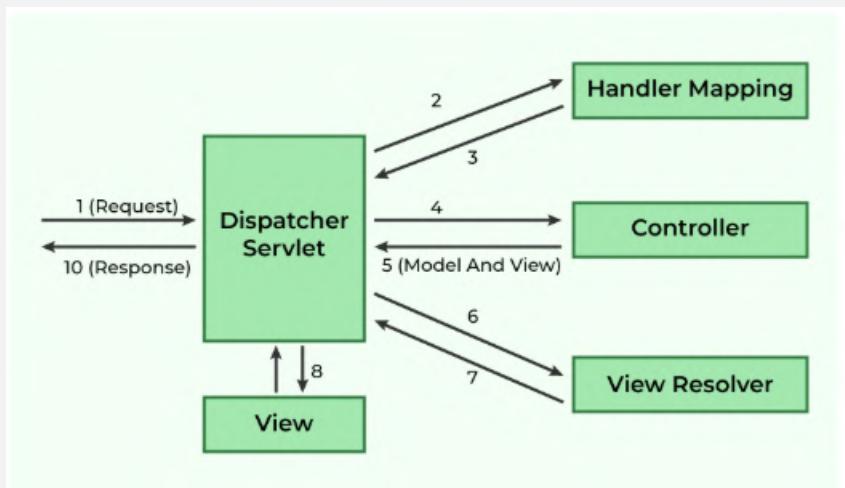
256 Spring 4: Difference between Spring and Spring Boot?

- **Spring:** It is a framework that gives us tools for dependency injection, data access, and building applications. But it requires a lot of setup, XML configurations, and deployment steps.

- **Spring Boot:** It is built on top of Spring and provides auto-configuration, embedded servers, and starter dependencies. It cuts down setup time and allows us to build production-ready apps with less effort.

In short: Spring is the foundation; Spring Boot is the fast and modern way to use Spring.

257 Spring 5: Explain Spring MVC architecture?



The Spring MVC architecture is a Model-View-Controller (MVC) framework used for building web applications in Java. It promotes a clear separation of concerns, making applications more modular, maintainable, and easier to develop.

Key Components of Spring MVC Architecture:

- **DispatcherServlet (Front Controller):**
 - Acts as the central entry point for all incoming HTTP requests.
 - Initializes the Spring application context and loads configurations.
 - Delegates requests to appropriate controllers based on configured mappings.
- **HandlerMapping:**
 - Maps incoming requests to specific controller methods.
 - Determines which controller should handle a particular request based on URL patterns, HTTP methods, and other criteria.
- **Controller:**
 - Handles user input and business logic.
 - Processes requests, interacts with the Model (business logic and data), and prepares data for the View.
 - Returns a **ModelAndView** object, which contains the view name and the model data.

- **ModelAndView:**
 - A container object that holds both the model data and the name of the view to be rendered.
- **ViewResolver:**
 - Resolves the logical view name returned by the controller into a concrete View implementation (e.g., a JSP page, Thymeleaf template).
- **View:**
 - Renders the user interface, displaying the data from the Model.
 - Responsible for generating the final output (e.g., HTML, JSON, XML) to be sent back to the client.

258 Spring 6: What is @SpringBootApplication?

It is a combination of three important Spring annotations:

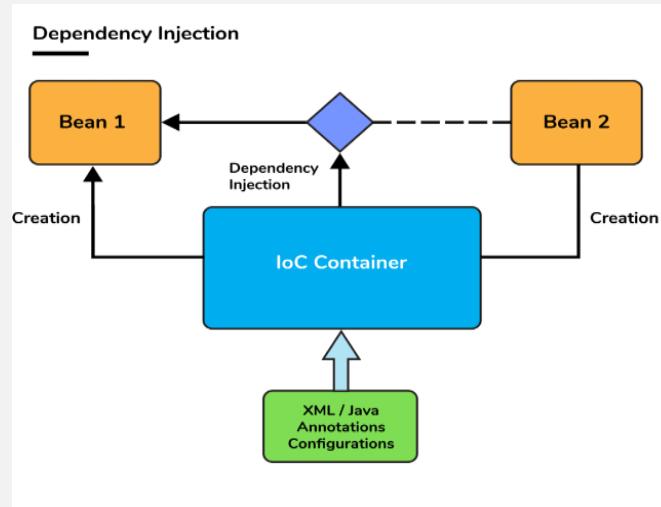
- **@EnableAutoConfiguration:** Automatically configures beans depending on what's on the classpath. For example, when we illustrate the spring-boot-starter-web dependency in the classpath, Spring boot auto-configures Tomcat , and Spring MVC .
- **@ComponentScan :** This annotation scans the components (@Component, @Service, etc.) in the package of annotated class and its sub-packages.
- **@Configuration:** This annotation configures the beans and packages in the class path.

So, using @SpringBootApplication at the main class makes the application ready with all these features at once.

259 Spring 7: What is Dependency Injection?

Dependency injection is a design pattern where objects don't create their own dependencies. Instead, the container (Spring IOC) provides the required objects.

For example, if a Car class needs an Engine, instead of writing new Engine(), Spring inject the engine for us. This reduces coupling, makes testing easier (we can inject mock objects), and improves maintainability.



```

@Component
class Engine {}

@Component
class Car {
    @Autowired
    private Engine engine; // Spring will inject Engine object here
}

```

Interview Tip:

- Dependency Injection is about providing an object with its dependencies from the outside rather than creating them internally. This makes the code loosely coupled, easier to maintain, and easier to unit test.
- In Spring, we can do this using constructor injection, setter injection, or field injection with annotations like `@Autowired` and `@Qualifier`.
- Mention about how you implemented a repository and service layers in your project. Example: In a service class which needs repository, we inject the repository instead of creating it, which makes testing and maintenance straightforward.

Indirect Questions:

1. How does Spring achieve Inversion of Control (IoC)?

IoC is achieved in Spring through Dependency Injection (DI), where the Spring container manages the lifecycle and dependencies of the objects, freeing the developer from manually instantiating dependencies.

2. How does Spring boot makes Dependency injection easier compared to traditional Spring MVC?

Spring boot makes dependency injection easier compared to traditional Spring by auto-configuring the beans and reducing the need for explicit configuration. In traditional Spring, we had to define beans and their dependencies in XML files or with annotations, which can be complex for large applications.

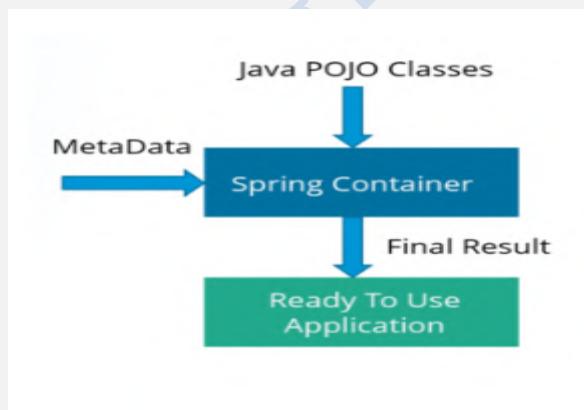
But in Spring boot, we use Auto-configuration and component scanning to automatically discover and register beans on the application's context and classpath. This means now we don't have manually wire up beans.

260 Spring 8: What is IOC Container?

IOC (Inversion of Control) container is the core part of Spring. It is responsible for:

- Creating objects (beans).
- Injecting dependencies.
- Managing the life cycle of those objects.

Instead of we controlling how objects are created and connected, the control is inverted to the container. Spring will take care of creating the beans and managing them. When ever we are ask for some object, Spring will provide it. There are two main types: **BeanFactory** (basic) and **ApplicationContext** (advanced, most used).



```

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        ApplicationContext context =
            SpringApplication.run(DemoApplication.class, args);

        Car car = context.getBean(Car.class);
        car.drive();
    }
}
  
```

Here, the **ApplicationContext** (IOC container) manages all beans (Car, Engine, etc.).

We just ask the container for the bean instead of creating it ourselves.

Reference: [Understanding IoC Container in Spring Boot with a Real-Time Project Example | by A cup of JAVA coffee with NeeSri | Medium](#)

Indirect Questions:

1. What are the types of IOC containers in Spring?

- a. BeanFactory: BeanFactory is like a factory class that contains a collection of beans. It instantiates the bean whenever asked for by clients.
- b. ApplicationContext: The ApplicationContext interface is built on top of the BeanFactory interface. It provides some extra functionality on top BeanFactory.

2. What is the default container?

ApplicationContext is the default and preferred container, adding enterprise features like events, internationalization, AOP, and annotation support.

3. Difference between BeanFactory and ApplicationContext.

| Aspect | BeanFactory | ApplicationContext |
|----------------------------|---|--|
| Definition | Basic IoC container, provides only DI | Advanced IoC container, built on top of BeanFactory |
| Bean Initialization | Lazy (beans created only when requested) | Eager by default (all singleton beans created at startup) |
| Features | Only manages beans | Adds event handling, internationalization, AOP integration, annotation scanning, BeanPostProcessor support |
| Performance | Lightweight, good for memory-constrained apps | Slightly heavier (loads more at startup) but faster access later |
| Configuration | XmlBeanFactory (deprecated after Spring 3.x) | ClassPathXmlApplicationContext, AnnotationConfigApplicationContext, WebApplicationContext |
| Use Case | Rarely used today, legacy/simple apps | Default choice for modern Spring & Spring Boot apps |
| Support | No support for @Autowired, | Full support for annotations, enterprise beans, and Boot autoconfiguration |

261 Spring 9: What is the difference between @Component, @Service, and @Repository?

- **@Component:** The most generic annotation. It marks a class as a Spring bean.
- **@Service:** A specialization of **@Component**, used for classes that hold business logic. This makes code easier to understand since we know it belongs to the service layer.
- **@Repository:** Another specialization, used for persistence layer (DAO classes).
Point to remember: It also provides automatic translation of database-related exceptions into Spring's `DataAccessException`.

So basically, they all register beans with the container, but **@Service** and **@Repository** give extra meaning and behaviour for their specific layers.

```
@Component // generic bean
class UtilityHelper {}

@Service // business logic
class PaymentService {
    public void processPayment() {
        System.out.println("Payment processed");
    }
}

@Repository // data access layer
class UserRepository {
    public void saveUser(String name) {
        System.out.println("User " + name + " saved to DB");
    }
}
```

Indirect Questions:

1. Why do we need to separate controller, business and DAO layers?

Think of it like building a house: We wouldn't ask an electrician to lay the foundation, and we wouldn't ask a plumber to build the roof. Each expert should focus on their own work. In the same, In Spring, each layer has its job, and separating them keeps things clear, safe, and easier to maintain.

- **Controller layer:** Handles HTTP requests and responses. It is the entry point where the outside world talks to our app.
- **Business (Service) layer:** It contains the business logic

- **DAO (Data Access Object) layer:** Talks to the database. It only cares about saving, reading, updating, and deleting data.

Why separate?

1. **Clean code:** Each part has one clear responsibility.
 2. **Easier to change:** We can swap a database or change business rules without touching controllers.
 3. **Testability:** We can test each layer in isolation.
 4. **Reusability:** Services can be reused across different controllers or APIs.
2. Is it required to write `@Repository` annotations in spring data JPA?

`@Repository` annotation is not strictly required when we use Spring Data JPA.

Why? Because when we create a repository interface that extends `JpaRepository`, `CrudRepository`, or `PagingAndSortingRepository`, Spring automatically finds it and creates an implementation at runtime. That implementation is registered as a Spring bean, so we can use it with `@Autowired` even without `@Repository`.

```
public interface UserRepository extends JpaRepository<User, Long> {  
}
```

So, what is the point of `@Repository` then?

It is mainly about **exception translation**. If something goes wrong in the database (like a `SQLException`), `@Repository` tells Spring to wrap it into a generic `DataAccessException`. This way, the error handling stays consistent

262 Spring 10: What is application.properties file?

It is the file used to store application configuration in Spring Boot. Instead of hardcoding values, we put them here, so they can be easily changed without touching the code. Example:

```
server.port=8081  
spring.datasource.url=jdbc:mysql://localhost:3306/test  
spring.datasource.username=root  
spring.datasource.password=1234
```

This makes the application flexible and easier to configure across different environments (dev, test, prod).

Additional Question:

1. How will you configure different Databases in your App for different environments?

We can do this using application.properties and Profile. Spring gives us profiles like dev, test, and prod. Each profile can have its own configuration. When the app runs, we simply tell Spring which profile to use, and it picks the right settings.

The most common way is to create different property files for each environment like application-dev.properties, application-prod.properties. When we switch the profile, Spring automatically loads the right file.

Sometimes it is not just the properties that change, the beans themselves need to be different. In that case, we can use @Profile.

```
@Configuration
public class DataSourceConfig {

    @Bean
    @Profile("dev")
    public DataSource devDataSource() {
        return null;
    }

    @Bean
    @Profile("prod")
    public DataSource prodDataSource() {
        return null;
    }
}
```

2. What is relaxed binding in spring boot?

Relaxed binding in Spring Boot means we can define configuration properties in different styles (kebab-case, snake_case, camelCase, or UPPER_CASE) and Spring will still map them correctly to the Java fields.

For example, server.port, server_port, or SERVER_PORT all bind to the serverPort property. This makes configuration flexible and developer-friendly, especially when working with environment variables, YAML, or properties files across different operating systems and deployment setups.

```
# application.properties
myapp.server-url=http://localhost:8080
myapp.serverUrl=http://localhost:8080
myapp.server_url=http://localhost:8080
MYAPP_SERVER_URL=http://localhost:8080 # from environment variable
```

All of these map correctly to serverUrl in MyAppProperties.

```
@ConfigurationProperties(prefix = "myapp")
public class MyAppProperties {
    private String serverUrl;
    // getter & setter
}
```

263 Spring 11: What are Spring boot starters?

Starters are preconfigured dependency bundles that make it easy to add features to the application.

For example, if we want to create a web app, just add **spring-boot-starter-web** and it will bring everything we need: Spring MVC, Tomcat, JSON libraries, etc. Without starters, we would have to add each dependency manually.

- Data JPA starter
- Web starter
- Security starter
- Test Starter
- Thymeleaf starter

264 Spring 12: How many ways dependency injection can be done or Types of Dependency Injection?

Spring supports three main ways:

- Constructor Injection
- Setter Injection
- Field Injection

Constructor Injection: Here, we pass dependencies using the class constructor.

```
@Component
class Car {
    private final Engine engine;

    @Autowired
    public Car(Engine engine) {
        this.engine = engine;
    }
}
```

Setter Injection: We provide Dependencies are passed through setter methods. It is good when dependencies are optional

```
@Component
class Car {
    private Engine engine;

    @Autowired
    public void setEngine(Engine engine) {
        this.engine = engine;
    }
}
```

Field Injection: Dependency is directly injected into the field using @Autowired. It is very easy but not recommended for large projects since it makes testing and refactoring harder.

```
@Component  
class Car {  
    @Autowired  
    private Engine engine;  
}
```

Interview tip: Always say constructor injection is best practice because it makes the class immutable and easier to test.

Indirect Questions:

1. Why Setter Injection is not recommended?

Setter injection is okay for optional things, but not for important ones. If we forget to call the setter, the object may not work and can throw errors later. If we forget to give @Autowired to setter we will get Null pointer. We can also alter the objects using setters.

2. When to choose field injection over constructor injection?

Choose field injection only for very simple, non-critical beans and optional. For example, a logger or a utility that's not central to the object's main job. It makes the code shorter but it hides what class actually needs, so it's not recommended for important dependencies.

3. Why constructor injection is recommended?

Constructor injection is recommended because it forces all required dependencies to be provided when the object is created. The object is complete and stable from the beginning, and its dependencies can't be changed later. This makes the design clear, the code easier to test, and the application more reliable.

4. If a class needs too many dependencies for example 7-10, then how do you design it?

If a class needs 7 different dependencies, you still use **constructor injection**, but instead of making a huge constructor hard to read, you can:

- **Group related dependencies** into a separate class (a configuration or service aggregator).
- **Use interfaces** to reduce the number of direct dependencies.

Reference: [Spring Core: Why Constructor Based Dependency Injection is Recommended?](#)

265 Spring 13: What do you mean by Annotation-based container configuration?

Before Spring annotations, people used **XML files** to declare beans. That was lengthy and hard to maintain.

Now Instead of XML files, we can configure Spring beans using annotations. This is called annotation-based configuration.

Some common annotations we use:

- `@Component`: Marks a class as a Spring bean.
- `@Service` : Marks a class as service layer bean.
- `@Repository` : Marks a class as DAO bean.
- `@Controller / @RestController` : For web controllers.
- `@Configuration` : Defines configuration class.
- `@Bean` : Defines a bean inside `@Configuration`.
- `@Autowired` : Injects dependencies.
- `@Value` : Injects values from properties file.

266 Spring 14: Explain about `@Autowired`?

The `@Autowired` annotation in Spring is a mechanism for automatic dependency injection. It allows the Spring IoC (Inversion of Control) container to automatically resolve and inject dependencies into the Spring-managed components (beans) without requiring explicit configuration in XML or Java.

```
// Service class
@Service
class GreetingService {
    public String greet() {
        return "Hello, Spring!";
    }
}

// Component that uses the service via constructor injection
@Component
class MyApp {
    private final GreetingService greetingService;

    // Constructor injection
    @Autowired // Optional since Spring 4.3 if there's only one constructor
    public MyApp(GreetingService greetingService) {
        this.greetingService = greetingService;
    }

    public void run() {
        System.out.println(greetingService.greet());
    }
}
```

Point to remember: If there are multiple beans of the same type, Spring throws an error unless we use @Qualifier or @Primary.

Indirect Question:

- How would you achieve the same thing that the auto-wired annotation does without using the annotation?

Instead of @Autowired, we declare beans in a @Configuration class:

```
@Configuration
class AppConfig {
    @Bean
    public Service service() {
        return new ServiceImpl();
    }

    @Bean
    public Client client() {
        return new Client(service()); // manual injection
    }
}
```

Here, Client gets Service injected, same as @Autowired.

@Autowired is just a shortcut for dependency injection. Without it, we do the wiring explicitly in XML or @Bean methods.

- Is Autowired is mandatory when there is only one constructor?

Autowired is Optional since Spring 4.3 if there's only one constructor

267 Spring 15: Explain about @Qualifier and @Primary?

In Spring, both @Qualifier and @Primary are used to resolve ambiguity when multiple beans of the same type are available for dependency injection.

@Qualifier

Why we need it: When there are multiple beans of the same type, Spring gets confused. @Qualifier tells Spring to pick the exact one using the name.

How it works: We need to give the bean a name and then point to it when injecting.

```
@Component("creditCard")
class CreditCardPayment implements Payment {}

@Component("debitCard")
class DebitCardPayment implements Payment {}

@Component
class PaymentService {
    @Autowired
    @Qualifier("creditCard") // pick credit card specifically
    private Payment payment;
}
```

@Primary

Why we need it: If multiple beans exist but one is used most often, then we can mark it as Primary.

How it works: Mark one bean as @Primary, and Spring will pick it by default when we use @Autowired without qualifiers.

```
@Component
@Primary
class CreditCardPayment implements Payment {} // default

@Component
class DebitCardPayment implements Payment {}

@Component
class PaymentService {
    @Autowired
    private Payment payment; // gets CreditCardPayment by default
}
```

Indirect Questions:

1. How do you handle multiple beans of the same type in Spring??

Use @Qualifier with @Autowired to specify which bean to inject. This ensures the correct bean is selected when duplicates exist.

2. When you use both @Primary and @Qualifier, which one will take precedence?

When both @Qualifier and @Primary are used in, @Qualifier takes precedence. @Primary indicates a default bean to use when multiple beans exist, while @Qualifier allows for very specific bean selection. When both are present, Spring prioritizes the bean identified by @Qualifier

3. Two beans of same type are found and Spring throw NoUniqueBeanDefinitionException. How would you resolve it?

Use @Qualifier with @Autowired to specify which bean to inject. This ensures the correct bean is selected when duplicates exist.

4. There is only one bean but you annotated with @Qualifier and @Primary, how does it affect?

If there is only one bean of that type:

- @Primary has no effect, because there is nothing to choose between.
- @Qualifier also doesn't matter, the single bean will be injected regardless.

They only come into play when multiple beans of the same type exist.

268 Spring 16: Difference between application.properties vs application.yml?

Both are used for external configuration in Spring Boot, but the **syntax style** is different.

application.properties (key-value pairs):

```
server.port=8081
spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.username=root
```

application.yml (YAML format):

```
server:
  port: 8081
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/test
    username: root
```

Indirect Question:

- When do you prefer .properties and .yml file?

For simplicity and small projects: application.properties might be preferred due to its directness.

For complex, hierarchical configurations and improved readability: application.yml is generally considered superior due to its structured nature and support for various data types.

Interview Tip: These file helps us to keep configs outside the code, so we can change them per environment without touching Java classes.

269 Spring 17: What is the use of @Value / How to load config values dynamically?

@Value is used to inject values from application.properties or application.yml into our code.

```
app.name=MySpringApp

@Component
class Config {
    @Value("${app.name}")
    private String appName;

    public void printName() {
        System.out.println(appName); // MySpringApp
    }
}
```

Indirect Question:

- How will you read values mentioned in properties file? Using @value

2. If you want to inject dynamic port from a different config file, how will you use value?

```
@Value("${server.port}")
private int port;
```

270 Spring 18: What is the use of @ConfigurationProperties / How to load config values dynamically?

`@ConfigurationProperties` is a Spring annotation used to bind external configuration (like `application.properties` or `application.yml`) to a Java class. Instead of reading each property with `@Value`, we can map a whole group of related properties into a strongly-typed class.

```
app:
email:
  host: smtp.example.com
  port: 587
  username: user@example.com
```

```
@Component
@ConfigurationProperties(prefix = "app.email")
public class EmailProperties {
    private String host;
    private int port;
    private String username;

    // getters and setters
}
```

Spring automatically reads `app.email.host`, `app.email.port`, `app.email.username` from the configuration file.

Values are injected into the `EmailProperties` bean.

271 Spring 19: How do profiles work in Spring boot, Explain about `@Profile`?

In the real time projects, we might need different configurations for different environments like Prod, QA and Dev.

This is where profiles come into picture. Spring boot allows us to manage application configurations and components based on the environment in which the application is running.

This can be done through various methods:

- `application-{profile}.properties` or `application-{profile}.yml`: Separate configuration files are created for each profile (e.g., `application-dev.properties`, `application-prod.yml`).
- `@Profile` annotation: This annotation is used on classes (e.g., `@Configuration`, `@Component`, `@Service`, `@Repository`) or methods to indicate that they should only be loaded or activated when a specific profile is active

Key points:

Profiles are activated at run time using JVM system properties: -Dspring.profiles.active=prod.

Or Using spring.profiles.active property, Set in application.properties, application.yml

@Profile

The @Profile annotation used to conditionally register beans or load configuration based on the active profiles.

It is applied to:

@Configuration classes: To load specific configuration classes only when a particular profile is active.

```
@Configuration
@Profile("production")
public class ProductionConfiguration {
    // ... Beans specific to production environment
}
```

@Component, @Service, @Repository: To register specific implementations of interfaces or classes based on the active profile.

```
@Service
@Profile("dev")
public class DevUserServiceImpl implements UserService {
    // ... Development-specific user service implementation
}

@Service
@Profile("!dev") // Active when 'dev' profile is NOT active
public class ProdUserServiceImpl implements UserService {
    // ... Production-specific user service implementation
}
```

@Bean methods: To conditionally create beans within a configuration class.

```
@Configuration
public class AppConfig {
    @Bean
    @Profile("dev")
    public DataSource devDataSource() {
        // ... Development data source
    }

    @Bean
    @Profile("prod")
    public DataSource prodDataSource() {
        // ... Production data source
    }
}
```

272 Spring 20: How spring auto configuration works internally?

Spring Boot Auto Configuration is like having a smart assistant that automatically sets up the Spring application based on what it detects in the project.

When we add a dependency (like Spring Data JPA or Spring MVC), Spring Boot notices this and auto configure the Data source

How it works:

Spring Boot looks at the classpath (all the jars in the project) and the beans we have already defined.

It checks what's missing and tries to auto-configure.

Point to remember: Auto Configuration is triggered by the `@EnableAutoConfiguration` annotation (or just `@SpringBootApplication` which includes it).

Example: JPA Auto Configuration

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

Let's say when we have dependencies.

Now, we don't need to define a `DataSource` or `EntityManagerFactory`. Spring Boot auto-configures, means we don't need to write the below code, Sprint does it for us.

```
@Bean
public DataSource dataSource() {
    return new org.h2.jdbc.JdbcDataSource();
}
```

Indirect Questions:

1. How Spring Boot knows what to configure?

Spring Uses conditional annotations like:

- `@ConditionalOnClass`: configures bean only if a certain class is on the classpath.
- `@ConditionalOnMissingBean`: configures bean only if we haven't already defined one.
- `@ConditionalOnProperty`: configures bean only if a certain property is set.

2. What happens if I define my own DataSource bean?

Auto Configuration will skip creating a DataSource because of `@ConditionalOnMissingBean`.

3. How do conditional annotations work in Auto Configuration?

They check the environment, classpath, and existing beans to decide whether to create a bean or skip it

4. Can Auto Configuration be disabled?

Yes, using `@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)` or via properties

5. How do you override default auto configurations?

- Define our own bean: Spring Boot will skip its default because most auto configs use `@ConditionalOnMissingBean`.
- Use properties in `application.properties` (e.g., `spring.datasource.url`, `spring.jpa.show-sql`) to change the defaults.
- Exclude specific auto configs with `@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)`.

6. What if two auto-configurations conflict?

If two auto-configurations provide the same bean or configure the same feature differently, we can end up with a conflict. This might cause:

1. **BeanDefinitionOverrideException** : when two beans with the same name get created.
2. **Unexpected behavior** : e.g., two DataSource beans when Spring expected only one.

Solutions:

- Use `@ConditionalOnMissingBean`, `@ConditionalOnClass`, etc., to make configs load only when needed.
- Exclude one of the auto-configs via `spring.autoconfigure.exclude` in `application.properties`.

7. How to tell auto-configuration not to create bean when already bean exists?

Use `@ConditionalOnMissingBean`

273 Spring 21: Difference between `@Bean` vs `@Configuration`

`@Configuration`

It is a class-level annotation. When we annotate a class with `@Configuration`, Spring treats it as a source of bean definitions and manages it as a full Spring configuration.

Point to remember: `@Configuration` classes are **enhanced with CGLIB proxies**. This means if one method calls another bean method inside the same class, Spring ensures the bean is **singleton** and doesn't create a new instance each time.

```
@Configuration
public class AppConfig {

    @Bean
    public Service service() {
```

```

        return new Service(repository());
    }

    @Bean
    public Repository repository() {
        return new Repository();
    }
}

```

Here, service() depends on repository(). Even though we call repository() directly, Spring ensures the same singleton Repository is used.

@Bean:

@Bean is a method-level annotation. It tells Spring: “*The object returned by this method should be registered as a bean in the Spring context.*”

We can declare @Bean methods in both @Configuration and @Component classes, but there's an **important difference**, if one @Bean method calls another in a @Component class, we might accidentally create multiple instances instead of getting the singleton behaviour.

```

@Component
public class AppConfig {

    @Bean
    public Service service() {
        return new Service(repository());
    }

    @Bean
    public Repository repository() {
        return new Repository();
    }
}

```

When we call repository() directly, Spring may create multiple beans.

Indirect Question:

1. Why do you sometimes get multiple instances of a bean if you don't use @Configuration but just @Bean in a class?

Without @Configuration, Spring doesn't proxy the class, so calling one @Bean method from another creates a new instance instead of reusing the singleton.

2. What happens if a @Bean method calls another @Bean method inside the same class?

In a non-@Configuration class, calling a @Bean method directly bypasses Spring, so it creates a fresh object instead of using the Spring-managed bean.

3. Can you use @Bean without @Configuration? What are the implications?

Yes, it works, but we lose singleton guarantees between methods, and Spring simply registers whatever object the method returns.

274 Spring 22: Difference between @Bean vs @Component

@Component

It is a class-level annotation. If we put this annotation on top of a class and that class is inside a package that Spring scans, Spring will automatically create an object of it and manage it as a bean.

@Component works when we write our own code. We put the annotation right on the class.

@Component

```
public class EmailService {
    public void sendMail(String to) {
        System.out.println("Mail sent to " + to);
    }
}
```

Now, as long as the @SpringBootApplication (or @ComponentScan) is scanning the package, EmailService is picked up with no extra configuration. It is fast, simple, and we usually use it for our application-level services, repositories, and controllers.

@Bean:

@Bean is a method-level annotation. @Bean is more like the “manual way.” We use it inside a class annotated with @Configuration. We explicitly tell Spring *how* to build it

For Example, If we use a class from an outside library because we can't add @Component to it as we don't have control to its code. Instead, we use @Bean in our own configuration class. We write a method that creates the object for that library class and returns it.

```
@Bean
public DataSource dataSource(Environment env) {
    return DataSourceBuilder.create()
        .url(env.getProperty("db.url"))
        .username(env.getProperty("db.user"))
        .build();
}
```

For Example. DataSource is a class from a third-party JDBC library, not our own code, so we can't mark it with @Component. Instead, we create it as a Spring bean by writing a @Bean method that returns a DataSource object with the right configuration.

275 Spring 23: What is Spring Bean

- A java class which is managed by the IOC container is called as Spring Bean. The life cycle of the spring bean is taken care by the IOC container.
- Spring beans are instantiated, configured, wired, and managed by IoC container.
- Beans are created with the configuration metadata that the users supply to the container (by means of XML or java annotations configurations.)

Additional questions:

1. What is Pojo and DTO?

A POJO (Plain Old Java Object) is a simple Java class that holds data and can also include business logic. A DTO (Data Transfer Object) is a lightweight object used only to carry data between layers or services.

Pojo:

```
public class User {
    private Long id;
    private String name;
    private String email;
    private String password; // sensitive
    private LocalDate createdAt; // internal

    // getters, setters, maybe business methods
}
```

DTO:

```
public class UserDTO {
    private String name;
    private String email;

    // only the data we want to expose
}
```

How they differ

- **POJO** has everything: id, password, timestamps, business rules.
- **DTO** is trimmed down: only what the client needs (name + email).

So, when we return user data in an API, we send UserDTO, not the raw User.

2. How is POJO and DTO are different from Spring bean?

A **Spring Bean** is any object managed by the Spring IoC container (with lifecycle, scope, and dependency injection).

POJOs/DTOS are just plain data holders, not managed by Spring unless explicitly registered as beans.

3. When will you get NoSuchBeanException even though there is bean?

- If package scanning is wrong, Spring won't even see our class, so it never creates the bean.
- If we use a wrong qualifier name, Spring looks for a bean with that exact name, can't find it, and throws the exception.
- If the bean is marked with a Profile or Condition, but that profile isn't active or the condition fails, Spring will skip creating the bean.

276 Spring 24: What are bean scopes?

1. Singleton:

Spring creates only one instance of the bean for the entire application context. Spring manages full lifecycle (init and destroy). It is the Default scope.

When to use: For stateless beans or shared services/repositories where all components can use the same instance.

2. Prototype:

Spring creates a new instance every time the bean is requested. Spring calls init methods but does not call destroy methods.

When to use: For stateful beans or temporary objects where each client/request needs a fresh copy, like form objects or calculations.

3. Request:

A new bean instance is created for each HTTP request. Only relevant in web applications. Each request gets a fresh bean.

When to use: For request-specific beans, like handling data for a single web request or form submission.

4. Session:

A new bean instance is created per HTTP session. All requests in the same session share the same bean.

When to use: For user-specific session data, like shopping carts, logged-in user info.

5. Application:

A single bean instance per ServletContext. It is shared across the whole web application.

When to use: For application-wide shared resources, like configuration objects, caches, or global services.

Indirect Questions:

1. If multiple components autowire the same service, do they share the same object?

Yes, with singleton scope, they all share the same instance

2. If I autowire a prototype bean in a singleton bean, will it be a new object every time?

No, the singleton will get one instance at initialization unless we use @Lookup or ObjectFactory.

3. With @Scope("request"), Will two HTTP requests get the same bean instance?

No, each request gets a new instance.

4. If you autowire Prototype bean in singleton bean how many instances of prototype bean would be created?

- If we autowire a prototype bean into a singleton bean, only one instance of the prototype bean gets created at the time of injection.
- That same instance will be reused by the singleton throughout its lifetime.
- Spring doesn't create a new prototype object every time we call it from the singleton.
- To get a fresh prototype on each use, we need **ObjectFactory** or **Provider**.

```

@Component
@Scope("prototype")
class PrototypeBean {
    public PrototypeBean() {
        System.out.println("New PrototypeBean created");
    }
}

@Component
class SingletonBean {
    @Autowired
    private ObjectFactory<PrototypeBean> objectFactory;

    @Autowired
    private Provider<PrototypeBean> provider;

    public void usePrototype() {
        PrototypeBean bean1 = objectFactory.getObject(); // fresh instance
        PrototypeBean bean2 = provider.get();           // fresh instance
    }
}

```

Whenever we call `usePrototype()`, both `objectFactory.getObject()` and `provider.get()` will give a new `PrototypeBean` instance instead of reusing the old one.

5. Can we inject singleton bean into request-scoped bean?

Yes, the request-scoped bean has a shorter lifetime (one HTTP request) than the singleton bean (the entire application). The singleton bean is fully initialized and available before any request comes in. The request-scoped bean can simply have a reference to the singleton bean injected into it, just like any other dependency.

6. Can we inject request-scoped bean into singleton bean?

The singleton bean is created once at application startup. At that time, there is no active HTTP request, so the request-scoped bean does not exist yet. If Spring tried to inject the request-scoped bean directly into the singleton during construction, it would fail because the context for the required scope (HTTP request) isn't active.

The Solution: Scoped Proxy

Spring's solution is to not inject the actual request-scoped bean. Instead, it injects a proxy object that implements the same interface (or extends the class) as the request-scoped bean.

- This proxy is created at startup and injected into the singleton.
- Whenever a method is called on this proxy during an HTTP request, the proxy delegates the call to the real request-scoped bean instance that was created for that specific request.

How to do it: We must specify `proxyMode = ScopedProxyMode.TARGET_CLASS` (for class-based proxies) or `ScopedProxyMode.INTERFACES` (for interface-based proxies) in the `@Scope` annotation of the request-scoped bean.

Refer below question to understand it more.

What is proxyMode in Spring, how it helps? (Refer Question No. 367, Spring 115)

7. How does scope behave in multithread rest controller?

Singleton (default): Only one instance exists for the entire application. Multiple threads (HTTP requests) share the same instance, so instance variables are shared across threads. we must avoid mutable state unless properly synchronized.

Prototype: A new instance is created each time it is requested from the Spring context. In a REST controller, if we inject a prototype bean, Spring resolves it once at startup for the singleton controller. To get a fresh prototype per request, we need a proxy (@Scope("prototype", proxyMode = ScopedProxyMode.TARGET_CLASS)).

Request / Session / Web Scopes: These scopes create a new bean per HTTP request or session, so each thread (request) gets its own instance. Using proxies ensures the singleton controller can safely access these shorter-lived beans.

Reference: [The Scope of Beans in Spring Boot: A Comprehensive Guide | by Dev Cookies | Medium](#)

277 Spring 25: What is the bean life cycle?

Bean Lifecycle in Spring (Step by Step)

Think of a Spring Bean like a person's life: it is born, it learns things, it works, and finally it is retired (destroyed). The IOC container manages all of this.

Note: There are many life cycle methods but here I am mentioning only the important. We can read from the link below and check all the methods in the image

1. Instantiation (Bean is created)

The Spring container creates an object of the class. Usually, it calls the no-arg constructor or a factory method.

```
public class MyBean {
    public MyBean() {
        System.out.println("Bean is being instantiated");
    }
}
```

2. Populating Properties (Dependency Injection)

After creating the bean instances, Spring injects all the dependencies (other beans or values) into the bean, which can be done through constructor injection, setter injection, or field injection.

```
@Component
public class MyBean {
    private final MyDependency myDependency;
    @Autowired
    public MyBean(MyDependency myDependency) {
        this.myDependency = myDependency;
    }
}
```

3. Initialization (@PostConstruct)

After dependencies are injected, Spring calls methods marked with @PostConstruct.

This is where we put setup logic (like opening a DB connection, loading configs).

It is the safest place to initialize things because all dependencies are already injected.

```
@Component
public class DatabaseInitializer {

    @PostConstruct
    public void init() {
        System.out.println("DB connection established!");
        // Load initial data, cache, etc.
    }
}
```

4. Destruction (@PreDestroy)

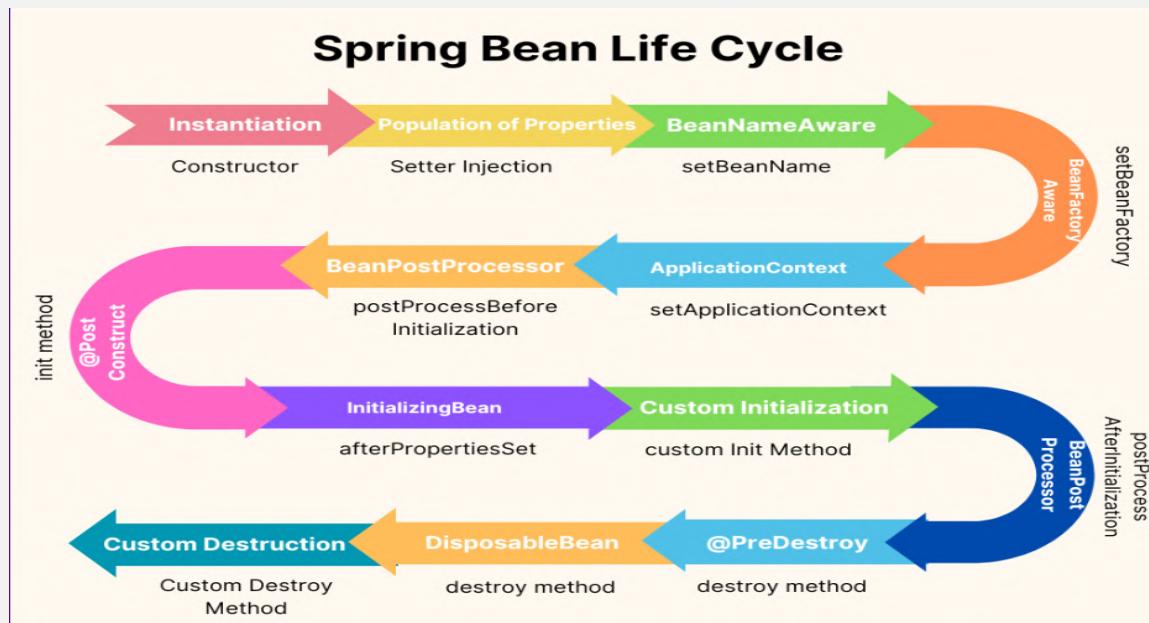
When the Spring container shuts down, methods marked with @PreDestroy run.

Used for cleanup (closing DB connections, releasing resources).

Prevents memory leaks by properly releasing resources.

```
@Component
public class CacheManager {

    @PreDestroy
    public void cleanup() {
        System.out.println("Clearing cache before shutdown!");
        // Close cache, release memory
    }
}
```



Reference: [Spring Bean Life cycle. Content | by Himani Prasad | Medium](#)

278 Spring 26: Can we create beans conditionally, if yes how?

Spring has always been famous for Dependency Injection, but sometimes we don't want a bean to always be created. For example, we may need one type of bean in dev environment and another type in prod. That is where `@Conditional` comes in.

Spring checks the condition **at runtime** before putting the bean in the application context.

Where can we use it?

- **Method level:** On a `@Bean` method > conditionally register that single bean.
- **Class level:** On a `@Configuration` class > all beans inside follow the condition.
- **Component level:** On `@Component`, `@Service`, `@Repository` > conditional scanning.

There are 4 types of `@Conditional`

The `@ConditionalOnProperty` annotation creates the bean only if the specified property is present in the environment and optionally has a specific value. It is commonly used to enable or disable beans based on application configuration properties

```
@Bean
@ConditionalOnProperty(name = "feature.email.enabled", havingValue = "true")
public EmailService emailService() {
    return new EmailService();
}
```

The `@ConditionalOnClass` annotation creates the bean only if the specified class is present on the classpath. It is useful for creating beans that depend on the presence of a particular library or class.

```
@Bean
@ConditionalOnClass(name = "com.mongodb.MongoClient")
public MongoTemplate mongoTemplate() {
    return new MongoTemplate();
}
```

The `@ConditionalOnMissingBean` annotation creates the bean only if the specified bean type or name is not already defined in the application context.

```
@Bean
@ConditionalOnMissingBean
public MyBean defaultBean() {
    return new MyBean();
}
```

The **@ConditionalOnBean** annotation creates the bean only if the specified bean type or name is already defined in the application context. This annotation is useful for defining beans that depend on the presence of other beans.

```
@Bean
@ConditionalOnBean(DataSource.class)
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}
```

Real Time Example: Say we want our Spring Boot app to work with Firestore in dev and MongoDB in prod.

application.properties: db.type=firebase # or mongo

```
@Configuration
public class DatabaseConfig {

    @Bean
    @ConditionalOnProperty(name = "db.type", havingValue = "firebase")
    public FirestoreService firestoreService() {
        return new FirestoreService();
    }

    @Bean
    @ConditionalOnProperty(name = "db.type", havingValue = "mongo")
    public MongoService mongoService() {
        return new MongoService();
    }
}
```

This way, we don't have to change the code when moving environments, just changing the property will work.

Indirect Questions:

1. How would you enable or disable a bean without touching the code, just by configuration?

Use **@ConditionalOnProperty**.

2. If I want to create a bean only if MongoDB driver is on the classpath, how do I handle it?

Use **@ConditionalOnClass**.

3. Suppose you already have a custom bean defined. How do you stop Spring Boot from creating its default one?

Use **@ConditionalOnMissingBean**.

279 Spring 27: How does @Conditional works internally?

Under the hood, Spring's @Conditional annotation works by leveraging the **Condition** interface. This interface must be implemented by a class that contains the conditional logic. When Spring evaluates this logic, it decides whether to register the annotated bean or not.

```
public class MyCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        String env = context.getEnvironment().getProperty("app.env");
        return "prod".equalsIgnoreCase(env);
    }
}
```

```
@Configuration
@Conditional(MyCondition.class)
public class ProdConfig {
    @Bean
    public DataSource prodDataSource() {
        return new HikariDataSource();
    }
}
```

Here, the bean will only load if app.env=prod.

280 Spring 28: Difference between CommandLineRunner and ApplicationRunner?

These two interfaces in Spring Boot are used to run some code right after the application starts up (After the Spring application context is fully initialized. They are super useful for initialization tasks like:

- Loading test data
- Running startup checks
- Starting background processes

CommandLineRunner

- It is an interface in Spring Boot.
- If the class implements it, the run(String... args) method will execute right after the Spring ApplicationContext is loaded and before the application fully starts serving requests.
- The method gives the access to the raw command-line arguments as a String[].

```
@Component
public class MyStartupRunner implements CommandLineRunner {

    @Override
    public void run(String... args) throws Exception {
        System.out.println("App started with args: " + Arrays.toString(args));

        // Example: Load test data if "--load-test-data" is passed
    }
}
```

```

        if (args.length > 0 && args[0].equals("--load-test-data")) {
            System.out.println("Loading test data...");
        }
    }
}

```

ApplicationRunner

ApplicationRunner is another interface in Spring Boot that serves a similar purpose to CommandLineRunner, but it provides a more structured way to access and parse command-line arguments. The main difference is: instead of giving a raw String[], it provides an ApplicationArguments object.

Indirect Questions:

1. How do you run some logic right after Spring Boot starts?

Using CommandLineRunner or ApplicationRunner.

2. If you just need to check if a simple flag like --enable-feature was passed, which runner would you use?

CommandLineRunner is enough since we are just checking raw args.

3. If your Spring Boot app needs to read --db-url=localhost:3306 from command line, which runner is better?

ApplicationRunner because it easily handles --key=value arguments.

4. If you had both CommandLineRunner and ApplicationRunner in your project, which one runs first?

Both run, order depends on @Order annotation or Ordered interface.

```

@Component
@Order(1) // Runs first
public class FirstRunner implements CommandLineRunner { ... }

@Component
@Order(2) // Runs second
public class SecondRunner implements ApplicationRunner { ... }

```

281 Spring 29: How to handle Circular dependencies?

A circular dependency happens when two (or more) beans depend on each other to get created.

For example:

```

@Component
class A {
    private final B b;
    public A(B b) { this.b = b; }
}

```

```

}

@Component
class B {
    private final A a;
    public B(A a) { this.a = a; }
}

```

Here, A needs B and B needs A.

When Spring tries to create them, it gets stuck in a loop and fails with a **BeanCurrentlyInCreationException**.

There are a few ways to handle this:

1. Use `@Lazy` injection

Lazy tells Spring: “don’t create this bean immediately, create it only when needed.”

This breaks the cycle because Spring can first create one bean and keep a proxy for the other.

```

@Component
class A {
    private final B b;
    public A(@Lazy B b) { this.b = b; }
}

```

Now Spring can create A first, and only when we call a.getB(), it will initialize B.

2. Use Setter or Field injection instead of Constructor

Constructor injection creates problems in circular dependency because both beans are required immediately.

But if we use a setter, Spring can first create the object and later “inject” the dependency.

```

@Component
class A {
    private B b;
    @Autowired
    public void setB(B b) {
        this.b = b;
    }
}

```

This way, Spring can create A without needing B instantly.

3. Refactor the design (Best Solution)

Sometimes, circular dependency is a **code smell**. It means the classes are too tightly coupled.

Example: Instead of having A depend on B and B depend on A, write a third-class ‘C’ and move shared logic , and both A & B classes depend on C.

```

@Component
class C {
    // shared logic
}

```

```

}
@Component
class A {
    private final C c;
    public A(C c) { this.c = c; }
}
@Component
class B {
    private final C c;
    public B(C c) { this.c = c; }
}

```

282 Spring 30: What is eager loading and lazy loading of beans??

By default, Spring creates all singleton beans eagerly at the time the ApplicationContext starts.

```

@Component
class EagerBean {
    public EagerBean() {
        System.out.println("Eager bean created!");
    }
}

```

When we run the app, we will see "Eager bean created!" printed immediately, even if we never used it.

Lazy loading means: Create the bean only when it is first requested (injected or accessed).

We mark it with @Lazy.

```

@Component
@Lazy
class LazyBean {
    public LazyBean() {
        System.out.println("Lazy bean created!");
    }
}

```

Now, the bean won't be created when the app starts.

It will only be created when some other bean asks for it, like:

```

@Component
class Service {
    @Autowired
    private LazyBean lazyBean; // LazyBean created only when Service is called
}

```

283 Spring 31: What is Dispatcher Servlet?

The DispatcherServlet is a core component of the Spring MVC framework, acting as the front controller for handling all incoming HTTP requests in a web application. Spring Boot automatically configures the DispatcherServlet. Its Role:

Front Controller:

It is the central entry point for all web requests. When a client sends an HTTP request, it first arrives at the DispatcherServlet.

Request Routing:

The DispatcherServlet is responsible for analyzing the incoming request and determining which specific controller and handler method should process it. This is often done based on URL patterns and annotations like @RequestMapping.

Request Delegation:

After identifying the appropriate handler, it delegates the request to that controller method for processing.

View Resolution:

Once the controller processes the request and returns a logical view name (or a direct response), the DispatcherServlet works with ViewResolvers to resolve this logical view name into an actual view (e.g., a JSP page, a Thymeleaf template).

Response Handling: It then renders the selected view and sends the final response back to the client.

Request Handling flow in Spring Boot:

Client Request: Browser sends an HTTP request : DispatcherServlet catches it.

HandlerMapping: DispatcherServlet asks: *Which controller method should handle this URL?* That is decided by HandlerMapping.

```
@GetMapping("/hello")
public String sayHello() {
    return "Hello World!";
}
```

Here, /hello is mapped to sayHello().

HandlerAdapter: Once the handler (controller method) is found, HandlerAdapter helps in executing it.

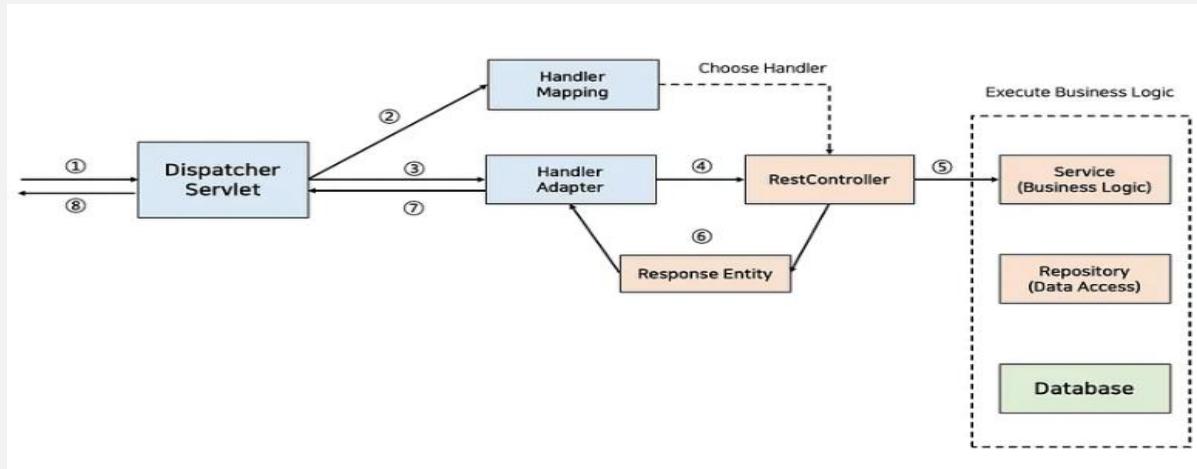
Controller: The controller contains the business logic and usually returns data or view name.

ViewResolver: If the controller returns a view name (like "home"), ViewResolver decides which actual page (like home.html or home.jsp) should be shown.

View: Finally, the view (JSP, Thymeleaf, etc.) is rendered with the model data, and the response goes back to the client.

Full Lifecycle in One Line:

Request > DispatcherServlet > HandlerMapping > HandlerAdapter > Controller > ViewResolver > View > Response



Reference: [The DispatcherServlet: The Engine of Request Handling in Spring Boot | by Lakshya Agarwal | Medium](#)

Indirect Question: What happens internally when you hit a REST API in Spring Boot?

284 Spring 32: How embedded server works in Spring boot?

In traditional Java web applications (before Spring Boot), we had to deploy the .war file into an external server like Tomcat, Jetty, or WebSphere. That meant setting up the server separately, copying the WAR into its webapps folder, and then starting it.

Spring Boot makes life easier by giving us an embedded server. This means the server (like Tomcat/Jetty/Undertow) is packaged inside the application as a dependency, so when we run the app, the server also starts automatically.

In short: the app carries its own server. We don't deploy it anywhere ; we just run the JAR, and it runs like a normal Java program.

Step-by-Step Flow

1. We add dependency in pom.xml or build.gradle

Example (pom.xml):

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

By default, this pulls Tomcat as the embedded server.

2. Spring Boot auto-configures the server

When the application starts, Spring Boot sees spring-boot-starter-web and auto-configures an embedded Tomcat (or Jetty/Undertow if we choose those).

3. DispatcherServlet is registered inside that server

Check the DispatcherServlet we discussed above. It is automatically mapped to "/", so it can handle all incoming requests.

4. Server starts with the application

If we run: mvn spring-boot:run

We will see logs like: Tomcat started on port(s): 8080

That means the app + server are bundled and started together.

5. Requests are served directly

Now we just hit <http://localhost:8080/hello> and Tomcat (running inside the JAR) handles the request.

285 Spring 33: What is ORM?

ORM stands for Object Relational Mapping.

Here is the idea:

- In Java, we work with objects (classes, objects, methods).
- In databases, we work with tables, rows, and columns.
- But they don't directly match. Example: a User object in Java is not the same as a user table in SQL.

ORM is the bridge between objects and database tables. It automatically maps:

- A class to a table
- An object to a row
- A field/variable to a column

So instead of writing raw SQL like this:

```
INSERT INTO user (id, name) VALUES (1, CodingLyf);
```

We can just do

```
User u = new User(1, "CodingLyf");
session.save(u);
```

The ORM framework will convert that into SQL internally and talk to the database.

Hibernate is the popular ORM Framework.

286 Spring 34: What is Hibernate and how it works?

Hibernate is the **most popular ORM framework** in Java. It implements ORM and makes database interaction much simpler.

Key features:

1. **Automatic Mapping**: Maps Java classes to DB tables using annotations.
2. **No need for boilerplate SQL**: We use methods like save(), update(), delete().
3. **HQL (Hibernate Query Language)**: We can query using object names instead of table names.
4. **Caching**: Improves performance by storing frequently used data in memory.
5. **Database independence**: Same code works for MySQL, PostgreSQL, Oracle, etc. We need to just change the config.

Example:

Entity Class

```
@Entity
@Table(name="users")
public class User {
    @Id
    private int id;

    private String name;

    // getters, setters
}
```

Saving the data:

```
User user = new User();
user.setId(1);
user.setName("CodingLyf");

Session session = sessionFactory.openSession();
session.beginTransaction();
session.save(user);    // Hibernate will generate INSERT query
session.getTransaction().commit();
session.close();
```

Here we just call save(). Hibernate internally does: INSERT INTO users (id, name) VALUES (1, 'CodingLyf');

287 Spring 35: What is JPA and how does it different from Hibernate?

Originally, Java developers used **JDBC** to talk to databases. But JDBC code was repetitive and messy:

```
Connection con = DriverManager.getConnection(...);
PreparedStatement ps = con.prepareStatement("INSERT INTO user VALUES (?, ?)");
ps.setInt(1, 1);
ps.setString(2, "CodingLyf");
ps.executeUpdate();
```

We had to write SQL everywhere, handle connections, manage transactions. Too much boilerplate.

To fix this, **Hibernate** came along. Hibernate is an **ORM (Object Relational Mapping) framework** that maps Java classes to DB tables. So instead of SQL, we work with objects:

```
User u = new User(1, "CodingLyf");
session.save(u); // Hibernate converts to SQL behind the scenes
```

Problem with Hibernate

Hibernate was powerful, but it was **vendor-specific**. Everyone using Hibernate was tied to Hibernate APIs.

Example:

```
Session session = sessionFactory.openSession();
session.save(user);
```

If tomorrow we wanted to switch to another ORM framework (like EclipseLink or OpenJPA), we have to rewrite a lot of code, because Hibernate APIs are not standard.

To solve this, **JPA** (Java Persistence API) was introduced by Sun/Oracle as a standard specification.

- JPA defines a set of common annotations (@Entity, @Id, @Table, etc.) and methods (persist(), remove(), find()).
- Different frameworks like Hibernate, EclipseLink, OpenJPA can implement JPA.

So, if the code uses JPA annotations and APIs, you can easily switch providers.

Spring Data JPA:

Even with JPA, we still need to write repository/DAO classes, manage transactions, etc. That is still boilerplate.

Spring Data JPA makes life easier:

- We just create a repository interface, extend JpaRepository, and Spring generates all the CRUD methods.
- Under the hood, Spring uses JPA (and Hibernate as provider by default). It many manages all the transactions.

Example with Spring Data JPA:

```
public interface UserRepository extends JpaRepository<User, Integer> { }
```

```
@Autowired
UserRepository repo;

repo.save(new User(1, "CodingLyf")); // INSERT automatically
repo.findById(1); // SELECT
repo.findByName("CodingLyf"); // Custom query by name
```

In Short:

JPA (Java Persistence API) is a specification (or interface) for ORM (Object-Relational Mapping) in Java. Hibernate is a concrete implementation of JPA.

Indirect Questions?

1. What will happen under the hood when you run `repo.save()`? (Refer Question No. 307 Spring 55)

2. How to configure Database in Spring

Using the `application.properties` file: This is the most common way to configure a database connection. We can specify the connection properties in the `application.properties` file, and Spring Boot will automatically configure the database connection.

Using the `@Configuration` class: This is a more advanced way to configure a database connection. We can create a `@Configuration` class and specify the connection properties in the class. Spring Boot will then use the `@Configuration` class to configure the database connection.

```
# Database URL
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase

# Database credentials
spring.datasource.username=myuser
spring.datasource.password=mypassword

# Database driver class
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Controls how Hibernate will handle schema generation.
# Options: none | validate | update | create | create-drop
spring.jpa.hibernate.ddl-auto=update
# Shows the SQL queries that Hibernate generates and executes.
spring.jpa.show-sql=true

# Formats the SQL queries printed in logs, so they look more readable (with line
breaks/indentation).
```

```
spring.jpa.properties.hibernate.format_sql=true
```

3. How do you make sure existing database schema matches the entity mappings.

- We need to set `spring.jpa.hibernate.ddl-auto=validate`, which makes Hibernate check if the entities match the database schema at startup.
- Better to use a migration tool like **Flyway** or **Liquibase** to version and align schema changes with entity updates.

4. What are derived query methods?

In Spring Data JPA, a derived query method is a method in a repository interface where Spring automatically generates the query based on the method name. We don't have to write `@Query` or JPQL. Spring parses the method name and creates the SQL behind the scenes.

```
public interface UserRepository extends JpaRepository<User, Long> {  
    List<User> findByLastName(String lastName); // Derived query  
}
```

Here, `findByLastName` will automatically generate a query like:

```
SELECT * FROM user WHERE last_name = ?;
```

5. Can we use Spring JPA without Spring boot?

Yes. Spring Boot is not required. Spring Boot just makes configuration easier with auto-configuration. We can use plain Spring:

With plain Spring, we need to configure `DataSource`, `EntityManagerFactory`, `TransactionManager` manually in XML or Java config.

6. How does Spring boot simplify the data access layer implementation?

- It auto-configures essential settings like data source and JPA/Hibernate based on the libraries present in the class-path. it reduces the manual set up.
- It also provides and Repository support like `JpaRepository`, enabling easy CRUD operations without the need for boilerplate code.
- Additionally, Spring boot can automatically, initialize database schemas and seeding process using scripts.
- Spring boot can automatically initialize database schemas and seed data using Scripts. It integrates smoothly with various databases and ORM technologies and translates SQL exceptions into Spring's data access exceptions, providing a consistent and simplified error handling.

288 Spring 36: JDBC vs Hibernate vs JPA?

JDBC (Java Database Connectivity)

- This is the lowest level API to interact with the database.
- We write SQL queries manually, set parameters, execute them, and handle ResultSet.
- Very flexible, but a lot of boilerplate code like opening connections, closing them, handling exceptions.

```
Connection con = DriverManager.getConnection(url, user, pass);
PreparedStatement ps = con.prepareStatement("SELECT * FROM student WHERE id=?");
ps.setInt(1, 1);
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    System.out.println(rs.getString("name"));
}
```

Hibernate

- Hibernate is an ORM (Object Relational Mapping) framework built on top of JDBC.
- Instead of writing SQL manually, we work with objects, and Hibernate translates them into SQL behind the scenes.
- It handles connection management, caching, lazy loading, and relationships (@OneToMany, @ManyToOne).
- Provides HQL (Hibernate Query Language) and Criteria API for queries.

JPA (Java Persistence API)

- JPA is not a framework but a specification (a set of rules/interfaces).
- It defines how Java objects should map to database tables, but doesn't provide implementation.
- Frameworks like Hibernate, EclipseLink, OpenJPA implement JPA.

Hibernate vs JDBC

Hibernate converts Checked Exceptions to Runtime exceptions, In JDBC we need to handle SQLException

Hibernate supports caching, where JDBC doesn't.

Hibernate has inbuilt transaction support, where In JDBC we have manage explicitly

289 Spring 37: How do you use @Entity, @Table, @Column in JPA?

@Entity

- Marks a Java class as a JPA entity (i.e., a table in the database).
- Without @Entity, JPA won't know that this class should be mapped to a database table.

```
import jakarta.persistence.*;
@Entity // This class will map to a table
```

```
public class User {
    @Id // Primary key
    private int id;
    private String name;
}
```

Here User class is nothing but User table.

@Table

- In JPA By default, the class name will be the table name (as we seen above)
- But if we want to customize the table name, we will have to use @Table.

```
@Entity
@Table(name = "users") // Map to 'users' table instead of 'User'
public class User {
    @Id
    private int id;
    private String name;
}
```

Now this class maps to the users table in the DB.

@Column

- By default, column name is nothing but the field name in the class.
- If we want to customize column name, length, nullability, uniqueness, etc., use @Column.

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @Column(name = "user_id") // custom column name
    private int id;

    @Column(name = "full_name", length = 50, nullable = false, unique = true)
    private String name;
}
```

This means:

- id maps to column user_id
- name maps to column full_name, max length = 50, cannot be null, must be unique

Indirect Questions:

1. How do you mark a column as not nullable or unique?

To make sure JPA saves only unique values, we define a unique constraint on the entity field. This can be done with @Column(unique = true).

Use @Column(nullable = false, unique = true)

2. How does JPA decide table/column names if you don't give any?

JPA uses the class name as table name and field names as column names.

3. What happens if you don't give @entity to class?

If we don't give `@Entity` on a class, JPA will not treat it as a database entity, it will just behave like a normal Java class means no table will be created.

4. Can we make `@Entity` as final class?

No, we should not make an `@Entity` class final

- JPA providers (like Hibernate) often use proxy classes at runtime to enable features like lazy loading.
- To create these proxies, they extend the entity class.
- If the entity is final, it cannot be subclassed, so Hibernate can't create the proxy which breaks lazy loading and other internal mechanisms.

5. Can we make `@Entity` as Abstract class?

Yes, it can be abstract, and other entities can extend it

290 Spring 38: What are best practices or rules while creating the Entity?

Use a Primary Key (`@Id`)

Every entity must have an identifier. Always define a primary key with `@Id`.

Avoid Final Classes

Entities should not be final because JPA providers like Hibernate use proxies (subclassing) for lazy loading.

Provide a No-Arg Constructor

Entities must have a default (public or protected) no-arg constructor for JPA to instantiate them.

Make Fields Private with Getters/Setters

Keep fields private and expose them via getters/setters to maintain encapsulation.

Don't Use Static or Transient Fields for Persistent State

JPA ignores them, so use `@Transient` explicitly for non-persistent fields.

Prefer Wrapper Types for Nullable Columns

Example: use `Integer` instead of `int` if the column can be null.

Use Meaningful Table and Column Names

Override defaults using `@Table(name="...")` and `@Column(name="...")` for clarity.

Override Equals and HashCode carefully

Override equals and hashCode based on primary keys. Avoid including lazy-loaded relationships to prevent performance issues.

Reference: [Entity Class best practices and rules | by Sumit sharma | Medium](#)

291 Spring 39: What are the different types of relationships in JPA and explain `@OneToOne` with example

Database relationships are fundamental concepts in relational database design, allowing data to be stored efficiently and retrieved accurately.

Different types are: `OneToOne`, `OneToOne`, `ManyToOne`, and `ManyToMany`.

Reference: [Types of Relationship in DBMS: Explained with Examples](#)

@OneToOne Relationship

This is when one entity is directly related to exactly one other entity. Like a Person and their Passport means each person has one passport, and each passport belongs to one person.

```
@Entity
public class Person {
    @Id
    private Long id;

    @OneToOne
    @JoinColumn(name = "passport_id")
    private Passport passport;
    // ... other fields
}

@Entity
public class Passport {
    @Id
    private Long id;

    @OneToOne(mappedBy = "passport")
    private Person person;
    // ... other fields
}
```

The **@JoinColumn** annotation is placed on the owning side of the relationship. It explicitly defines the foreign key column in the database table that links to the primary key of the associated entity.

mappedBy declares that the current entity is not responsible for managing the foreign key and that another entity (the one specified in mappedBy) handles the relationship's persistence.

From the Example:

```
@OneToOne
@JoinColumn(name = "passport_id")
private Passport passport;
```

This means:

- In the person table, there will be a column `passport_id`.
- That column acts as a foreign key pointing to the `Passport` table's `id`.
- So, the relationship is stored in the `person` table.

In `Passsport` class we wrote

```
@OneToOne(mappedBy = "passport")
private Person person;
```

`mappedBy = "passport"` tells Hibernate: "Don't create a new join column here. Just look at the field `passport` in the `Person` entity, because that is where the relationship is actually managed."

So passport table does not get an extra column like person_id.

This is how table looks like.

| Person Table | | |
|--------------|------|-------------|
| id | name | passport_id |
| 1 | John | 101 |
| 2 | Alex | 102 |

| Passport Table | | |
|----------------|-----------------|------------|
| id | passport_number | issue_date |
| 101 | A12345 | 2020-01-01 |
| 102 | B67890 | 2021-02-01 |

Here:

- passport_id in Person points to id in Passport.
- Only Person has the foreign key column, because it is the owning side.

Java code:

```
Passport passport = new Passport();
passport.setId(101L);
passport.setPassportNumber("A12345");

Person person = new Person();
person.setId(1L);
person.setName("John");
person.setPassport(passport);

// save person : will also link passport_id
```

Interview Tip: Always explain these relations with examples and dummy tables. So that it will be easy for the explanation.

292 Spring 40: Explain @OneToMany or @ManyToOne with example

@OneToMany Relationship

This represents a one-to-many connection. For example, a department can have many Employees, but each Employee belongs to only one Department.

```
@Entity
public class Employee {
    @Id
    private Long id;
```

```

@ManyToOne
@JoinColumn(name = "department_id")
private Department department;
// ... other fields
}

@Entity
public class Department {
    @Id
    private Long id;

    @OneToMany(mappedBy = "department")
    private List<Employee> employees = new ArrayList<>();
    // ... other fields
}

```

Employee owns the relationship because it has `@JoinColumn`.

Department is the inverse side because it uses `mappedBy`.

In the employee table, a column `department_id` will be created, that column is a foreign key referencing `department.id`.

The department table will **not** get any `employee_id` column or extra join table.

Department Table

| id | name |
|----|------|
| 10 | HR |
| 20 | IT |

Employee Table

| id | name | department_id |
|-----|---------|---------------|
| 101 | Alice | 10 |
| 102 | Bob | 10 |
| 103 | Charlie | 20 |

Here:

- Employees **Alice** and **Bob** belong to Department **HR (id=10)**.
- Employee **Charlie** belongs to Department **IT (id=20)**.
- The link is stored in `employee.department_id`.

Reference: [ONE TO MANY mapping in Spring JPA | by Mohammed Youssef | Medium](#)

@ManyToOne Relationship

This is the inverse of OneToMany. In the example above, the Employee side uses @ManyToOne because many employees can belong to one department.

Reference: [Many To One mapping in Spring Boot JPA \(unidirectional\) | by Vinotech | Medium](#)

293 Spring 41: Explain @ManyToMany with example

This is when multiple entities relate to multiple other entities. Like Students and Courses , means a student can take many courses, and a course can have many students.

Student is the owning side because it has @JoinTable.

Course is the inverse side because it uses mappedBy.

The **@JoinTable** annotation in Hibernate (part of JPA) is used to define the join table used in the mapping of associations, particularly for many-to-many relationships

```
ManyToMany  
@JoinTable(  
    name = "student_course",  
    joinColumns = @JoinColumn(name = "student_id"),  
    inverseJoinColumns = @JoinColumn(name = "course_id"))  
private Set<Course> courses;
```

This code means,

A new join table called student_course will be created.

It has two foreign key columns:

- student_id > references student.id
- course_id > references course.id

mappedBy = "courses" tells Hibernate: "*Don't create another join table here. Just use the join table defined in Student.*"

Without mappedBy, Hibernate would create **two join tables**, which is wasteful.

Student Table

| id | name |
|----|-------|
| 1 | John |
| 2 | Alice |

Course Table

| id | title |
|-----|---------|
| 100 | Math |
| 200 | Physics |

Student_Course Join Table

| student_id | course_id |
|------------|-----------|
| 1 | 100 |
| 1 | 200 |
| 2 | 100 |

Reference:

Theory: [Spring Boot Many To Many Relationship | by HK | Medium](#)

Example: [Java Spring Boot — Many to Many Relationship | by Zübeyr Bahadır Damar | Medium](#)

Scenario Question:

1. Supposed One employee has multiple managers, and one manager can have multiple employees.
How many tables do you use to save data?

We need **3 tables**:

1. **Employees**: stores employee details (employee_id, name, etc.)
2. **Managers**: stores manager details (manager_id, name, etc.)
3. **Employee_Manager**: a junction (bridge) table that connects employees and managers.
 - o It has at least two columns: employee_id and manager_id, both as foreign keys.

294 Spring 42: What is cascading in JPA and when would you use it?

In JPA, **cascading** means that an operation done on a parent entity automatically flows (or "cascades") to its child entities.

Think of it like this: if we delete a department, and we want all Employees under that Department to also get deleted automatically, we set cascade rules. Without it, we need to delete each Employee manually.

It saves us from writing extra code and keeps parent-child relationships consistent.

Different types of cascade types: ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH

Explanation with Relationships:

CascadeType.PERSIST:

Use Case: When a new parent entity is persisted, its newly created child entities should also be persisted.

Relationship Example (One-to-Many): A Customer entity has a list of Order entities. When a new Customer is saved, all new Order entities associated with that customer should also be saved.

PERSIST: Save parent also saves child.

REMOVE: Deleting parent deletes child.

MERGE: Updating parent updates child.

REFRESH: Refresh parent refreshes child.

ALL: Applies all above.

Reference: [JPA/Hibernate Cascade Types. Cascading relationships are designed to... | by Himani Prasad | Medium](#)

Indirect Questions:

1. If I delete a department, will all its Employees also get deleted automatically?

By default, deleting a parent does **not** delete children.

If we want that, we need to explicitly add cascade = CascadeType.REMOVE or CascadeType.ALL on the relationship.

2. Why does saving a parent entity not automatically save the child entities?

If cascade is missing, we must save the children separately.

With cascading, saving the parent pushes the persist operation down to its children.

3. Can you give me an example where you should NOT use CascadeType.ALL?

For example, deleting a customer should not delete all their Orders, since Orders might be needed for reporting or audit.

So, we will avoid CascadeType.REMOVE in that relationship.

4. What happens if you delete parent entity when cascading type is all?

Child entities will also be deleted.

295 Spring 43: How does eager loading and lazy loading works in JPA?

Whenever we fetch an entity from the database, it may have relationships (like Department has Employees).

The question is: Should JPA fetch everything immediately (Eager) or Will it fetch only when we actually need it (Lazy)?

Eager Loading: Related entities are loaded immediately along with the parent, even if we don't use them.

```
@Entity
public class Department {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @OneToMany(mappedBy = "department", fetch = FetchType.EAGER)
    private List<Employee> employees;
}
```

If we run: `Department dept = entityManager.find(Department.class, 1L);`

JPA will fetch the Department and also all its Employees in the same query or with extra queries. So even if we don't need Employees, they are still loaded.

Lazy Loading: Related entities are loaded only when accessed. JPA uses a proxy (like a placeholder object).

```
@Entity
public class Department {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @OneToMany(mappedBy = "department", fetch = FetchType.LAZY)
    private List<Employee> employees;
}
```

If we run `Department dept = entityManager.find(Department.class, 1L);` only the Department is fetched. Employees are not.

But when we do `dept.getEmployees();` now JPA fires another query to load Employees.

Point to remember: If we try to access Employees after the Hibernate session is closed (like outside a transaction), we will get `LazyInitializationException`.

Indirect questions

1. Why am I getting `LazyInitializationException` when accessing a collection after transaction is closed?
Because it was marked LAZY, and the data wasn't loaded before the session closed.
2. Which is the default fetch type for `@OneToOne` and `@ManyToOne`?

- `@OneToMany` : LAZY (default)
- `@ManyToOne` : EAGER (default)

3. How would you optimize performance if you always need parent + child data together?

Use `fetch = FetchType.EAGER` or better, use **JPQL JOIN FETCH** to control it at query level.

Interview Tip: We often prefer Lazy and control loading explicitly with JOIN FETCH in queries.

4. How does lazy loading breaks in rest Api, how do you solve it properly?

In JPA/Hibernate, associations are often marked as LAZY to avoid unnecessary data fetching. For example:

```
@Entity
class User {
    @OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
    private List<Order> orders;
}
```

How it breaks:

- In a Spring REST controller, we often return entities directly as JSON.
- When Jackson tries to serialize User, it accesses orders.
- If the Hibernate session is already closed (because the transaction ended after service layer), we get: `org.hibernate.LazyInitializationException`

Fix:

Use DTOs to Map the correct data before returning.

Use join fetch

296 Spring 44: Explain Repositories, like CrudRepository, JpaRepository, and PagingAndSortingRepository?

`CrudRepository`, `JpaRepository`, and `PagingAndSortingRepository` are interfaces in Spring Data JPA that offer different levels of functionality for data access. They form an inheritance hierarchy, with each extending the previous one to add more features:

CrudRepository:

- This is the most basic interface.
- It provides fundamental Create, Read, Update, and Delete (CRUD) operations. It includes methods like `save()`, `findById()`, `findAll()`, and `delete()`.
- If only basic data manipulation is required, `CrudRepository` is sufficient.

PagingAndSortingRepository:

- This interface extends `CrudRepository` and adds capabilities for pagination and sorting.

- It introduces methods like findAll(Pageable pageable) for retrieving data in pages and findAll(Sort sort) for sorting results.
- Use this interface when we need to display large datasets in a paginated or sorted manner.

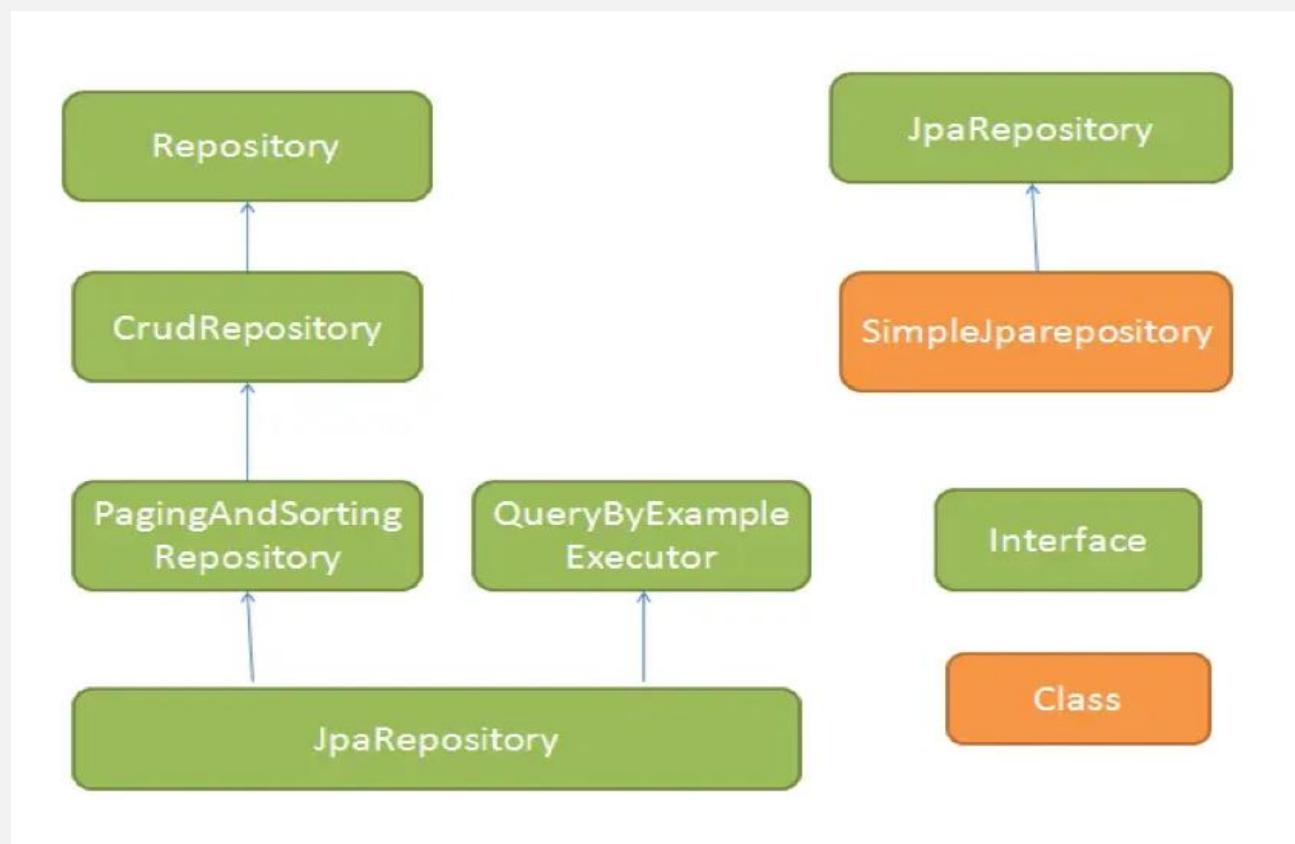
JpaRepository:

This is the most comprehensive interface, extending PagingAndSortingRepository and thus inheriting all its functionalities, as well as those of CrudRepository.

JpaRepository adds JPA-specific features such as:

- **Batch operations:** Methods for deleting entities in bulk.
- **Flushing the persistence context:** Methods like flush() to synchronize the in-memory changes with the database.
- **Support for custom queries:** Facilitates the definition of custom queries using annotations or method names.

Repository hierarchy:



Indirect Questions:

1. If you only need simple save and find operations, which repository will you choose?

Use CrudRepository.

2. How would you fetch 20 students at a time from the database?

Use PagingAndSortingRepository with PageRequest.

3. Which repository allows batch deletion or flushing?

Use JpaRepository.

4. If you extend JpaRepository, do you still get CrudRepository methods?

Yes, because it is a hierarchy.

297 Spring 45: What is JPQL and how does it different from HQL?

JPQL (Java Persistence Query Language) queries are written using the entity classes and their attributes instead of raw SQL queries.

```
//Entity: Employee (mapped to employees table)
String jpql = "SELECT e FROM Employee e WHERE e.department.name = :deptName";
List<Employee> employees = entityManager.createQuery(jpql, Employee.class)
.setParameter("deptName", "IT")
.getResultList();
```

Here Employee is the entity class, and department.name is a field in the object, not the DB column.

HQL (Hibernate Query Language) is a powerful object-oriented query language used in Hibernate ORM. HQL is a superset of the JPQL. A JPQL query is a valid HQL query, but not all HQL queries are valid JPQL queries.

1. If I switch from Hibernate to EclipseLink, will my queries still work?

Only JPQL queries are guaranteed to work. HQL has Hibernate-specific stuff.

2. What's the difference between using EntityManager.createQuery() and Session.createQuery()?

The first is JPQL (JPA standard), the second is HQL (Hibernate-specific).

3. Can JPQL query database tables directly?

No, it queries entities and their mappings, not raw tables. For raw SQL, we will use @NamedNativeQuery or createNativeQuery().

298 Spring 46: What is @Query (Custom Queries) and why should we use them??

By default, Spring Data JPA creates queries automatically just by looking at the method names.

Example: List<Employee> findBySalaryGreaterThan(double salary);

Spring generates the query behind the scenes.

But sometimes we need more control, maybe a complex join, or a query that can't be expressed with method naming. That is when we use **@Query**.

@Query allow us write the own queries directly inside the repository method.

Two ways to use it

1. JPQL with @Query (works on entities and fields)

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
    @Query("SELECT e FROM Employee e WHERE e.salary > :salary")
    List<Employee> findEmployeesWithHighSalary(@Param("salary") double salary);
}
```

Here, notice we wrote the query using entity class (Employee) and its field (salary), not database table/column.

2. Native SQL with @Query

Native queries are queries written in **plain SQL**

Why should we use them?

Complex queries

Sometimes JPQL can't express what we want, like database-specific functions, window functions, or advanced joins.

Example:

```
SELECT e.* , ROW_NUMBER() OVER (ORDER BY e.salary DESC) as rank
FROM employees e
```

This is not possible with JPQL, but we can do it with a native query.

Performance tuning

Native queries useful when we need to write highly optimized SQL, sometimes better than what JPA would generate for us.

Limitations of native queries:

- Native queries are tied to the database (Oracle, MySQL, PostgreSQL, etc.). But JPQL is database-independent.
- SQL is harder to maintain if the DB schema changes.

Indirect Questions:

1. How Native queries create problems in Spring?

Native queries are specific to a database; switching DBs may break the query.

Changes in table structure require updating raw SQL manually

299 Spring 47: What is @NamedQuery and @NamedNativeQuery?

Imagine we want to always get a list of users who are “admins” from a database.

Without @NamedQuery, we need to write the same query in several places in the code, which is repetitive.

@NamedQuery is an annotation in JPA (Java Persistence API) that allows to define a reusable query. This query is named (we give it a name) and defined once in the entity class. Later, we can reuse that query by referencing the name in the code.

- Named Queries are static, meaning once they are defined, they cannot be modified at runtime.

- The `@NamedQuery` annotation is placed at the top of the relevant Entity Class and consists of two parameters:
- name: A unique identifier used to reference the query.
- query: The JPQL statement (JPQL query) that will be executed at runtime.

Example:

```
@Entity
@NamedQuery(
    name = "User.findByRole", // Name of the query
    query = "SELECT u FROM User u WHERE u.role = :role" // The actual JPQL query
)
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String role;

    // Getters and setters...
}
```

- `name = "User.findByRole"` is the **name** of the query. We will reference this name later to execute the query.
- `query = "SELECT u FROM User u WHERE u.role = :role"` is the actual query. It is written in JPQL (Java Persistence Query Language), which is similar to SQL

Reference: [Spring Data JPA @NamedQuery and @NamedQueries Example](#)

We can also use `@NamedNativeQueries` Read here : [Spring Data JPA @NamedNativeQuery and @NamedNativeQueries Example](#)

300 Spring 48: What is the use of `@Transactional`?

In Spring (and JPA), `@Transactional` is used to manage database transactions automatically.

A transaction means a group of operations that should either:

- all succeed together (commit), or
- all fail together (rollback).

Without transactions, if something fails halfway, we might end up with corrupted data in the database.

Think of a banking example:

- We transfer money from Account A to Account B.
- Step 1: Deduct from Account A.
- Step 2: Add to Account B.

If step 1 succeeds but step 2 fails, out money is gone!

That is why both steps should be inside one transaction; either both happen, or both fail.

The `@Transactional` annotation in Spring is used to manage database transactions in a declarative manner. It simplifies transaction management by automatically handling the beginning, committing, or rolling back of transactions

```
@Service
public class BankService {

    @Autowired
    private AccountRepository accountRepo;

    @Transactional
    public void transferMoney(Long fromId, Long toId, double amount) {
        Account from = accountRepo.findById(fromId).get();
        Account to = accountRepo.findById(toId).get();

        from.setBalance(from.getBalance() - amount);
        to.setBalance(to.getBalance() + amount);

        accountRepo.save(from);
        accountRepo.save(to);

        // if something fails here, both updates rollback automatically
    }
}
```

With `@Transactional`, if anything inside fails, **both account updates are rolled back**.

Points to remember:

1. Use `@Transactional` on service layer methods, where business logic happens (not usually on repository methods).
2. In Spring, transactions only work when a method is called from outside the class itself, because Spring uses proxies to manage transactions.
3. By default, rollback only happens for **runtime exceptions**. If we want rollback for checked exceptions too, we must configure it:

```
@Transactional(rollbackFor = Exception.class)
```

Interview Tip:

- Mention real world scenario while explaining about the transactional.
- Use `@Transactional` to ensure a group of DB operations succeed or fail together, maintaining consistency.
- Mention Spring automatically handles commit and rollback, so you don't manage transactions manually.
- It can be applied at method or class level depending on the scope of the transaction.
- Real-world example: In an e-commerce service, deducting stock and creating an order are in the same transactional method; if order creation fails, stock deduction is rolled back automatically.

Indirect Question:

1. What happens if a Spring bean is @Transactional, but the method is called from inside the same class?

Answer: Self-invocation bypasses the proxy, so the transaction is ignored

Reference: [Discover how to use the @Transactional annotation in Spring Boot | by Tharindu Dulshan | Medium](#)

2. What is the best place to keep @Transactional Annotation and why?

In a Spring application, @Transactional is used to make sure that a group of database operations happen as one complete unit of work. If something goes wrong, all the changes are rolled back, so the data stays consistent.

Now the question is: where should you place it?

- **Controller layer**

Controllers should only handle requests and responses. They aren't supposed to know how transactions work, so putting @Transactional here mixes responsibilities.

- **Repository (DAO) layer**

Repositories are focused on single database operations like saving, finding, or deleting one entity. If we mark them transactional, we only control one small piece of the puzzle, but not the bigger business process.

- **Service layer (Best place)**

Services contain the actual business logic. They often call multiple repositories, maybe even across different entities. By putting @Transactional at the service method level, we ensure that all those operations run inside a single transaction. If anything fails, the whole set of operations rolls back.

Example

Imagine an e-commerce app:

1. When a customer places an order, the system saves the order details.
2. Then it deducts money from the customer's balance.
3. Finally, it updates the product stock.

All three must succeed together. If the payment fails but the stock is already reduced, you end up with inconsistent data.

That is why the service method should be transactional:

```
@Service
public class OrderService {

    @Transactional
    public void placeOrder(Order order) {
        orderRepository.save(order);
        paymentRepository.debit(order.getPayment());
        productRepository.updateStock(order.getProduct());
    }
}
```

Here, if payment fails, the saved order and stock update are rolled back automatically.

3. What type of proxy @Transactional creates?

When we put @Transactional on a method or class, Spring needs a way to wrap that code so it can start, commit, or roll back a transaction. It does this with proxies. Now, which proxy type gets used depends on the class.

1. JDK Dynamic Proxy

If the class implements an interface, Spring will by default create a JDK dynamic proxy. This proxy implements the same interface and intercepts calls to add transactional behavior.

Here, UserServiceImpl implements UserService. Spring creates a JDK proxy that also implements UserService. All calls go through that proxy.

```
public interface UserService {  
    void saveUser(User user);  
}  
  
@Service  
@Transactional  
public class UserServiceImpl implements UserService {  
    public void saveUser(User user) { ... }  
}
```

2. CGLIB Proxy

If the class does not implement any interface, Spring falls back to CGLIB, which creates a subclass at runtime. That subclass overrides the methods to add transaction logic.

```
@Service  
@Transactional  
public class OrderService {  
    public void placeOrder(Order order) { ... }  
}
```

Since OrderService has no interface, Spring uses CGLIB to generate something like OrderService\$\$EnhancerBySpringCGLIB, which wraps the real logic.

301 Spring 49: What is propagation in Transaction?

When we use @Transactional in Spring, we are telling Spring how the transaction should behave. But here is the thing, sometimes one method with @Transactional calls another method that also has @Transactional.

Now the big question is:

Should both methods share the same transaction, or should the second method start a new transaction?

That is what **Propagation** decides. It is basically the rule for how transactions should behave when one transactional method calls another.

Let's say we have two services:

```
@Service
public class OrderService {

    @Transactional
    public void placeOrder() {
        // some DB operations for order
        paymentService.processPayment();
    }
}
```

```
@Service
public class PaymentService {

    @Transactional
    public void processPayment() {
        // DB operations for payment
    }
}
```

Now, the question is:

- When placeOrder() calls processPayment(), do both run in one single transaction?
- Or should processPayment() start its own separate transaction?

This is controlled by Propagation.

Propagation Types:

REQUIRED (default)

- If there's already a transaction, join it.
- If not, create a new one.
- Example: Both order and payment will run in **one transaction**. If payment fails, the whole order fails.

```
@Transactional(propagation = Propagation.REQUIRED)
```

REQUIRES_NEW

- Always start a new transaction, even if one already exists.
- Example: Payment starts in a **new transaction**. So even if order fails later, payment can still be committed.

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
```

NESTED

- Runs inside the parent transaction but can roll back independently.
- Example: Payment failure will roll back payment only, but not necessarily the whole order.
 @Transactional(propagation = Propagation.NESTED)

SUPPORTS

- If a transaction exists, join it. If not, just run without a transaction.

NOT_SUPPORTED

- Runs the method non-transactionally and suspends any existing transaction.

MANDATORY

- Must run inside an existing transaction. If none exists throw exception.

NEVER

- Must run without a transaction. If one exists throw exception.

Let's take one simple example and apply all the 4 cases.

Imagine a **BankService** class with two methods: transferMoney() and sendEmailNotification().

```
@Service
public class BankService {

    @Transactional
    public void transferMoney() {
        // deduct from account A
        // add to account B
        sendEmailNotification();
    }

    @Transactional
    public void sendEmailNotification() {
        // send email to user
    }
}
```

Q1. If two methods are annotated with @Transactional

- When transferMoney() calls sendEmailNotification(), Spring reuses the same transaction (by default Propagation.REQUIRED).
- Both DB updates and email logs (if stored in DB) are part of the same transaction.
- If email fails then the whole transaction rolls back.

2. If transferMoney ()method is not annotated with @Transactional

- transferMoney() runs without any transaction.

- If `sendEmailNotification()` is annotated with `@Transactional`, Spring starts a new transaction only for that method.

3. If `sendEmailNotification()` method is NOT supported

```
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public void sendEmailNotification() {
    // send email without any transaction
}
```

If this method is called from a transactional method, Spring suspends the existing transaction and runs this code without transaction.

No error. It simply says "I don't want a transaction here."

4. If `transferMoney()` method is in transaction but you don't want `sendEmailNotification()` method in transaction

If `transferMoney()` is transactional but we want `sendEmailNotification()` to run outside the transaction, then:

- With **NOT_SUPPORTED**, Spring will suspend A's transaction, run B without any transaction, then resume A's transaction. Everything works fine.
- With **NEVER**, Spring will throw an exception because A already has an active transaction. B refuses to run at all.

So, in this case, `NOT_SUPPORTED` is the right choice, because we are saying: "*I want this method to run, but I don't want it wrapped in a transaction.*"

`NEVER` is useful only when we want to enforce a hard rule means "this method must never run if there's an active transaction, otherwise fail."

302 Spring 50: How transaction works under the hood?

When Spring sees a method annotated with `@Transactional`, it doesn't directly execute the method. Instead, it creates a **proxy** (a wrapper around the class).

This proxy is responsible for:

- Opening a transaction (start)
- Calling the actual method
- Deciding to **commit** or **rollback** the transaction based on the result

```
@Service
public class PaymentService {
    @Transactional
    public void makePayment() {
        // 1. deduct money
        // 2. update account balance
        // 3. save transaction log
    }
}
```

```
}
```

What happens when makePayment() is called?

1. Proxy Intercepts Call

When we call paymentService.makePayment().

The proxy catches this call before our method runs.

2. TransactionManager Starts Transaction

Proxy asks PlatformTransactionManager (usually DataSourceTransactionManager) to start a transaction.

It sets autoCommit=false on the JDBC connection.

3. Our Method Executes

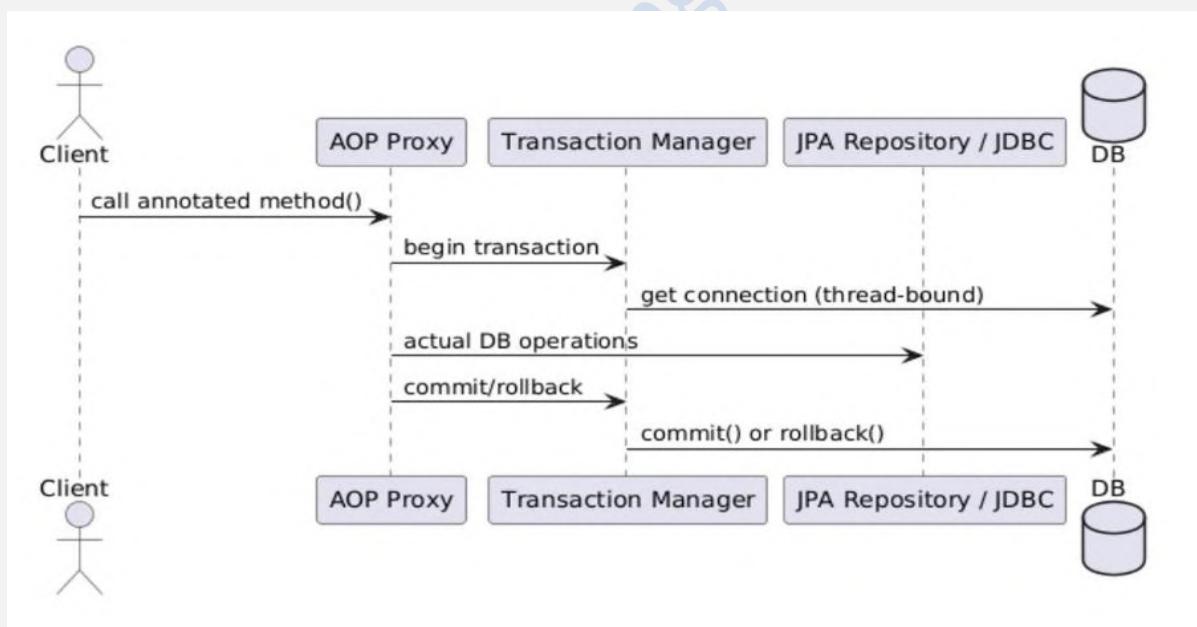
Now the actual method runs (deduct money, update balance, etc.).

All DB operations use the same connection and are part of one transaction.

4. Commit or Rollback

If our method finishes without exception > Proxy tells TransactionManager to commit.

If our method throws a RuntimeException > Proxy tells TransactionManager to rollback.



Reference: [Spring Transaction Internals — What Really Happens Under the Hood | by Arvind Kumar | Medium](https://medium.com/@arvindkumar_1111/spring-transaction-internals-what-really-happens-under-the-hood-113a2a2a2a2a)

303 Spring 51: What is N+1 problem how do you fix it?

Imagine we want to fetch all students and their courses from the database.

- First, we run 1 query to get all students.
- Then, for each student (say there are N students), we run 1 query per student to fetch their courses.

That means: 1 (main query) + N (extra queries) = N+1 queries.

So instead of just 2 queries (students + their courses), we end up making dozens or hundreds of queries, which slows everything down badly.

```
List<Student> students = entityManager
.createQuery("SELECT s FROM Student s", Student.class)
.getResultList();

for (Student s : students) {
// This triggers another query for each student (Lazy loading)
System.out.println(s.getCourses().size());
}
```

Example of N+1 problem in JPA/Hibernate

If there are 100 students, this will make:

- 1 query to fetch students
- 10000 extra queries for their courses

Total = 10001 queries (instead of just 2).

How to fix N+1 problem?

Use Join Fetch: It tells Hibernate to fetch related data in a single query. Instead of fetching students only, fetch students and their courses in a single query.

```
public interface StudentRepository extends JpaRepository<Student, Long> {

    @Query("SELECT s FROM Student s JOIN FETCH s.courses")
    List<Student> findAllStudentsWithCourses();
}
```

Use Entity Graphs (alternative way)

If we don't want to hardcode JOIN FETCH in queries, we can use entity graphs.

Reference: [JPA EntityGraphs: A Solution to N+1 Query Problem | by Thameena S | Geek Culture | Medium](#)

304 Spring 52: Difference Between JOIN and JOIN FETCH?

JOIN:

A SQL join combines data from two or more tables using a common column. In HQL or JPQL, a JOIN adds the joined table's data to the SQL query. But this doesn't mean the related objects in the code are loaded right away. If they are set to load lazily, they will only load when we actually use them, which can cause extra queries (the N+1 problem) if we access them many times.

JOIN FETCH:

JOIN FETCH in JPA/Hibernate is like a normal join, but it also forces Hibernate to load the related entities immediately in the same query, avoiding lazy loading and the N+1 problem.

Use JOIN when we only need the association for filtering, but don't actually need the associated entities in memory. Example: get students who have at least one course.

Use JOIN FETCH when we know we will access the associated entities and want to avoid the **N+1 problem** by loading everything in one query. Example: get students and load their courses right away.

Reference: [Spring JPA: when to use “Join Fetch” | by Dev INTJ Code | Javarevisited | Medium](#)

305 Spring 53: How does locking works in Spring JPA?

When multiple users/transactions try to read and update the same data at the same time, there is a risk of conflicts. To handle JPA provides the locking mechanisms. Transaction locks are crucial for managing concurrent access to the data in the database. It ensures data integrity and consistency when multiple transactions interact with the same data.

In Spring Data JPA, two primary types of locks are used: Pessimistic Locking and Optimistic Locking

Optimistic Locking assumes that conflicts are rare and allows the concurrent transactions to proceed without locking the data. It uses the @version field to detect the conflicts and ensure that updates do not overwrite each other.

```
@Entity
public class Product {
    @Id
    private Long id;

    private int stock;

    @Version
    private int version; // JPA uses this to detect conflicts
}
```

How it works:

1. Transaction A reads a row (version = 1).
2. Transaction B updates the same row then version becomes 2.
3. Transaction A tries to update with version = 1 so Hibernate detects mismatch and throws OptimisticLockException.

Best when conflicts are rare because it avoids unnecessary database locks and scales well.

Pessimistic Locking assumes that conflicts will occur and locks the data to prevent the other transactions from accessing it concurrently.

How it works?

1. Acquire Lock:

- When the transaction reads a record with the pessimistic lock, the database acquires the lock on that record.
- The lock can be of the different types, such as PESSIMISTIC_READ (prevents other transactions from writing) or PESSIMISTIC_WRITE (prevents other transactions from reading or writing).

2. Block Other Transactions

- Other transactions trying to access the same record will be blocked until the lock can be released. This prevents data inconsistency caused by the concurrent modifications.

3. Release Lock

- The lock is held for the duration of the transaction and it can be automatically released when the transaction commits or rolls back.

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
  
    @Lock(LockModeType.OPTIMISTIC)  
    Optional<Product> findById(Long id);  
  
    @Lock(LockModeType.PESSIMISTIC_WRITE)  
    Optional<Product> findByName(String name);  
}
```

Examples:

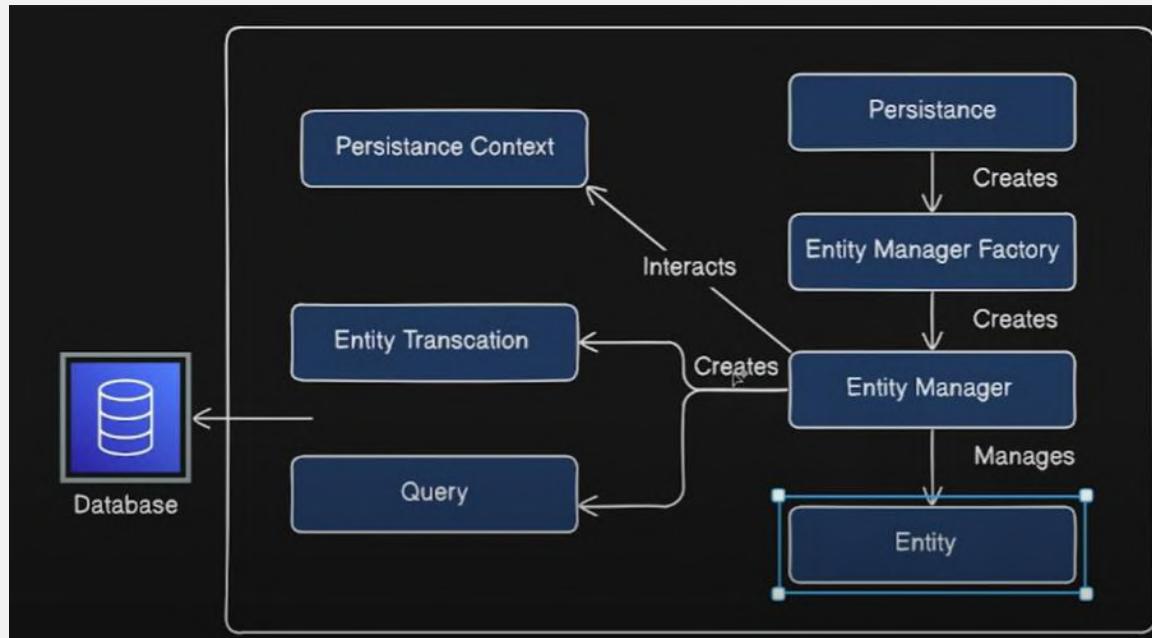
Optimistic Locking: Imagine an online shopping website. Thousands of people are viewing products, but only a few actually place orders at the same time. If two users try to buy the last piece of a product, both can proceed without blocking each other. But at the moment of update, Hibernate checks a @Version column. If the version in the database has changed, it throws an OptimisticLockException.

Pessimistic locking: Think about a bank account. Two people try to withdraw money from the same account at the same time. If both see ₹1000 and withdraw ₹800 each, the account would go negative. That is dangerous.

With pessimistic locking, the first transaction locks that row in the database. The second transaction has to wait until the first one commits or rolls back. That way, we never get inconsistent balances.

Reference: [Pessimistic and Optimistic Locking in JPA, Spring Boot | by Berat Yesbek | Medium](#)

306 Spring 54: Explain JPA Architecture?



1. Database

This is the actual relational database (like MySQL, PostgreSQL, Oracle) where the tables and data live. JPA's job is to talk to this database but using object, so we don't directly write a lot of SQL.

2. Entity

- An **Entity** is just a normal Java class mapped to a table in the database using `@Entity`.
- Each object of that class represents a row in the table.

```
@Entity
public class Student {
    @Id
    private Long id;
    private String name;
    private String email;
}
```

Here, the `Student` class is an entity, mapped to a table `Student`.

3. Persistence

- The Persistence class is a bootstrap helper provided by JPA.
- Its job is to create an EntityManagerFactory.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("my-persistence-unit");
```

4. EntityManagerFactory

- It is heavyweight object created only once for the whole application.
- It is Responsible for creating multiple EntityManagers.
- Think of it like a "factory" that handles database managers.

5. EntityManager

- This is the core interface we use in JPA.
- Manages all the entity operations like insert, update, delete, query)
- It interacts with the Persistence Context and Entity Transactions.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

Student s = new Student();
s.setId(1L);
s.setName("Ramesh");
s.setEmail("ramesh@test.com");

em.persist(s); // inserts into DB
em.getTransaction().commit();
```

6. Persistence Context

- A cache (in-memory storage) where EntityManager keeps track of all entities.
- If we fetch an entity, it is stored here.
- It is like a temporary storage area between the application and the database.
- Whenever we make changes to an entity during a transaction, the persistence context keeps track of those changes. Once the transaction is done, it saves these changes to the database.

```
Student s1 = em.find(Student.class, 1L); // comes from DB
Student s2 = em.find(Student.class, 1L); // comes from cache (same object)
```

Here, no second DB query is executed for s2 because JPA fetches it from the persistence context.

7. EntityTransaction

- Every operation in JPA happens inside a transaction.
- begin() starts it, commit() saves changes, rollback() undoes them.

```
EntityTransaction tx = em.getTransaction();
tx.begin();

Student s = em.find(Student.class, 1L);
s.setEmail("newmail@test.com");

tx.commit(); // update query fires here
```

8. Query

- We use JPQL (Java Persistence Query Language) or native SQL through EntityManager to fetch data.

```
List<Student> students = em
.createQuery("SELECT s FROM Student s", Student.class)
.getResultList();
```

All the code together:

```
//1. Entity (mapped to a table)
@Entity
public class Student {
    @Id
    private Long id;
    private String name;
    private String email;
}

//2. Create EntityManagerFactory using Persistence
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("my-persistence-unit");

//3. Create EntityManager
EntityManager em = emf.createEntityManager();

//4. Transaction example
EntityTransaction tx = em.getTransaction();
tx.begin();

Student s = new Student();
s.setId(1L);
s.setName("Ramesh");
s.setEmail("ramesh@test.com");

em.persist(s); // INSERT query happens here
tx.commit();

//5. Persistence Context example
Student s1 = em.find(Student.class, 1L); // from DB
Student s2 = em.find(Student.class, 1L); // from cache (same object)

//6. Query example
List<Student> students = em
    .createQuery("SELECT s FROM Student s", Student.class)
    .getResultList();
```

Reference: [JPA Architecture](#)

Here is the flow Your Code (Spring Data JPA) -> EntityManager -> Hibernate (JPA implementation) -> JDBC Driver -> Database.

1. Our Code (Spring Data JPA)

- You define repositories by extending JpaRepository.
- Spring auto-implements CRUD methods like save, findById, delete.

2. EntityManager (JPA)

- Central JPA interface that manages entities and queries.
- Spring Data JPA delegates all operations to this under the hood.

3. Hibernate (JPA Implementation)

- Hibernate implements JPA and powers the EntityManager.
- It generates SQL, manages caching, lazy loading, and ORM mapping.

4. JDBC Driver

- Vendor library like mysql-connector-java or postgresql.
- Converts JDBC calls into the database-specific protocol.

5. Database

- Stores your actual data (tables, rows, indexes).
- Executes SQL and returns results back up the stack.

Spring 55: What will happen under the hood when you call .save method of Repository?

The following code is for save. UserService is calling save() of userRepository.

```
public interface UserRepository extends CrudRepository<User, Integer> {
    // no need to write save(), findById(), etc.
    // CrudRepository already provides them
}
```

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User saveUser(User user) {
        return userRepository.save(user);
    }
}
```

Step 1 - Call save()

When we write the line `userRepository.save(new User(1, "CodingLyf"));`

At this moment, the Java object is just a plain User object in memory (a POJO). It is not yet connected to the database.

Step 2 - Repository Layer (Spring Data JPA)

The UserRepository we wrote extends JpaRepository or CrudRepository. That means Spring has already created a **proxy class** for it at runtime.

So, when we call save(), we are not calling the actual code, we are calling a generated proxy that knows how to talk to JPA's EntityManager.

Step 3 - EntityManager Comes Into Play

The proxy internally delegates to JPA's EntityManager.

Think of EntityManager as the gatekeeper between the Java objects and the database.

When save() is called, EntityManager decides whether to:

- **Insert** (if the object is new, not in DB yet)
- **Update** (if the object already exists in DB with the same id)

```

629@  @Override
630  @Transactional
631  public <S extends T> S save(S entity) {
632
633      Assert.notNull(entity, ENTITY_MUST_NOT_BE_NULL);
634
635      if (entityInformation.isNew(entity)) {
636          entityManager.persist(entity);
637          return entity;
638      } else {
639          return entityManager.merge(entity);
640      }
641  }

```

Inside SimpleJpaRepository, the save() method checks whether the entity is new or already exists.

Step 4 - Check Entity State (Transient, Managed, Detached, Removed)

- At this moment, new User(1, "CodingLyf") is in the **Transient state**.
- That means it exists in memory but is not yet known to the persistence context.
- save() will call EntityManager.persist() if it is new, or merge() if it thinks it already exists.

Step 5 - Persistence Context (a.k.a First-Level Cache)

EntityManager maintains something called a persistence context. Think of it as a cache in memory that holds all entities being tracked in the current transaction.

When we save the User:

- It first gets added to the persistence context.
- Now JPA starts tracking it. If we change the object later, JPA knows what changed.

Step 6 - SQL Generation

The actual **INSERT** or **UPDATE** statement is executed only when:

- The transaction commits, or
- EntityManager.flush() is called.

At that moment, Hibernate translates the entity into SQL. For example:

For a new user > JPA generates an INSERT INTO user ... query.

For an existing user > JPA generates an UPDATE user ... query.

Step 7 – Transaction Commit

Spring Data JPA usually runs the save() inside a transaction.

When the transaction commits:

- JPA flushes all pending changes from persistence context to the database.
- The SQL is executed.
- The record is physically written in the DB table.

Step 8 – Return Value

Finally, save() returns the persisted entity.

Indirect Questions:

1. Difference between save() and saveAndFlush()

When we call save(), Spring Data JPA doesn't immediately fire the SQL query. Instead it makes entry in the persistent context. It saves in DB only when the transaction is committed or a flush operation occurs

saveAndFlush () performs the same operation as save() but immediately flushes the changes to the database. This ensures that the entity's state is synchronized with the database right away

2. Difference between persist() and merge()

persist() is used to take a brand-new object (in the *transient* state) and make it managed state, scheduling it for insertion into the database.

merge() is used to take an existing object that is not managed (in the *detached* state) and bring it back into the managed state, scheduling it for update in the database.

307 Spring 56: what is Persistence context in JPA?

Persistence context in JPA as a kind of first-level cache that sits between the application and the database.

- When we fetch or save an entity using EntityManager, JPA doesn't immediately go to the database every time. Instead, it keeps those entity objects inside the persistence context (in memory).
- If we ask for the same entity again, JPA just gives the cached object from the persistence context , no extra SQL.

- Any changes we make to that entity are tracked automatically. When we call `commit()` or `flush()`, JPA writes those changes back to the database.

```
EntityManager em = ...;

em.getTransaction().begin();

Employee e1 = em.find(Employee.class, 1); // SQL fired, entity loaded
Employee e2 = em.find(Employee.class, 1); // No SQL, comes from persistence context

e1.setName("John Updated"); // Change tracked in persistence context

em.getTransaction().commit(); // SQL UPDATE happens here
```

Reference: [Hibernate Persistence Context. In this article, we'll talk about... | by Emre DALCI | emlakjet | Medium](#)

[\[Spring Boot\] Persistence Context | by Hwangon Jang | Medium](#)

Additional Question:

If I have a student object managed by JPA and I change the student's name multiple times (say 100 times) before committing, will all 100 changes be saved to the database, or only the final change? How does the persistence context determine what to update.

```
@Entity
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    // Constructors
    public Student() {}

    public Student(String name) {
        this.name = name;
    }
    // Getters and Setters
}
```

- In JPA/Hibernate, the persistence context acts like a cache for entity objects. When we fetch or create an entity and it is managed by the persistence context
- Changes are not immediately written to the database. They are flushed either when we explicitly call `entityManager.flush()` or at the end of a transaction.
- Persistence context compares the current in-memory state of the entity with the snapshot taken when the entity was first loaded to decide what to update. This is called **dirty checking**.

- Even if we change the student's name 100 times within the same transaction, only the last change will be saved. The intermediate changes are not individually saved because the persistence context only cares about the difference between the original and the final state at the time of flush/commit.

308 Spring 57: what are different states on an Entity?

1. Transient (New)

An entity is transient when it is just created using new, not linked to the database or JPA. It is not tracked, not stored, and no SQL will be triggered. Unless we explicitly persist it, it is just a regular object living in memory with no database interaction.

2. Managed (Persistent)

Once we call persist() or fetch it using find(), the entity becomes managed. JPA starts tracking it. Any changes we make are auto-saved to the database on commit. This state ensures automatic synchronization between the Java object and the underlying database table, reducing manual SQL handling.

3. Removed

If a managed entity is passed to remove(), it is marked for deletion. It stays in memory but will be deleted from the database when the transaction commits. While in this state, it is still tracked by JPA, but all changes become irrelevant as the entity is scheduled for removal.

4. Detached

A detached entity was once managed but is no longer tracked; usually after commit or closing the EntityManager. Changes to it won't reflect in the database unless reattached using merge().

309 Spring 58: How Cache works in JPA?

When we work with JPA, caching plays a huge role in performance. Without cache, every time we try to fetch an entity, JPA would hit the database, which is expensive. Caching avoids unnecessary round trips to the database and makes the application faster.

Now let's understand step by step.

1. First-Level Cache (Persistence Context Cache)

Think of the first-level cache like a short-term memory. When we start a transaction, JPA creates a persistence context. Inside this persistence context, all the entities we load or save are stored temporarily in memory.

The important thing is:

- This cache is tied to the lifecycle of the EntityManager.
- Once we close the EntityManager, the cache is gone.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
```

```
Student s1 = em.find(Student.class, 1L); // First time : SQL query runs
Student s2 = em.find(Student.class, 1L); // Comes from cache, no SQL

em.getTransaction().commit();
em.close();
```

Here, the second time we ask for the student, JPA doesn't bother going to the database. It already knows the entity is inside the persistence context, so it just returns the cached object.

Key Point: we cannot turn off the first-level cache. It is always there.

2. Second-Level Cache (Shared Cache)

Now imagine your app has multiple sessions and same entity is needed by multiple sessions? Without a shared cache, each session would query the database again.

That is where the **second-level cache** comes in.

- It sits at the EntityManagerFactory level.
- It can be shared across multiple sessions.
- It is not enabled by default; we have to configure it.
- Common providers are **EhCache, Infinispan, Hazelcast**.

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(
    usage = CacheConcurrencyStrategy.READ_WRITE
)
public class Student {
    @Id
    private Long id;
    private String name;
}
```

When this is enabled, if one session loads a Student from DB, the next session can pick it up directly from the cache instead of querying the database again. This is where the real performance boost happens in large applications.

Real time Example:

Without Second-Level Cache

- **User A** logs in > System opens a session
- They view **Product with ID=101** > DB is hit, product details are fetched.
- Later, **User B** logs in > New session (
- They also view **Product with ID=101** > Again DB is hit because each session has its own first-level cache only.

So, the same product info is fetched twice from DB, even though nothing changed.

With Second-Level Cache

- **User A** logs in > System fetches **Product 101** from DB and puts it into second-level cache.
- **User B** logs in > When they view **Product 101**, Hibernate/JPA picks it from the cache instead of hitting DB again.

That is why in real-world apps with many users requesting the same data (products, courses, profiles), second-level cache saves number of DB hits.

Point to remember: First level cache is session-scoped, whereas second level cache is application scope

Indirect Questions:

1. When will cache be updated?

The second-level cache is updated automatically whenever Hibernate changes the entity (insert, update, delete). If changes happen outside Hibernate means if we update the DB directly, we need refresh or eviction to sync it.

2. Can caching cause stale data issues?

Yes. If another transaction updates the database, the cache might still hold the old value. To avoid this, second-level caches use cache invalidation or concurrency strategies like `READ_ONLY`, `NONSTRICT_READ_WRITE`, etc.

- `READ_ONLY` > cache never updated (safe for reference data like countries, constants).
- `READ_WRITE` > updates both DB and cache when entity changes.
- `NONSTRICT_READ_WRITE` > DB updated immediately, cache updated lazily (may serve slightly stale data).

We must have proper configuration for cache, like TTL Time-to-Live

If the application is clustered, we must use a distributed L2 cache like Hazelcast

310 Spring 59: How to connect to multiple databases?

I would recommend to watch this video first, then go through the theory below.

Reference video: [Spring Boot : How to connect with multiple databases using Spring Data JPA](#)

When we work with multiple databases, we need three main things for *each database*:

1. **DataSource**: the connection details (URL, username, password). We need to define multiple datasources
2. **EntityManagerFactory**: responsible for creating and managing JPA's entity managers (basically the bridge between the entities and the DB).
3. **TransactionManager**: tells Spring how to handle transactions (commit/rollback).

And finally, we map specific **entities** and **repositories** to the right database.

Let's say we have two databases:

- Primary DB: users (userdb)
- Secondary DB: orders (orderdb)

Step 1: First, we need to configure the data sources in properties file.

```
# Primary DB (users)
spring.datasource.primary.url=jdbc:mysql://localhost:3306/usersdb
spring.datasource.primary.username=root
spring.datasource.primary.password=secret
spring.datasource.primary.driver-class-name=com.mysql.cj.jdbc.Driver

# Secondary DB (orders)
spring.datasource.secondary.url=jdbc:mysql://localhost:3306/ordersdb
spring.datasource.secondary.username=root
spring.datasource.secondary.password=secret
spring.datasource.secondary.driver-class-name=com.mysql.cj.jdbc.Driver
```

Step 2: Package structure

A neat way is to group everything by database:

```
com.example.app
  └── userdb
      ├── entity
      │   └── User.java
      ├── repository
      │   └── UserRepository.java
      └── config
          └── UserDbConfig.java
  └── orderdb
      ├── entity
      │   └── Order.java
      ├── repository
      │   └── OrderRepository.java
      └── config
          └── OrderDbConfig.java
  └── service
      └── DemoService.java
```

- Each DB has its **own package**.
- Inside it, we separate entity, repository, and config.
- Shared business logic (services, controllers) sits outside, so they can use repos from both DBs if needed.

Step 3: Define @EnableJpaRepositories

1. **@EnableJpaRepositories** is a Spring Data JPA annotation used to enable the detection and creation of Spring Data JPA repositories within a Spring application.

Purpose:

- It instructs Spring to scan specified packages for interfaces extending Spring Data JPA's JpaRepository (or related interfaces like CrudRepository, PagingAndSortingRepository).
- For each such interface, Spring automatically generates a concrete implementation (proxy) at runtime, allowing to inject and use these repository interfaces directly in the application.

```
@EnableJpaRepositories(
    basePackages = "com.example.users",
    entityManagerFactoryRef = "userEntityManagerFactory",
    transactionManagerRef = "userTransactionManager"
)
```

2. **basePackages = "com.example.users"**

This is the folder (package) where our repository interfaces exist (Check above package info).

- Example: inside com.example.users.repository, we have UserRepository, UserProfileRepository.
- Spring will scan this package, find those interfaces, and auto-generate classes for them.

Without this, Spring won't know where the repositories are hiding.

3. **entityManagerFactoryRef = "userEntityManagerFactory"**

Now, repositories need to know *which database connection + mapping rules* they belong to.

That is handled by the **EntityManagerFactory**.

"Repositories in com.example.users should use the userEntityManagerFactory."

Why is this important?

- In single-DB apps, there's only one entity manager, so we don't need this.
- In multi-DB apps, each DB has its own entity manager, so we must explicitly set it.

4. **transactionManagerRef = "userTransactionManager"**

Databases don't just read and write; they also commit or roll back transactions.

It says "When these repositories do database work, handle their transactions using userTransactionManager."

So:

- If something fails in a UserRepository call, this manager decides whether to rollback or commit.
- Again, in multi-DB setups, each DB has its own transaction manager.

Code link: [ashokitschool/springboot_multi_db_config: springboot_multi_db_config](#)

Indirect Questions:

1. What happens if you don't specify basePackages in @EnableJpaRepositories?

- By default, it scans the same package where our configuration class is placed (and sub-packages).
- If our repos are elsewhere, they won't be found and we will get No qualifying bean errors.

2. Why do we need entityManagerFactoryRef in a multi-database setup?

- Because each DB has its own entity manager.
- Without this reference, Spring wouldn't know which database connection to use for those repositories.
- It is used to mix up entities of DB1 with DB2.

3. Can we skip transactionManagerRef? What happens?

- If we have only one database, Spring will auto-wire the default transaction manager.
- In multi-DB apps, we must specify it, otherwise the wrong transaction manager might get applied, leading to runtime errors or commits going to the wrong DB.

4. If I put all my repositories (User and Order) in the same package, will @EnableJpaRepositories still work?

- Yes, @EnableJpaRepositories will still work even if we put all the repositories (User, Order, etc.) in the same package.
- But it is good to put repositories according to the DB for better maintainability:

5. What's the difference between @EnableJpaRepositories and @Repository on an interface?

- @Repository marks an interface or class as a Spring bean.
- @EnableJpaRepositories is what actually scans for those interfaces and generates implementations.
- Without @EnableJpaRepositories, our @Repository interfaces won't be picked up automatically.

6. How does Spring Boot behave if you don't even use @EnableJpaRepositories anywhere?

- For a single database setup, Spring Boot auto-configures it.
- That is why in single DB projects, we never see @EnableJpaRepositories
- For multiple DBs, we must configure it explicitly, otherwise Spring won't know how to separate repositories by database.

7. Can a repository use multiple EntityManagerFactories at once?

- No. A repository is tied to exactly one entity manager (hence one DB).
- If we need data from multiple DBs, we
- fetch them via different repositories in the service layer.

311 Spring 60: How do you create Custom repository in Spring JPA?

Spring Data JPA gives us a lot out of the box with JpaRepository, like findAll(), save(), delete(), etc. But sometimes, we need custom queries or complex logic when we can't achieve with derived query methods. That is where a custom repository comes in.

Reference: [Spring data JPA custom repository](#). Watch this video before reading the theory.

Steps to create a custom repository

1. Create a custom interface, this is where we declare methods that aren't in JpaRepository.

```
public interface EmployeeRepositoryCustom {
    List<Employee> findEmployeesWithHighSalary(double minSalary);
}
```

2. Create an implementation class: By convention, name it <RepositoryName>Impl. Here we directly use EntityManager.

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import java.util.List;

public class EmployeeRepositoryImpl implements EmployeeRepositoryCustom {

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public List<Employee> findEmployeesWithHighSalary(double minSalary) {
        return entityManager.createQuery(
            "SELECT e FROM Employee e WHERE e.salary > :salary", Employee.class)
            .setParameter("salary", minSalary)
            .getResultList();
    }
}
```

3. Extend the main repository with the custom one. Now EmployeeRepository has both the standard JPA methods and the custom one.

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository
    extends JpaRepository<Employee, Long>, EmployeeRepositoryCustom {
}
```

Code: [shameed1910/spring-jpa-custom-repo](#)

312 Spring 61: What is connection pooling, how it helps in Spring?

Connection pooling is a technique used to manage and optimize database connections in applications. Instead of opening a new database connection for each request and then closing it after

use, connection pooling creates a pool of pre-defined and reusable connections. When an application needs to interact with the database, it borrows a connection from this pool, uses it, and then returns it to the pool for later reuse.

Since Spring Boot 2.x, **HikariCP** has been the default connection pool due to its speed, simplicity, and reliability. HikariCP is designed to be a lightweight and high-performance connection pooling library that performs well under various loads.

Setting up HikariCP in Spring Boot is straightforward since Spring Boot automatically configures HikariCP as the default connection pool if it is available in the classpath. HikariCP settings can be customized in the application.properties or application.yml file.

```
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.connection-timeout=30000
spring.datasource.hikari.idle-timeout=600000
spring.datasource.hikari.max-lifetime=1800000
```

Reference: [How to Use Connection Pooling for Faster Database Access in Spring Boot | by Alexander Obregon | Medium](#)

313 Spring 62: What do you know about @NoRepositoryBean?

```
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> { }
```

Spring Data JPA is a powerful tool for simplifying database access in Spring Boot applications, providing repository interfaces that reduce the boilerplate code for CRUD operations.

Sometimes we want to define a **base repository** with shared methods for multiple repositories, but we don't want Spring to create a bean for that base interface itself. That is where **@NoRepositoryBean** comes in.

The primary function of **@NoRepositoryBean** is to prevent Spring Data JPA from creating a repository proxy for the annotated interface or class. It is typically used on base repository interfaces or classes that define common methods for a set of repositories.

Example: Library System

Let's say we are building a library system. We want a base repository with common methods for all items in the library (books, magazines, etc.).

Entities:

```
@Entity
public class Book {
    @Id
    private Long id;
    private String title;
    private String author;
    // getters/setters
```

```

}

@Entity
public class Magazine {
    @Id
    private Long id;
    private String title;
    private int issueNumber;
    // getters/setters
}

```

Now create a BaseRepository called `LibraryItemRepository`, which contains common method to both Book and Magazine.

```

@NoArgsConstructor
public interface LibraryItemRepository<T, ID> extends JpaRepository<T, ID> {

    // A custom method shared by all library items
    List<T> findByTitleContaining(String keyword);
}

```

Now Create Separate Respositories for Book and Magazine.

```

public interface BookRepository extends LibraryItemRepository<Book, Long> {
    List<Book> findByAuthor(String author);
}

public interface MagazineRepository extends LibraryItemRepository<Magazine, Long> {
    List<Magazine> findByIssueNumber(int issueNumber);
}

```

`LibraryItemRepository` is marked with `@NoRepositoryBean`, so Spring **does not create a bean** for it.

But Spring will create beans for `BookRepository` and `MagazineRepository`, and they both inherit the shared `findByTitleContaining()` method.

It is useful to avoid duplicate code in multiple repositories.

Reference: [Simplifying Database Access with `@NoRepositoryBean` | by Arsen Sargsyan | Octa Labs Insights](#)

314 Spring 63: What is Spring Security?

For Spring security along with JWT check this video: [Spring Security 6 with Spring Boot 3 and JWT Tutorial](#)

Spring Security is a framework that helps us to protect the Java applications. Think of Spring Security as a bodyguard for the application. Whenever a request comes into the system, it doesn't let anyone walk in freely.

It first checks who the user is (authentication) and then decides what the user is allowed to do (authorization).

So, in short:

- Authentication: Verifying identity (like logging in with username/password).
- Authorization: Checking permissions (like only admins can delete users).

Without Spring Security, we would have to write a lot of this logic by hand; validating users, handling login failures. Spring Security handles all of this in a structured way.

Imagine we are building an online banking app. we don't want just anyone to:

- Log in with any random credentials
- View someone else's account details
- Make unauthorized transactions

Spring Security makes sure that doesn't happen. It places a **security filter chain** in front of the application and intercepts requests before they reach the business logic.

Here is the most basic Spring Security setup in Spring Boot (using the new way with SecurityFilterChain). We will see every option in the upcoming questions.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/user/**").hasRole("USER")
                .anyRequest().authenticated()
            )
            .formLogin(Customizer.withDefaults()); // enables default login form
        return http.build();
    }
}
```

315 Spring 64: How do you enable Security in Spring applications?

Just add the dependency spring-boot-starter-security.

Once this is on the classpath, Spring Boot automatically configures a basic security setup.

By default, it secures all endpoints with HTTP Basic authentication and generates a default password (we can see it in logs).

Spring boot does default configuration when we added starter-security to class path.

Spring Boot Defaults (with spring-boot-starter-security)

1. All endpoints are secured

By default, every HTTP endpoint requires authentication.

Even / or /home is protected unless explicitly allowed.

2. HTTP Basic authentication enabled

Spring Boot automatically provides a login mechanism using HTTP Basic.

Spring will give default login page.

3. Auto-generated user

Spring Boot creates a default user with username user.

Password is auto-generated at runtime and printed in logs.

This password changes every time the app restarts.

4. Password is encoded

The default password uses BCryptPasswordEncoder internally.

5. CSRF protection enabled

For web applications, POST, PUT, DELETE, PATCH requests are protected with CSRF by default.

But if we want override the default we need to a security configuration file with **@EnableWebSecurity**.

316 Spring 65: What is @EnableWebSecurity?

The **@EnableWebSecurity** annotation is used to enable the web security of an application. When this annotation is added to a configuration class, it indicates that the class will provide the necessary configurations and settings for securing the web application.

In Spring boot application, using the **@EnableWebSecurity** annotation is optional. If we don't write one, Spring uses the default one.

When we want to override the default configuration we will use **EnableWebSecurity**.

Ex: when we want to make some pages public or when we want to provide custom filters

```
@Configuration
@EnableWebSecurity // This is still used to mark this as the security config class
public class SecurityConfig {

    // Bean to define security rules (replaces configure(HttpSecurity http))
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public/**").permitAll()
                .anyRequest().authenticated())
    }
}
```

```
        )
        .formLogin(withDefaults())
        .httpBasic(withDefaults())
        .build();
    }

    // Bean to ignore certain paths (e.g., for static resources)
    @Bean
    public WebSecurityCustomizer webSecurityCustomizer() {
        return (web) -> web.ignoring().requestMatchers("/css/**", "/js/**");
    }
}
```

317 Spring 66: What are the key features of Spring Security?

1. Authentication: Authentication is the process of validating the identity of a user who is attempting to access the application. Spring Security provides multiple authentication methodologies (Username/Password Authentication, JWT, OAUTH2, SSO).

2. Authorization (Access Control): Authorization is the process of granting permissions or rights to authenticated users. Once a user is successfully authenticated, authorization determines what actions or resources they are allowed to access within an application.

Example: Only admins can delete users.

3. Protection against Common Attacks

Spring Security automatically protects against many common vulnerabilities:

- CSRF (Cross-Site Request Forgery) > prevents fake form submissions.
- XSS (Cross-Site Scripting) > Escapes dangerous inputs.

4. Password Security

Passwords are never stored in plain text. Spring Security provides hashing algorithms like BCrypt, so even if the database is stolen, raw passwords aren't exposed.

Indirect Questions:

1. Difference between Authentication and Authorization

Authentication is checking who the user is. Example: logging in with the email and password.

Authorization is checking what the users are allowed to do. Example: an admin can delete users, but a normal user cannot.

318 Spring 67: What is authentication and different types of it?

Authentication is the process of verifying the identity of a user or system before granting access to resources.

Types of Authentications:

1. Username and Password Authentication: User provides a username and password and system verifies against the DB.

2. Token-Based Authentication (JWT): User logs in once and receives a token (like JWT). Every subsequent request includes the token (usually in Authorization header).

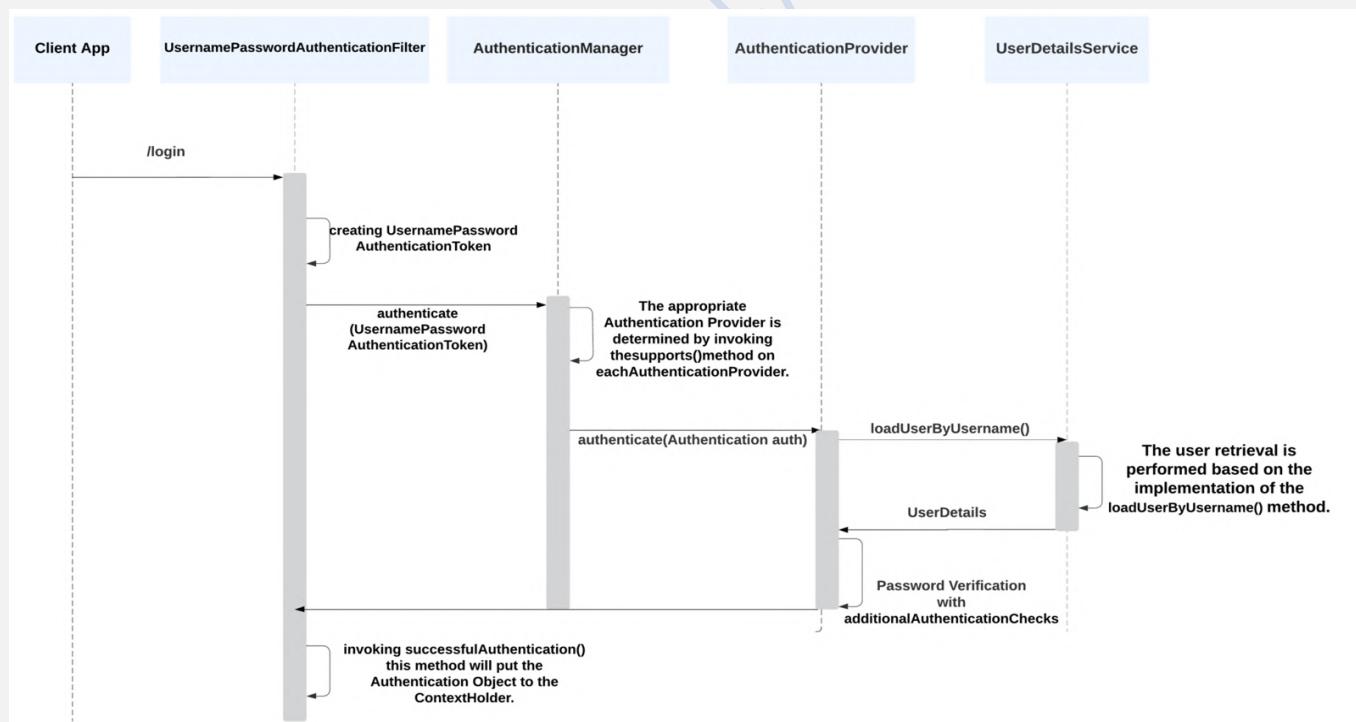
3. HTTP Basic Authentication: Credentials are sent in base64-encoded format in the request header. Simple, but not recommended for production unless combined with HTTPS.

4. OAuth2 / OpenID Connect: Delegated authentication via third-party providers like Google, Facebook, or Okta. The system relies on the external provider to authenticate the user and provide an access token. Useful for SSO (Single Sign-On).

5. Multi-Factor Authentication (MFA): Combines two or more authentication factors:

1. Something we know (password)
2. Something we have (OTP, mobile app)
3. Something we are (fingerprint, face)

319 Spring 68: What is the authentication flow in Spring security?



Authentication Flow in Spring Security

1. Request Enters the System

Every request first passes through the **SecurityFilterChain**. Here, Spring Security decides which filter should handle it.

- If the request is a POST with username and password, the **UsernamePasswordAuthenticationFilter** will be invoked.
- If the request has an Authorization: Basic base64(username:password) header, then the **BasicAuthenticationFilter** will be called.
- And so on, depending on the type of authentication method.

2. Filter Extracts Credentials

The chosen filter pulls authentication details (like username/password or token) from the request and builds an **Authentication object**.

3. AuthenticationManager Delegates

The filter then hands this Authentication object to the **AuthenticationManager**. This manager doesn't validate directly but it delegates to a chain of **AuthenticationProviders**.

4. AuthenticationProviders Validate

Each AuthenticationProvider tries to handle the request:

- If one recognizes the type of authentication and validates it successfully (say, by comparing username and password against the database), the process moves forward.
- If not, the next provider in the list gets a chance.
- AuthenticationProvider Uses **UserDetailsService** (Refer Question No. 327. Spring 75)
- If the provider is something like **DaoAuthenticationProvider**, it calls the UserDetailsService.
- UserDetailsService loads the user's details from the system (e.g., a database row with username, password, and roles).
- The provider then compares the supplied credentials (from the request) with the stored credentials (from UserDetailsService).

5. Success : Authentication Stored

Once a provider authenticates successfully, a fully populated **Authentication object** is created. This object contains the user's details and their authorities (roles/permissions).

6. SecurityContext Updated

Finally, this Authentication object is placed into the **SecurityContext**, which is like a session-level memory of "who is currently logged in." From this point on, Spring Security uses this context to decide whether the user has access to specific resources.

7. Failed AuthenticationException will be thrown

Additional Question:

1. What is SecurityContext and SecurityContextHolder

When a user logs into a Spring Security application, the system needs a place to store information about who the user is and what he can do. That is where the `SecurityContext` comes in.

Think of the `SecurityContext` as a little box that holds the authentication details:

- Who the user is (username, roles, authorities).
- Whether they are authenticated or anonymous.

Every request has its own `SecurityContext`, so Spring Security knows *which user* is making the request and what they're allowed to access.

SecurityContextHolder

Now, how do we access the `SecurityContext`? That is the job of the `SecurityContextHolder`. It is like the “manager” that gives the current `SecurityContext` for the running thread.

```
Authentication auth = SecurityContextHolder  
.getContext()  
.getAuthentication();  
  
String username = auth.getName();  
Collection<?> roles = auth.getAuthorities();
```

`getContext()` gives the `SecurityContext`.

`getAuthentication()` gives the actual login details.

320 Spring 69: What is JWT and how does it differ from Basic auth?

Both JWT (JSON Web Token) and Basic Authentication are ways to check a user's identity.

Basic Authentication

With Basic Auth, the client (like a browser or mobile app) sends the username and password with every request. These credentials are usually sent in the HTTP header, encoded in Base64 (which is not encryption, just a form of encoding).

- **Pros:** Simple to implement, no extra setup needed.
- **Cons:** Insecure if not used over HTTPS (credentials can be stolen), and credentials are repeatedly exposed on every request.

JWT takes a different approach. Instead of sending the username and password every time, the server gives the client a **token** after a successful login. This token is a signed string containing encoded information about the user (like their username and roles).

From that point on, the client only needs to send the token with each request, no passwords involved. The server checks if the token is valid and not expired, then processes the request.

Good to know

Before JWTs became popular, web apps commonly used session IDs. Here is how that worked:

1. When user logs in, server creates a session and stores user info on the server.
2. Server gives the client a session ID, usually stored in a cookie.
3. For each request, the client sends the session ID, and the server looks up the session data.

This works fine for small apps, but it doesn't scale well:

- The server must keep session data in memory or a database. More users mean server requires more memory.
- In a load-balanced system with multiple servers, we need session to store in shared cache (Redis, Memcached). That adds complexity.
- REST APIs are supposed to be stateless, but sessions make the server stateful.

JWT solves these problems by keeping all the necessary user information inside the token itself, so the server doesn't need to store anything about the user between requests.

Reference: [What is JWT? JSON Web Tokens Explained \(Java Brains\)](#)

Additional Question:

1. Should you use JWT or Session-based authentication in the microservices environment?

In a microservices environment, JSON Web Tokens (JWTs) are generally preferred over session-based authentication.

In session-based authentication, the server has to store session data. Works fine for small apps, but in microservices it is hard because every service would need to share session data.

With JWT after login, we get a token. We send that token every time. Each service can check it without talking to others. Much easier for microservices.

2. How does Basic Authentication work in Rest API?

Basic Authentication in REST API works like this:

1. When a client (say, a browser or Postman) makes a request, it attaches an HTTP header:
Authorization: Basic <Base64 encoded username:password>
2. The server receives it, decodes the Base64 string back to username:password, and checks against its user store (like DB or in-memory).
3. If it matches, the request is allowed. If not, it is rejected with 401 Unauthorized.

Problem: credentials are sent on every request, so it must be used over HTTPS to avoid the exposure of passwords

3. In Spring boot, how is session managed?

When a user logs in, the server needs to remember the user between requests. HTTP itself is stateless, meaning it forgets the user every time. To fix that, Spring Boot uses session management.

1. Session Creation

- After a successful login (say with form login or basic authentication), the server creates a session object (HttpSession).
- This session is just a unique ID (called JSESSIONID) stored on the server.

2. Session ID Sharing

- The server sends this ID back to the browser as a cookie.
- On every new request, the browser automatically sends this cookie back.
- Spring uses it to look up the stored session and retrieve the user's authentication details.

3. Where the Session Data Lives

By default, Spring Boot keeps session data in memory (in the server's JVM).

But we can change this to:

- Database
- Redis (common for microservices and scaling)
- Custom storage

321 Spring 70: What is structure of the JWT?

A JWT (JSON Web Token) is just a long string, but it is actually made of three distinct parts, separated by dots (.):

xxxxx.yyyyy.zzzzz

1. Header

A small JSON object that describes the token.

It usually contains:

- The type of token (typ) , always "JWT".
- The signing algorithm used (alg) like "HS256" or "RS256".

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

2. Payload

This is the main content of the token.

It holds the information about the user and some metadata.

Claims are key-value pairs, for example:

sub: Subject (the user ID)

name: Username
 roles: User's roles or authorities
 exp: Expiration time of the token

```
{
  "sub": "12345",
  "name": "CodingLyf",
  "roles": ["ADMIN", "USER"],
  "exp": 1699999999
}
```

3. Signature

This is the security part of the token

It is created by `Base64UrlEncode(header) + "." + Base64UrlEncode(payload)` and then signing it with a secret key or private key using the algorithm from the header.

Reference: [What is the structure of a JWT - Java Brains](#)

322 Spring 71: How server validates the JWT token?

When a user logs in successfully, the server creates a JWT token and sends it back to the client (a browser or an API client). From then on, the client must store this token (usually in local storage or cookies) and attach it to every request it makes. This way, the server can check if the request comes from an authenticated user.

But here is the real question is, how does the server know the JWT is valid?

1. Browsers send Token in Authorization header. Authorization: Bearer <jwt_token>
2. Server will read this header and extracts the token.
3. From Token, we need to extract the username and now verify this username with the usernames stored in DB. If it matches token is valid , if it is invalid token and returns 401 error.

Point to remember: All these steps will be done in OncePerRequestFilter.

OncePerRequestFilter ensures the JWT validation logic runs exactly once for each request, extracts the username from the token, validates it, and sets authentication in the security context.

Reference: [#38 Spring Security | Validating JWT Token](#)

Additional Question:

If JWT Token is expired midway of the request, how would you handle this?

Handling a JWT token that expires midway through a request typically involves a combination of client-side and server-side mechanisms, often leveraging refresh tokens.

1. Server-Side Handling:

Token Validation:

The server-side API endpoint receiving the request must validate the JWT token. If the token is found to be expired during this validation, the server should respond with a 401 Unauthorized status code.

Refresh Token Endpoint:

A dedicated endpoint for refreshing access tokens is crucial. This endpoint accepts a valid refresh token and, upon successful verification, issues a new, unexpired access token.

2. Client-Side Handling:

Intercepting Responses:

The client-side application (e.g., a web or mobile app) should have a mechanism to intercept API responses, specifically looking for 401 Unauthorized errors.

Refresh Token Flow:

When a 401 Unauthorized response is received due to an expired access token, the client should:

- Attempt to use the stored refresh token to request a new access token from the refresh token endpoint.
- If the refresh token is valid and a new access token is successfully obtained, the client should update its stored access token and then retry the original failed request with the new access token.

323 Spring 72: How to implement JWT authentication in Spring boot?

I recommend to watch this video: [Easy JWT Authentication & Authorization with Spring Security | Step-by-Step Guide](#)

Code Link: [learnwithiftekhar/Spring-Security-JWT](#)

JWT (JSON Web Token) authentication in Spring Boot is basically about replacing the default session-based authentication with stateless token-based authentication.

1. User Login Request

- The client (browser or mobile app) sends a login request with username and password.
- This hits the AuthenticationController.

2. Authentication Manager + UserDetailsService

- Spring Security passes the credentials to the AuthenticationManager.
- That, in turn, uses the custom UserDetailsService to fetch the user and check the password with a PasswordEncoder.

If it is valid > user is authenticated.

If not > reject immediately.

3. JWT Token Creation

- Instead of creating a session, we now generate a JWT.

- JWT contains claims like: username, roles, issued time, expiration.
- It is signed with the secret key (so no one can tamper with it).
- In Spring we use JWT Library (commonly used is jjwt)

This token is then sent back to the client in the response.

4. Client Stores the JWT

- The client usually stores this JWT in localStorage or sessionStorage or cookies (in browsers), or secure storage (in mobile apps).
- From now on, every request from client : server includes this token in the Authorization header: `Authorization: Bearer <jwt_token>`

5. JWT Authentication Filter

- We add a custom filter by extending OncePerRequestFilter in the Spring Security filter chain (before UsernamePasswordAuthenticationFilter).
- This filter:
 1. Reads the Authorization header.
 2. Extracts the JWT token.
 3. Validates it (check signature + expiration).
 4. If valid : create an Authentication object and put it into the SecurityContext.
 5. If invalid : reject request with 401.

324 Spring 73: How does SecurityFilterChain works and list down the filters?

In Question No 320, Spring 68 we have seen Architecture of the Spring security. From here we are going depth into every component of the Spring security.

In Spring Security, everything that protects the application runs through a chain of filters. Think of it like airport security , we don't just pass through one gate, we pass through a series of checks: baggage scan, passport check, body scan, etc. Each check is a filter with a specific job.

The SecurityFilterChain is literally that chain of filters applied to incoming requests.

- Every HTTP request enters the filter chain.
- Each filter either does some security work (like authentication, authorization, CSRF check, session check) or passes control to the next filter.
- If a request fails a check (say invalid credentials), the chain is broken, and the request is rejected immediately.

Without this chain, Spring Security can't intercept and secure requests.

Types of filters:

- **Authentication Filter:** Responsible for handling user authentication.
- **Authorization Filter:** Enforces access control policies and checks whether a user is authorized to access a resource.
- **CSRF (Cross-Site Request Forgery) Filter:** Protects against CSRF attacks. (Refer Question No)
- **UsernamePasswordAuthenticationFilter:** Validates the username and password for the URL (/login) with the default credentials provided at startup.
- **BasicAuthenticationFilter:** This filter is responsible for processing any request that has an HTTP request header of Authorization, Basic Authentication scheme, Base64 encoded username-password.
- **BearerTokenAuthenticationFilter:** Extracts JWT tokens for authentication.
- **FilterSecurityInterceptor :** This filter is responsible for authorizing every request that passes through the filter chain before the request hits the controller.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws
Exception {
    httpSecurity
        .csrf(csrf -> csrf.disable())
        .authorizeHttpRequests(
            request -> request
                .requestMatchers("register", "login").permitAll()
                .anyRequest().authenticated()
        )
        .httpBasic(Customizer.withDefaults())
        .addFilterBefore(jwtAuthenticationFilter,
            UsernamePasswordAuthenticationFilter.class);
    return httpSecurity.build();
}
```

Explanation for above code Snippet:

authorizeHttpRequests: It sets authorization rules.

requestMatchers is used to define which request URLs or HTTP methods a rule should apply to. Basically, it tells Spring Security, “for these specific paths, apply the following access rules.”

From the above code, Requests to /register and /login are public means no authentication needed. Other than these APIs, remaining all should go through authentication process.

Behind the scenes, this adds a FilterSecurityInterceptor into the chain, which checks permissions before letting the request proceed.

httpBasic: This enables HTTP Basic Authentication. Spring Security will insert a BasicAuthenticationFilter into the filter chain. This filter checks the Authorization: Basic ... header, extracts username/password, and tries to authenticate the user.

addFilterBefore allows us to insert the custom filter into the SecurityFilterChain before another specific filter.

addFilterBefore(myFilter, UsernamePasswordAuthenticationFilter.class)

Here myFilter runs *before login/authentication* happens. Commonly used for JWT validation so that requests are authenticated without hitting the username-password login flow.

Indirect Questions:

[1. How you make certain APIs skip authentication?](#)

Using requestMatchers

[2. How can you disable the CSRF](#)

Using CSRF filter and csrf.disable() method

325 Spring 74: Internals on SecurityFilterChain?

1. DelegatingFilterProxy

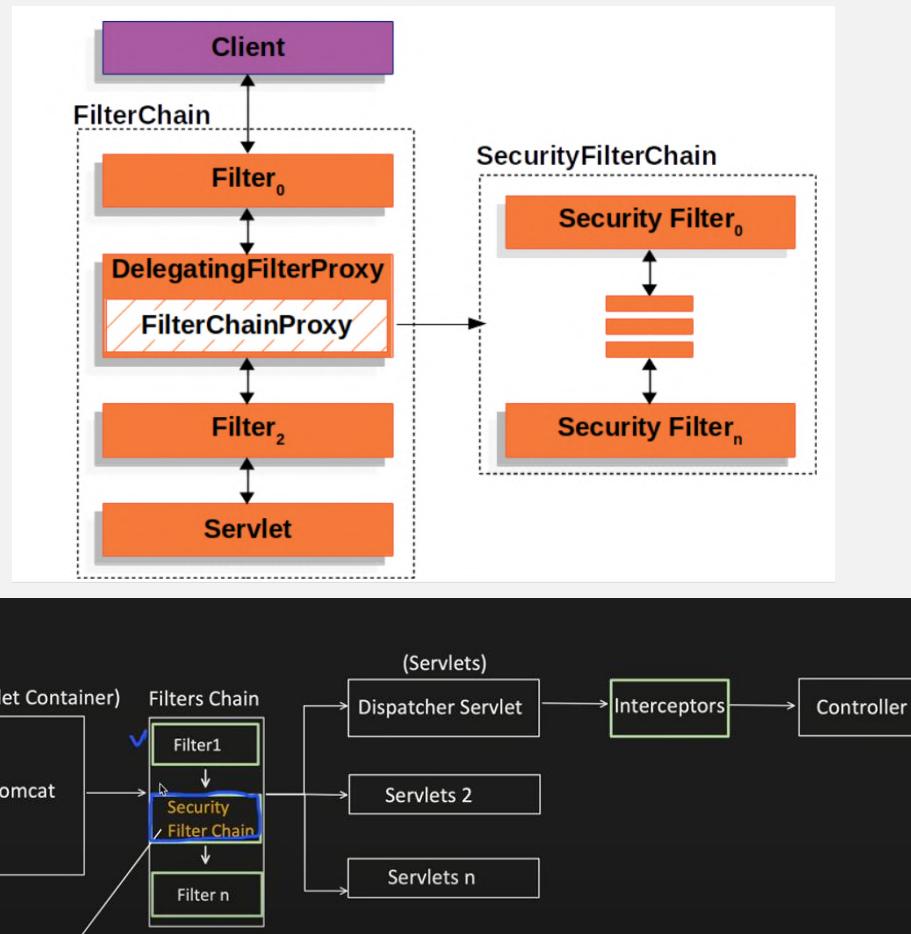
Spring's security filters live inside the Spring ApplicationContext (managed beans). But the servlet container (like Tomcat, Jetty) doesn't know anything about that

So DelegatingFilterProxy acts as a bridge. This is a Servlet Filter that comes from the Spring Framework, not just Spring Security. It acts as a bridge between the Servlet container (like Tomcat, Jetty) and the Spring-managed beans

Servlet container calls > DelegatingFilterProxy > delegates to Spring bean (FilterChainProxy).

2. FilterChainProxy

- This is a Spring Security-specific filter.
- Inside Spring, all the security rules (authentication, authorization, CSRF, JWT filters, etc.) are arranged as a list of security filter chains.
- FilterChainProxy is responsible for picking the right chain based on the request (URL matching) and executing it.



Flow:

Request > DelegatingFilterProxy > FilterChainProxy > Security Filters > DispatcherServlet > Controller

326 Spring 75: What is UserDetailsService and how it works?

It is just an interface provided by Spring Security.

Its main job is to fetch user information (like username, password, roles) from the data source (database, LDAP, in-memory, etc.) so that Spring Security can use it during authentication.

So, if Spring Security wants to log someone in, it calls loadUserByUsername().

We return a UserDetails object that represents that user.

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
}
```

How does it work in the authentication process?

1. User tries to log in (for example, submitting a username/password).

2. AuthenticationManager receives the credentials.
3. AuthenticationManager asks a UserDetailsService to fetch the user by username.
4. The implementation of UserDetailsService loads the user details from database (or wherever).
5. Spring Security wraps it in a UserDetails object.
6. The AuthenticationProvider checks if the password matches and assigns roles/authorities.
7. If all checks out, authentication succeeds and a SecurityContext is created for that user.

```
@Service
public class MyUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository; // JPA repo

    public MyUserDetailsService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        UserEntity user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not found: " +
username));

        return org.springframework.security.core.userdetails.User
            .withUsername(user.getUsername())
            .password(user.getPassword()) // already encrypted
            .authorities("ROLE_USER")
            .build();
    }
}
```

Key points

- UserDetailsService does not authenticate. It only fetches user data.
- Password check is done by an AuthenticationProvider (like DaoAuthenticationProvider).
- We can have multiple UserDetailsService implementations if needed.
- If we don't configure one, Spring can use an in-memory we by default.

327 Spring 76: Explain about Password Encryption and why it is important?

When a user registers or logs in, their password is the most sensitive piece of data. If we store or transmit it as plain text, anyone with database access (admins, hackers, even accidental log dumps) can see it. That is a massive security risk.

Instead, Spring Security (and security best practices in general) never stores plain passwords. Instead, it stores a hashed and salted representation of the password.

- Hashing means converting the password into a fixed-length string using an algorithm (like BCrypt).
- Salting means adding a random string before hashing, so that even if two users have the same password, their stored hashes are different.

How Password Encryption Works in Spring Security

PasswordEncoder Interface

- Spring Security provides PasswordEncoder, an interface for encoding and matching passwords.
- The most commonly used implementation is BCryptPasswordEncoder.

```
PasswordEncoder encoder = new BCryptPasswordEncoder();
String encodedPassword = encoder.encode("mySecretPassword");
```

328 Spring 77: What is CSRF and how it works in Spring?

Cross-Site Request Forgery (CSRF) is a web security vulnerability that tricks a user's browser into performing unwanted actions on a website where the user is authenticated.

For example:

- A user is logged into his bank site in one tab.
- In another tab, the user visits an attacker's site (Thinking that it is normal website).
- That site secretly submits a money transfer request to the bank using user session cookies.
- Since the browser automatically sends cookies with every request, the bank thinks the user triggered the request.

Spring Security protects against CSRF by using a synchronizer token pattern:

How it works in Spring Security:

- When the server renders a form or page, it generates a unique CSRF token and stores it in the user's session.
- This token is also embedded into the HTML (for example, as a hidden input field or in a meta tag for AJAX).
- When the user submits a form or makes a state-changing request (POST, PUT, DELETE), the token must be included in the request.
- Spring Security checks if the token in the request matches the token stored in the session.

If the tokens don't match > request is rejected.

If they match > request is valid and processed.

329 Spring 78: What is CORS, how we need to resolve it?

CORS stands for **Cross-Origin Resource Sharing**.

- Browsers enforce a security policy called the **Same-Origin Policy**, which prevents a webpage's scripts from contacting or loading data from other websites.
- For example: If frontend is at `http://localhost:3000` and backend API is at `http://localhost:8080`. A fetch request from frontend to backend is **cross-origin**.
- By default, the browser blocks such requests unless the server explicitly allows them. That is where CORS comes in.

Using NGINX to handle CORS is common, especially when we want to manage it outside the application.

NGINX acts as a reverse proxy in front of the Spring Boot app. we can configure it to add the CORS headers in HTTP responses so browsers allow cross-origin requests.

How CORS works in Spring (Refer Question No. 378, Spring 126)

330 Spring 79: What is OAuth2 and how does it differ from JWT?

OAuth2 is an authorization framework that allows a user to grant a third-party application limited access to their resources without sharing credentials.

- It is not an authentication protocol by itself, though it is often used with OpenID Connect (OIDC) for authentication.
- OAuth2 defines roles and flows for issuing access tokens to clients.
- It is widely used for APIs, social logins, and microservices.

Core components in OAuth2:

1. **Resource Owner:** The person who owns the data or resources being accessed (e.g., the Google account or Facebook profile).
2. **Client:** The app or service that requests access to the user's resources (e.g., a travel booking app that wants access to your Google Calendar).
3. **Authorization Server:** The server responsible for authenticating the user and issuing access tokens (e.g., Google's OAuth server).
4. **Resource Server:** The server that hosts the user's data (e.g., Google, Facebook, Twitter). It verifies and grants access to the requested resources based on the token provided.

Flows / Grant Types:

- **Authorization Code:** Web apps redirect users to login at the provider.
- **Client Credentials:** Service-to-service communication.
- **Password Grant:** User provides username/password directly to client (less secure, rarely used).
- **Implicit Grant:** For single-page apps (now discouraged).

Where as

- JWT (JSON Web Token) is a token format, not a framework.
- JWT can be used inside OAuth2 as the access token.

Understand with example:

Scenario: A third-party travel app wants to access your Google Calendar to book trips.

1. Resource Owner: It is the user, the Google account holder. The user owns the calendar data.
2. Client: The third-party travel booking app that wants to read the user calendar to avoid scheduling conflicts.
3. Authorization Server: Google's OAuth server. It asks the user to log in and consent: "Do you allow the travel app to read your calendar?"
4. Resource Server: Google Calendar API. Once the app has a valid access token, this server returns the user calendar data to the travel app.

Flow in Simple Terms

1. Travel app redirects us to Google login page (Authorization Server).
2. You log in and consent.
3. Google issues an access token to the travel app.
4. Travel app calls Google Calendar API (Resource Server) with the token.
5. Resource server validates the token and sends back your calendar data.

331 Spring 80: How do you implement OAuth2 in Spring?

Reference: [#39 Spring Security | Google and Github Login using OAuth2](#)

Code Link: [SpringBoot-Oauth2-Demo/Oauth2-springdemo at master · VarshaDas/SpringBoot-Oauth2-Demo](#)

332 Spring 81: What is role-based access?

Role-Based Access Control (RBAC) is a way to manage user permissions based on roles. Instead of assigning permissions directly to each user, we assign roles (like ADMIN, USER, MANAGER) to users, and each role has specific permissions.

How it Works

1. **Define Roles** For example: ADMIN, USER, EDITOR.
2. **Assign Permissions to Roles:** Each role has specific access rights.
 - ADMIN, can create, read, update, delete everything.
 - USER, can only read their own data.

3. **Assign Roles to Users** A user can have one or multiple roles.
4. **Check Access** When a user tries to access a resource, the system checks if the user's role allows that action.

Example in Real Life

- **Let's say in Banking App:**

USER view account balance, transfer money.

ADMIN approve loans, manage accounts, view all users.

- A user logs in; Spring Security populates their roles (authorities) in the SecurityContext.
- When the user tries to access /admin/dashboard, the system checks if they have ROLE_ADMIN.

Implementation in Spring Security

- Each role is represented as an authority (GrantedAuthority).
- We can secure endpoints using:

Method-level security: @PreAuthorize("hasRole('ADMIN')")

HTTP configuration:

```
http.authorizeHttpRequests()
  .requestMatchers("/admin/**").hasRole("ADMIN")
  .anyRequest().authenticated();
  • Roles are usually prefixed with ROLE_ internally (ROLE_ADMIN).
```

Additional Questions:

1. How do you secure sensitive data in a Spring boot application that is accessed by multiple users with different roles?

Securing sensitive data in a Spring Boot application accessed by multiple users with different roles requires a multi-layered approach, primarily using Spring Security.

Authentication and Authorization with Spring Security:

- **User Authentication:** Implement robust authentication mechanisms. This can involve username/password authentication (using BCryptPasswordEncoder for secure password hashing), JWT-based authentication for stateless APIs, or integrating with external identity providers.
- **Role-Based Access Control (RBAC):** Define distinct roles (e.g., ADMIN, USER, GUEST) and assign them to users.
- **Method-Level Security:** Use annotations like @PreAuthorize, @PostAuthorize, and @Secured on service methods or controller endpoints to restrict access based on roles or custom expressions.
- **URL-Based Security:** Configure SecurityFilterChain to secure specific URL patterns based on roles.

333 Spring 82: Difference between Role and Grant?

Authority

- An authority represents a specific permission or action a user can perform.
- It is a more fine-grained concept.
- In Spring Security, it is represented by the interface GrantedAuthority.
- Examples:
 - READ_PRIVILEGE
 - WRITE_PRIVILEGE
 - DELETE_USER

2. Role

- A **role** is just a group of permissions.
- Roles are higher-level and bundle related permissions together.
- In Spring Security, roles are usually written as ROLE_<ROLE_NAME>.

Examples:

- ROLE_ADMIN might include READ_PRIVILEGE, WRITE_PRIVILEGE, DELETE_USER
- ROLE_USER might include READ_PRIVILEGE only

Reference: [Spring Security Roles: A Comprehensive Guide | Medium](#)

334 Spring 83: What is method level security?

Method-level security in Spring Security is a way to protect individual methods in the code by specifying who can access them. Instead of securing only URLs or endpoints, we can secure specific service or controller methods.

We need to enable method-level security in the configuration class with

`@EnableGlobalMethodSecurity(prePostEnabled = true)` , Before (Spring Security 5.x ,From 6.x it is deprecated.

`@EnableMethodSecurity` from After Spring Security 6.

We annotate methods with security annotations:

- `@PreAuthorize`: Check before executing the method.
- `@PostAuthorize`: Check after executing the method.
- `@Secured`: Basic role-based check before method execution.
- `@RolesAllowed`: Standard JSR-250 role-based check.

335 Spring 84: What is Spring Boot Actuator, and why is it useful?

Spring Boot Actuator is a sub-project of Spring Boot that provides production-ready features to monitor and manage the application.

- It adds a set of built-in endpoints that expose information about the application, like health, metrics, environment properties, and more.
- These endpoints help to understand, monitor, and manage the app without writing extra code.

Key Features

1. **Health Checks:** Verify if the app and its dependencies (database, messaging queues, etc.) are working properly.
2. **Metrics:** Access information about memory usage, CPU, active threads, HTTP request statistics, etc.
3. **Environment & Config Info:** View active profiles, properties, and configuration settings.
4. **Logging:** Check and change logging levels at runtime.
5. **Tracing & Auditing:** Track HTTP requests and audit application events (with optional integrations).

How do you check if a running service is up in microservice

We can REST endpoint like /health or /actuator/health.

336 Spring 85: How do you enable and configure Actuator endpoints?

1. Add Dependency

Add the Actuator dependency in the pom.xml (Maven) or build.gradle (Gradle):

Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2. Enable Endpoints

- By default, **only a few endpoints are exposed** (/actuator/health, /actuator/info).
- We can enable more endpoints in application.properties or application.yml.

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

# Exclude certain endpoints if needed
management.endpoints.web.exposure.exclude=env,beans
```

337 Spring 86: List out some important Actuator endpoints?

1. /actuator/health

This endpoint checks the health of the application. It tells if the app is running fine, DB is connected, disk space is available, etc.

2. /actuator/info

This endpoint shows application info that we configure in application.properties (like version, description).

3. /actuator/metrics

The `/actuator/metrics` endpoint provides various application metrics. For example, to see JVM memory usage, we can query: localhost:8080/actuator/metrics/jvm.memory.used

4. /actuator/beans: Displays a complete list of all Spring beans in the application.

Indirect Question:

1. How do you check how many times each API endpoint was called, along with timings?

We need to /actuator/metrics/http.server.requests

For a particular API: actuator/metrics/http.server.requests?tag=uri:/hello-world&tag=method:GET

Reference: [A Comprehensive Guide to Spring Boot Actuator | by Pratik T | Medium](#)

338 Spring 87: What is Micrometer?

Micrometer is basically the metrics collection library that Spring Boot (and many other frameworks) use under the hood. Think of it as the middleman between the code and monitoring systems like Prometheus.

Here is the thing:

- Spring Boot Actuator doesn't collect metrics all by itself.
- Instead, it uses Micrometer as the core library to record, manage, and expose metrics.
- Micrometer gives a vendor-neutral API , Means We can switch between different monitoring systems (like Prometheus, Datadog, New Relic, AWS CloudWatch, etc.)

339 Spring 88: How to create Custom Endpoint in Actuator?

We can create our own custom actuator endpoints using **@Endpoint** annotation on a class. Then we have to use **@ReadOperation**, **@WriteOperation**, or **@DeleteOperation** annotations on the methods to expose them as actuator endpoint bean.

```
@Component
@Endpoint(id = "custominfo")
public class CustomInfoEndpoint {
```

```

@ReadOperation
public Map<String, Object> customInfo() {
    Map<String, Object> data = new HashMap<>();
    data.put("appName", "Payment-Service");
    data.put("version", "1.0.3");
    data.put("activeUsers", 153);
    data.put("lastDeployed", "2025-08-20 14:30:00");
    return data;
}
}

```

The above code creates a custom actuator endpoint “custominfo”.

1. We should register this endpoint in application.properties

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

2. When we access it ` <http://localhost:8080/actuator/custominfo> ` , It gives

```
{
  "appName": "Payment-Service",
  "version": "1.0.3",
  "activeUsers": 153,
  "lastDeployed": "2025-08-20 14:30:00"
}
```

Real time use case:

Example: If we want to see Business Metrics of an Application like Number of active users, current load, pending transactions.

Instead of DB checking in the DB or writing SQL, we can write /actuator/activeUsers.

```

@Component
@Endpoint(id = "activeUsers") // Exposed at /actuator/activeUsers
public class ActiveUsersEndpoint {

    @ReadOperation
    public Map<String, Object> activeUsers() {
        Map<String, Object> metrics = new HashMap<>();

        // Pretend fetching from service or cache
        int activeUsers = 125;
        int pendingTransactions = 48;
        double currentLoad = 0.67; // like CPU/memory load, etc.

        metrics.put("activeUsers", activeUsers);
        metrics.put("pendingTransactions", pendingTransactions);
        metrics.put("currentLoad", currentLoad);

        return metrics;
    }
}

```

<http://localhost:8080/actuator/activeUsers> This will give all the metrics

340 Spring 89: How to secure Actuator endpoints and why to secure?

Actuator exposes very sensitive information about the application:

- /actuator/health: can tell if the app is up/down (attackers can use this).
- /actuator/metrics: exposes internal performance metrics.
- /actuator/env or /actuator/configprops: might reveal, API keys, or DB URLs if not properly masked.
- Custom endpoints: could leak business data if left open.

If we don't secure them, anyone with the URL can hit those endpoints and can use the information. That is why in production we always secure actuator endpoints.

To secure them, we need to use SecurityFilterChain in Spring security

```
@Configuration
@EnableWebSecurity
public class ActuatorSecurityConfig {

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/actuator/health", "/actuator/info").permitAll()
                .requestMatchers("/actuator/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .httpBasic();
        return http.build();
    }
}
```

Here:

- /actuator/health and /actuator/info are public (safe for probes).
- All other Actuator endpoints need **ADMIN role**.
- Authentication is via **basic auth**.

341 Spring 90: What is AOP?

AOP is a programming technique where we separate cross-cutting concerns from the main business logic.

Cross cutting concerns are nothing but

- Logging
- Security checks
- Transactions
- Caching
- Performance monitoring

If we write these inside every class/method, our code gets messy and repetitive.

So instead of writing the same logging or security code in every method, we write it in separately.

Our business logic stays in different files and cross cutting concerns stays in separate files, both can be managed separately

Example: Let's say we have a payment service where we want to log when Payment started and after payment ended.

Without AOP: (**Note:** Used System.out.print() just for example purpose, In prod we should not use it.)

```
class PaymentService {
    public void processPayment() {
        System.out.println("Starting payment..."); // logging
        // business logic
        System.out.println("Payment done."); // logging
    }
}
```

With AOP:

```
class PaymentService {
    public void processPayment() {
        // only business logic
    }
}

@Aspect
class LoggingAspect {
    @Before("execution(* PaymentService.*(..))")
    public void logBefore() {
        System.out.println("Starting payment...");
    }

    @After("execution(* PaymentService.*(..))")
    public void logAfter() {
        System.out.println("Payment done.");
    }
}
```

Here the logging concern is completely separate and placed in something called **@Aspect** (Will see next questions).

Reference: [Spring AOP Explained: How to Implement Aspect-Oriented Programming in Your Spring Application | by Alex Klimenko | Medium](#)

Indirect Questions:

1. When will you use the AOP?

When we want to separate cross-cutting concerns from the main business logic.

2. What are cross cutting concerns?

3. How you implement AOP in Spring boot application?

To implement AOP, start with including the *spring-boot-starter-aop* module in the application dependencies.

4. What is `@EnableAspectJAutoProxy` and when to use?

We use `@EnableAspectJAutoProxy` when we need to enable Spring's ability to automatically create proxies for the beans. It used to implement AOP features like `@Transactional`, `@Secured`, or the own custom `@Aspect` classes.

In practice, we often don't need to use it explicitly because many Spring modules (like Spring Boot, Spring Transaction Management) already auto-configure it for us.

In a Non-Boot, Java-based Configuration Application, if we are building a standard Spring application (not using Spring Boot) with `@Configuration` classes, and we want to use AOP, then we must add this annotation to one of the configuration classes.

Code Link: [Spring-Samples/aop at main · CodingLyf-Fullstack/Spring-Samples](#)

Run this code and hit `GET /users/1`. You will see the output

```
Around (Before): getUserId
Before method: getUserId | Args: [1]
After method: getUserId | Returned: CodingLyf
Around (After): getUserId
```

342 Spring 91: What are the core components of AOP?

Core components of AOP:

- Aspect.
- Join Point.
- Point Cut.
- Advice.
- Weaving.
- Proxy.

Aspect: It is basically a class where we define cross-cutting code. Ex: The class where our log statements exist.

```

class PaymentService {
    public void processPayment() {
        // only business logic
    }
}

@Aspect
class LoggingAspect {
    @Before("execution(* PaymentService.*(..))")
    public void logBefore() {
        System.out.println("Starting payment...");
    }

    @After("execution(* PaymentService.*(..))")
    public void logAfter() {
        System.out.println("Payment done.");
    }
}

```

Here, LoggingAspect is the **Aspect**.

Join Point: A specific point in the application, like a method execution or exception handling, where we can apply additional behaviour.

In the above example, `processPayment()` is a Join Point because we applying logging there in the method execution

Advice: The actual code inside the aspect that runs at certain points. (or Action taken by an aspect)

Types of advice:

1. **@Before:** Runs before the method execution.
2. **@After:** Runs after the method execution.
3. **@Around:** Wraps the method, running both before and after. This is the most powerful advice type. It surrounds a join point, such as a method invocation, and has the ability to prevent the actual method execution. It takes a ProceedingJoinPoint as a parameter, which is useful to execute the target method. By calling the proceed() method on the ProceedingJoinPoint, we can proceed with the original method execution.
4. **After Returning:** Runs only if the method completes successfully. We can use it for Auditing or logging after methods execution or resource clean up.
5. **After Throwing:** Runs only if the method throws an exception.

```

@Before("execution(* PaymentService.processPayment(..))")
public void logBefore() {
    System.out.println("Payment starting...");
}

```

Here `logBefore()` is an advice which runs before the method execution.

Pointcut: A rule that decides where our extra code (aspect) should run.

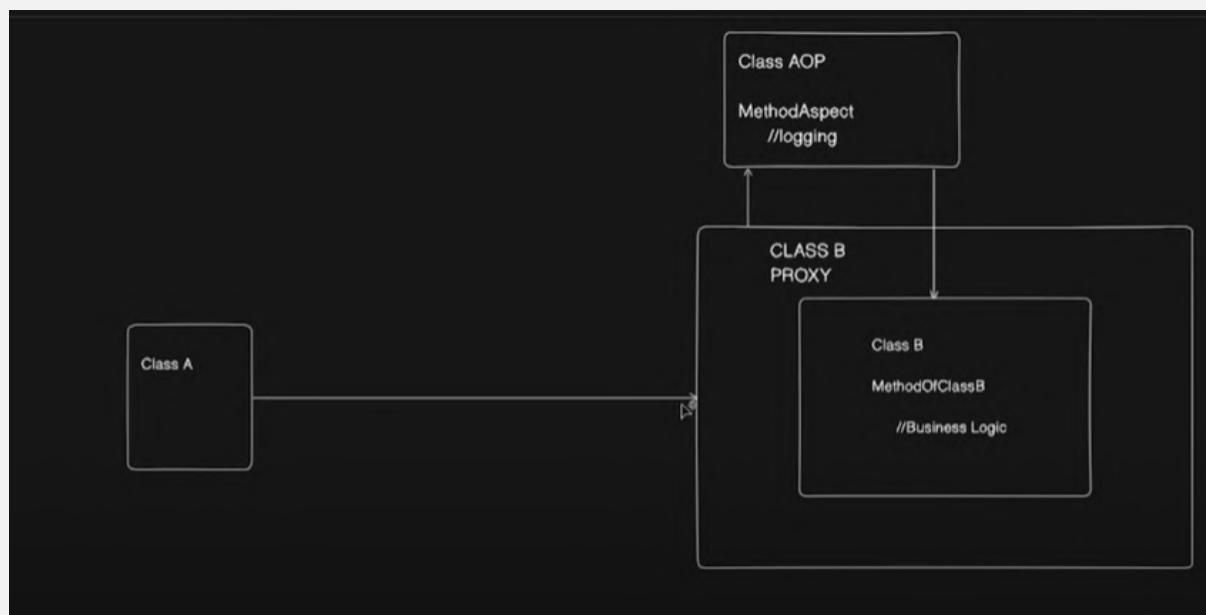
If we say `execution(* PaymentService.*(..))`, it means "apply this aspect to all methods in `PaymentService`."

Weaving: The process of applying aspects to the target objects. Spring does this dynamically at runtime using proxies.

Proxy: This is very important in Spring. All the cross-cutting concerns use Proxy. We will see it as separate question (Refer Question No 362 Spring 110. then go to next question)

343 Spring 92: How proxy works in AOP?

Let's understand with an example:



Here we have Class A, calling a method in classB and class AOP has a method which is responsible for logging.

1. Class A

- This is a normal class in the application.
- It calls a method of Class B (let's say `methodOfClassB()`).

2. Class B

- This contains the business logic. For example, it may process payments, calculate discounts, etc.
- Normally, Class A would just call Class B directly.
- But with AOP, it doesn't call Class B directly; instead, it calls a proxy of Class B.

3. Class B Proxy

- A proxy is like a wrapper around Class B.
- When Class A makes a call, it doesn't go straight to Class B.
Instead, it goes to the proxy, which has the power to Decide whether to call the actual method in Class B.
 - Add extra behaviour before or after the call.
- In this case, that extra behaviour is logging.

4. Class AOP (Aspect-Oriented Programming)

- This defines the Aspect, it does the extra behaviour we want to apply.
- In the diagram, the Aspect is logging (MethodAspect > logging).
- The logging aspect will run before or after methodOfClassB() executes.

5. Flow of execution

1. Class A calls methodOfClassB() : but it hits the proxy instead of the real class.
2. The proxy checks if any aspect is attached (here: logging).
3. The logging aspect runs (e.g., "Method started" / "Method ended").
4. Then the proxy calls the real method in Class B to execute business logic.
5. Optionally, the aspect can also run after the method (e.g., log execution time, exceptions).

344 Spring 93: Which classes or methods in Spring cannot be declared as final, and why??

Spring creates proxies for certain features like @Transactional, @Async, @Configuration, and AOP in general.

Why final matters in Spring

Spring often uses CGLIB proxies to add extra behaviour to the classes at runtime. CGLIB works by subclassing the class and overriding methods.

If a class or method is final, it cannot be subclassed or overridden, so CGLIB cannot apply its proxy. That breaks things like:

- @Transactional
- @Async
- @Cacheable, @CacheEvict
- @Configuration (when calling @Bean methods internally)

Rules

1. Classes

- We cannot make a class final if Spring needs to create a proxy for it (like a @Service or @Configuration with transactional beans).
- We can make it final if no proxying is needed (simple component with no AOP features).

2. Methods

- Any method annotated with @Transactional, @Async, @Cacheable, etc., cannot be final, because the proxy needs to override it.
- Final methods are ignored by proxies.

```
@Service
@Transactional
public final class BankService { // final class breaks transactional proxy
    public void transferMoney() { ... } // if method is final, proxy cannot intercept
}
```

```
@Service
@Transactional
public class BankService { // class is non-final
    public void transferMoney() { ... } // method is non-final
}
```

345 Spring 94: What is Self-Invocation Problem?

In Spring AOP, aspects (like logging, transactions, security) are applied using proxies. But if a method inside the same class calls another method of that class, the proxy is bypassed, and the aspect will not run.

That is the self-invocation problem.

Example

```
@Service
public class PaymentService {

    @Transactional // AOP-based annotation
    public void processPayment() {
        System.out.println("Processing payment...");
        saveTransaction(); // calling another method inside same class
    }

    @Transactional
    public void saveTransaction() {
        System.out.println("Saving transaction...");
    }
}
```

Now when we call: paymentService.processPayment();

We might expect both `processPayment()` and `saveTransaction()` to run inside a transaction.
But here is the catch:

- `processPayment()` is called through the proxy so Transaction applies.
- `saveTransaction()` is called directly inside the same class, not through the proxy so transaction does not apply.

So `saveTransaction()` runs without the `@Transactional` aspect.

Why does this happen?

- Spring AOP works by wrapping the bean with a proxy.
- When an external class calls `paymentService.processPayment()`, the proxy intercepts the call.
- But when one method in the class calls another (self-invocation), it goes straight to the real object (`this.saveTransaction()`), by skipping the proxy.

Ways to Fix It

1. **Move the method to another bean (class)**

- So, the call goes through the proxy.

2. **Use AopContext to call through proxy**

```
((PaymentService) AopContext.currentProxy()).saveTransaction();
```

Requires `@EnableAspectJAutoProxy(exposeProxy = true)` in config.

346 Spring 95: How to do Scheduling in Spring boot?

In Spring, scheduling basically means running tasks automatically at fixed times, intervals, or according to cron expressions. Spring makes this super simple with its scheduling support.

1. Enable Scheduling

First, we need to turn on scheduling in the Spring Boot app with `@EnableScheduling`.

```
@SpringBootApplication
@EnableScheduling // Enables scheduling
public class MyApp {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}
```

2. Use `@Scheduled Annotation`

Now we can mark methods that should run on a schedule. Spring will run them in the background automatically.

3. Scheduling Strategies

We have Scheduling Strategies like:

- fixedDelay,
- fixedRate
- and using cron expressions.

```
@Scheduled(cron = "0 0 9 * * ?") // every day at 9 AM
public void morningTask() {
    System.out.println("Good Morning! It is 9 AM");
}
```

Indirect Question:

1. You have configured scheduled task only once but it is running twice. What might be the issue?

Multiple application instances: if the app is running on multiple servers or multiple Spring contexts, each instance runs its own scheduler.

Use Distributed cache lock like Redisson's RLock to ensure only one instance runs the task.

Multiple bean definitions: if the same @Scheduled method is present in multiple beans or contexts, it runs multiple times.

2. You have configured scheduled but it is not running. What might be the issue?

- @EnableScheduling is missing in the configuration.
- The scheduled method must be public and inside a Spring-managed bean (e.g., annotated with @Component, @Service, or defined as a @Bean). If it is private, protected, or in a class Spring doesn't manage, the scheduler won't detect or run it.
- The cron/fixedRate/fixedDelay expression is incorrect or misconfigured.

347 Spring 96: What is the difference between fixedDelay and fixedRate?

fixedDelay: This is used to give specific delay between the completion of one task execution and the start of the next.

- The next run starts only after the current run finishes + the delay time.
- Example: If a task takes 4 seconds and delay is 3 seconds: then the next run starts at 7 seconds.
- Use Case: Good for tasks where execution time may vary (like sending emails, file processing, DB cleanup) and we always want a gap between runs.

fixedRate: This property schedules the task to run at a consistent, fixed interval, regardless of how long the previous execution took.

- Example: If rate is 5 seconds, it runs at 0s, 5s, 10s, 15s... even if the previous task is still running.
- Use Case: Best for tasks that need **strict periodic execution**, like health checks, monitoring, refreshing cache, or fetching data at fixed times.

348 Spring 97: How can you make tasks run in parallel?

By default, Spring runs all @Scheduled tasks in a single thread. That means if one task takes time, the next one waits. To make them run in parallel, we need to give Spring a thread pool for scheduling.

We can use ThreadPoolTaskScheduler, a scheduler that supports various scheduling options and manages a thread pool for task execution. The ThreadPoolTaskScheduler uses a java.util.concurrent.ScheduledThreadPoolExecutor to manage a pool of threads.

We can configure the ThreadPoolTaskScheduler to suit the application's needs. Some common configuration options include:

- **Pool size:** The number of threads in the thread pool
- **Thread name prefix:** A prefix for thread names, useful for debugging
- **Wait for tasks to complete on shutdown:** A flag to indicate whether the scheduler should wait for scheduled tasks to complete before shutting down

```
@Configuration
@EnableScheduling
public class SchedulerConfig {
    @Bean
    public TaskScheduler taskScheduler() {
        ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();
        scheduler.setPoolSize(5);
        scheduler.setThreadNamePrefix("MyTaskScheduler-");
        scheduler.setWaitForTasksToCompleteOnShutdown(true);
        return scheduler;
    }
}
```

Tasks:

```
@Component
public class MyTasks {

    @Scheduled(fixedRate = 2000)
    public void task1() throws InterruptedException {
        System.out.println(Thread.currentThread().getName() + " running task1");
        Thread.sleep(4000); // simulate long work
    }

    @Scheduled(fixedRate = 2000)
    public void task2() {
        System.out.println(Thread.currentThread().getName() + " running task2");
    }
}
```

Without the thread pool: task2 will be delayed until task1 finishes.

With the thread pool: both can run at the same time.

349 Spring 98: How can we handle scheduled task failures?

1. Try-Catch Inside the Task

The simplest way, just wrap the task logic with try-catch and log or recover gracefully.

2. Use a Custom ErrorHandler

Spring provides an **ErrorHandler** interface to handle exceptions thrown out of @Scheduled methods. We can create a custom error handler by implementing this interface:

```
@Component
public class MyErrorHandler implements ErrorHandler {

    @Override
    public void handleError(Throwable t) {
        // Custom logic for handling errors
        System.err.println("Error encountered in scheduled task: " + t.getMessage());
    }
}
```

To use this custom error handler, we would need to set it in the task scheduler.

```
@Configuration
@EnableScheduling
public class SchedulerConfig {

    @Bean
    public ThreadPoolTaskScheduler taskScheduler() {
        ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();
        scheduler.setPoolSize(5);
        scheduler.setThreadNamePrefix("scheduler-");
        scheduler.setErrorHandler(new MyErrorHandler()); //Error Handler
        return scheduler;
    }
}
```

3. Add Retry Logic (Spring Retry)

If a task should retry automatically on failure, we can integrate Spring Retry (Refer Question No. 379 Spring 127 to know more about this)

350 Spring 99: How to run a task in the background in Spring boot?

In Spring Boot, if we want to run a task in the background (async execution), we typically use @Async.

Step 1: Enable Async Support

In the main application class (or any config class), enable async execution:

```
@Configuration
```

```
@EnableAsync
public class Config {

}
```

Step 2: Mark Methods with @Async

When we annotate a method with @Async, Spring will execute it in a separate thread (it won't block blocking the main thread)

```
@Service
public class Service {

    @Async
    public void process(String userEmail) {
        try {
            Thread.sleep(5000); // simulate delay
            System.out.println("Delayed task");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Real-world uses of @Async

- Sending emails, SMS, or push notifications.
- Triggering heavy reports.
- Calling external APIs that can run in parallel.
- File uploads, image processing, or background data syncs.

Reference: [Spring Boot @Async Controller with CompletableFuture](#)

Indirect Question

1. CompletableFuture vs @Aysnc

CompletableFuture is Java 8 feature (`java.util.concurrent.CompletableFuture`) for asynchronous computation. It runs task in the background and handle results, exceptions, or chaining multiple tasks.

Key Points:

- It is Fully programmatic; we create, execute, and combine futures yourself.
- It supports chaining, combining, and callback methods (`thenApply`, `thenAccept`, `exceptionally`).
- It Works in any Java class, not just Spring beans.

@Async is a Spring annotation that enables a method to be executed in a separate thread, managed by Spring's task execution infrastructure.

It provides a convenient way to mark a method for asynchronous execution and here we don't need manually manage the threads.

2. How Spring manages the thread pool in @Async task?

Spring manages @Async tasks using a **TaskExecutor** (default is SimpleAsyncTaskExecutor), which handles a thread pool to run methods asynchronously. We can customize the pool size and behaviour by defining a @Bean of type Executor. Spring submits each @Async method call to this executor, allowing it to run in a separate thread without blocking the main thread.

```
@Bean(name = "taskExecutor")
public Executor taskExecutor() {
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(5);          // Minimum threads
    executor.setMaxPoolSize(10);         // Maximum threads
    executor.setQueueCapacity(25);        // Queue size
    executor.setThreadNamePrefix("AsyncThread-");
    executor.initialize();
    return executor;
}
```

Reference: [How Spring Boot Configures Thread Pools | Medium](#)

351 Spring 100: How @Async works behind the scenes?

When we put @Async on a method in Spring Boot, here is what actually happens behind the scenes:

1. Spring creates a proxy (Refer Question no 362 Spring 110 to know more about the proxy)

- When the class has a method annotated with @Async, Spring doesn't call the method directly.
- Instead, it creates a proxy object (using JDK dynamic proxies or CGLIB, depending on the class).
- The proxy intercepts method calls and decides how to execute them.

2. Intercepts the call

- When we call a method, instead of running it on the caller's thread, the proxy hands the task over to a **TaskExecutor** (basically a thread pool).
- By default, spring uses SimpleAsyncTaskExecutor if no executor is provided, but in real projects we usually configure the own ThreadPoolTaskExecutor.

3. Method execution happens on a different thread

- The proxy submits the method as a task (Runnable or Callable) to the executor.
- The original caller thread **doesn't wait** for the task to finish (unless we return Future/CompletableFuture).
- This is how it becomes asynchronous.

4. Return type decides behavior

- void: fire and forget. Caller has no clue if the task succeeded/failed.
- Future / CompletableFuture : Caller can track progress, get results, or handle exceptions.

Point to remember:

Due to the proxy-based nature of Spring's AOP framework, if we call an @Async method from another method within the same class, it will not actually execute asynchronously. This is because the call won't go through the proxy but it is just a direct method call. Always be aware of this limitation. (Self-invocation problem Refer Question 346 No. Spring 94)

```
@Service
public class MyAsyncService {

    @Async
    public void asyncVoidTask() throws InterruptedException {
        System.out.println(Thread.currentThread().getName() + " running void task");
        Thread.sleep(1000);
        System.out.println(Thread.currentThread().getName() + " finished void task");
    }

    @Async
    public CompletableFuture<String> asyncFutureTask() throws InterruptedException {
        System.out.println(Thread.currentThread().getName() + " running future task");
        Thread.sleep(1000);
        System.out.println(Thread.currentThread().getName() + " finished future task");
        return CompletableFuture.completedFuture("Task Done");
    }
}
```

In this example, first method is not returning anything whereas second method is returning CompletableFuture

```
@GetMapping("/runAsync")
public String run() throws Exception {
    service.asyncVoidTask(); // fire-and-forget
    CompletableFuture<String> result = service.asyncFutureTask(); // trackable
    System.out.println("Controller thread: " + Thread.currentThread().getName());
    return "Async tasks triggered";
}
```

352 Spring 101: Can you run Database transactional tasks in Async?

It takes too long to write here. I would suggest you to go through the link, worth reading.

Link: [Handling Asynchronous Execution with Transactions in Spring: A Common Pitfall and How to Solve It - DEV Community](#)

353 Spring 102: How cache works in spring boot?

Caching in Spring Boot is a mechanism to improve application performance by storing frequently accessed data in a temporary, faster-access storage location, such as in-memory or a dedicated cache server. This reduces the need to repeatedly fetch data from slower sources like databases or external APIs.

How it works:

Enabling Caching:

Caching is enabled in a Spring Boot application by adding the `@EnableCaching` annotation to a configuration class or the main application class. This activates Spring's caching infrastructure.

Cache Abstraction:

Spring provides a cache abstraction layer that allows developers to use various cache providers (e.g., Caffeine, Ehcache, Redis) without changing the core application logic.

354 Spring 103: How `@cachable` works?

`@Cacheable` is a Spring annotation used to **store the result of a method call in a cache**, so the next time we call that method with the same arguments, Spring will fetch the result directly from the cache instead of executing the method again.

Basically, it saves computation time and database calls.

```
@Service
public class ProductService {

    @Cacheable(value = "products", key = "'allProducts'")
    public Product getProductId(Long id) {
        // Imagine this is a slow DB call
        System.out.println("Fetching from DB for id: " + id);
        return new Product(id, "Laptop");
    }
}
```

`value = "products"` is the cache name where Spring will store and look up the cached data
`key = "'allProducts'"` means Spring will always use the string "allProducts" as the cache key

355 Spring 104: What is the importance of the key while caching?

- Keys are critical in caching because they uniquely identify the cached data.
- When we call a cached method, Spring creates a key based on the method parameters (by default all arguments are considered).
- If two calls have the same key, Spring returns the cached result instead of executing the method.
- Without proper keys, data could be incorrectly reused or duplicated. We can customize keys using SpEL (`@Cacheable(key = "#id")`) to cache only the values we need.
- Good key strategy ensures accurate retrieval, avoids collisions, and optimizes memory usage for the cache.

356 Spring 105 Can you cache based on the condition?

Yes, Spring allows conditional caching with the `condition` and `unless` attributes.

- **Condition:** Evaluated before method execution, decides whether to cache.
- **Unless:** Evaluated after execution, decides whether to store the result.

```
@Cacheable(value = "products", key = "#id", condition = "#id > 10")
public Product getProduct(Long id) {
}
```

Here, caching happens only if id > 10.

```
@Cacheable(value = "products", unless = "#result == null")
public Product getProduct(Long id) {
}
```

This prevents caching null results.

357 Spring 106: How to update the cache?

When we want to refresh the cache with a new value after updating data, use @CachePut. Unlike @Cacheable, it always executes the method and updates the cache with the new result.

Example:

```
@CachePut(value = "products", key = "#product.id")
public Product updateProduct(Product product) {
    // Save to DB
    return productRepository.save(product);
}
```

Whenever this method runs, the cache is updated with the latest product, ensuring consistency between DB and cache.

358 Spring 107: How to delete the cache?

To remove data from the cache, use @CacheEvict. This is important when cached data is stale or invalid.

```
@CacheEvict(value = "products", key = "#id")
public void deleteProduct(Long id) {
    productRepository.deleteById(id);
}
```

This removes the cache entry for the given product ID.

We can also clear the entire cache:

```
@CacheEvict(value = "products", allEntries = true)
public void clearCache() {}
```

359 Spring 108: How cache work behind the scenes?

When Spring Boot starts, it creates a CacheManager bean. @Cacheable methods are wrapped in a proxy. When the method is called:

1. Proxy checks the cache using the generated key.
2. If a value is found, it returns it immediately (method body is skipped).
3. If not found, the method executes, result is stored in the cache, and then returned.
The actual storage is managed by the configured cache provider (ConcurrentMap, Redis, EhCache, etc.).

The default CacheManager is ConcurrentHashMapCacheManager

360 Spring 109: How CocurrentMapCacheManager works?

ConcurrentMapCacheManager is Spring's default cache manager. It uses Java's in-memory **ConcurrentHashMap** to store cached data. It is lightweight, thread-safe, and requires no external configuration. Each cache is basically a ConcurrentHashMap.

```
@Bean
public CacheManager cacheManager() {
    return new ConcurrentHashMapCacheManager("products", "users");
}
```

It is great for development and testing, but not recommended for production because the cache is limited to the JVM, not distributed, and is lost when the app restarts.

Point to remember: For production, Redis or Caffeine is usually preferred.

361 Spring 110: What is proxy and how does it work Spring?

A proxy is like a middleman (or wrapper) that Spring creates around the actual object (the target).

When we call a method, we are not calling the real object directly but we are calling the proxy.

The proxy can do extra work before or after calling the real method.

Point to remember: This is how Spring applies proxies like AOP, transactions, security, or caching without writing extra code in our class.

Let's see an example:

We have an interface called 'Service' and we created Payment Service class implementing the 'Service'.

```
interface Service {
    void processPayment();
}

class PaymentService implements Service {
    public void processPayment() {
        System.out.println("Processing payment...");
    }
}
```

```
    }  
}
```

When we call new PaymentService().processPayment(); Spring creates a proxy for PaymentService like this,

```
//Proxy class  
class PaymentServiceProxy implements Service {  
    private final PaymentService target;  
  
    public PaymentServiceProxy(PaymentService target) {  
        this.target = target;  
    }  
  
    @Override  
    public void processPayment() {  
        System.out.println("Before: Logging before payment...");  
        target.processPayment(); // call the real object  
        System.out.println("After: Logging after payment...");  
    }  
}
```

And this is how Spring calls:

```
Service service = new PaymentServiceProxy(new PaymentService());  
service.processPayment();
```

In our example, PaymentServiceProxy has been created and it will pass the real service to the proxy. Now Proxy can do extra code before or after the original method (processPayment()) execution.

There are two types of proxies in Spring:

1. JDK Dynamic Proxies (Requires an Interface)

If the target class implements an interface, Spring will create a dynamic proxy that implements the same interface.

2. CGLIB Proxies (No Interface Required)

If there is no interface, Spring generates a subclass proxy using CGLIB (Code Generation Library).

Reference: [Understanding the Proxy Pattern in Spring Boot with Practical Insights | by Davoud Badamchi | Medium](#) (It contains how proxies work in AOP, @Transactional and Caching).

362 Spring 111: How a CGLIB Proxy works with @Bean?

When we mark a class with `@Configuration`, Spring doesn't just use it as-is. Behind the scenes, Spring creates a CGLIB-enhanced subclass of that class. This proxy ensures that calls to `@Bean` methods inside the same configuration class still return the same singleton bean from the container, not a new object.

Example:

```
@Configuration
public class AppConfig {

    @Bean
    public UserService userService() {
        return new UserService(orderService()); // direct method call
    }

    @Bean
    public OrderService orderService() {
        return new OrderService();
    }
}
```

At first glance, we might think:

- Every time `userService()` calls `orderService()`, a new `OrderService` is created.
- But Spring wants `OrderService` to be a singleton by default

What actually happens

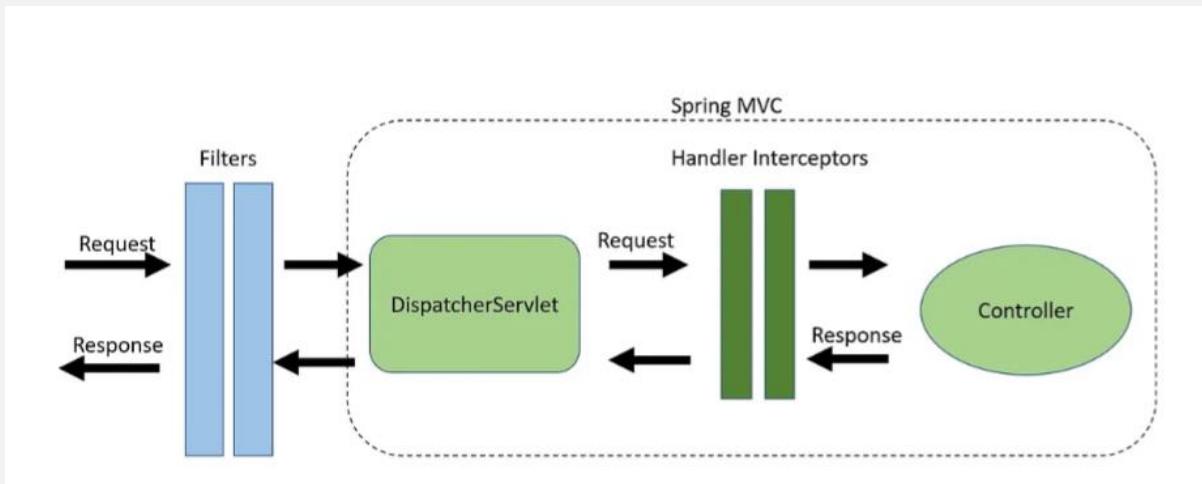
Spring uses **CGLIB** to generate a subclass of `AppConfig`.

- When `userService()` calls `orderService()`, it doesn't directly execute the method.
- Instead, the CGLIB proxy intercepts the call, checks the container, and returns the already cached singleton `OrderService` bean.

So even though it looks like a normal method call, it is actually routed through Spring's bean factory.

363 Spring 112: What is filter in Spring boot and how it will used?

In Spring Boot, a Filter is a component that intercepts and processes HTTP requests and responses at the web container level, before they reach the Spring DispatcherServlet and after the DispatcherServlet has handled them. Filters are part of the Java Servlet API and are used to implement cross-cutting concerns in web applications.



Common use cases include:

- **Authentication and Authorization:** Filters can verify user credentials and permissions before allowing access to resources (e.g., Spring Security utilizes filters).
- **Logging and Auditing:** Recording request details, response status, and other relevant information for monitoring and debugging.
- **Request/Response Modification:** Altering request headers, parameters, or response content (e.g., compression, content negotiation).
- **Data Validation:** Pre-processing requests to validate input data before it reaches the application logic.
- **CORS (Cross-Origin Resource Sharing):** Managing access control for requests from different origins.

364 Spring 113: How do you create a custom filter?

Steps to create the filter

- Implement the `jakarta.servlet.Filter` interface (or `javax.servlet.Filter` for older versions).
- Override the `doFilter()` method, which contains the logic for processing the request and response.
- Use `chain.doFilter(request, response)` to pass the request to the next filter in the chain or to the target servlet.

Registering the Filter:

- Using `@Component` and `@Order`: Annotate the filter class with `@Component` to make it a Spring bean. Use `@Order` to define the order of execution if we have multiple filters.

```
@Component
public class MyCustomFilter implements Filter {
    @Override
```

```

public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain)
        throws IOException, ServletException {
    HttpServletRequest req = (HttpServletRequest) request;
    System.out.println("Incoming request URL: " + req.getRequestURI());

    // Continue the request
    chain.doFilter(request, response);

    System.out.println("Outgoing response processed");
}
}

```

Point to remember: If we annotate the filter with @Component, Spring Boot automatically adds it to the filter chain.

Use cases of filters.

- **Authentication & Authorization:** Check JWT tokens, session IDs, or API keys.
- **Logging & Auditing:** Log request URLs, headers, or response times.
- **Compression or Transformation:** Compress responses, modify headers, or transform JSON.
- **Rate Limiting:** Block excessive requests.
- **CORS Handling:** Add headers for cross-origin requests.

Indirect Question:

1. How can you implement JWT token validation for all APIs without modifying each controller?

Use a custom filter (e.g., extending OncePerRequestFilter) and register it in the Spring Security filter chain.

2. If you want to log every incoming API call and its response time, which component would you use?

Implement a Spring Filter or HandlerInterceptor (Refer Question no 100 for Interceptor)

3. How does Spring Boot decide the order in which filters are executed?

Using @Order annotation

365 Spring 114: What is interceptor and its uses?

Interceptors in Spring Boot are components within the Spring Web MVC framework that are used to intercept and process HTTP requests and responses. They provide a mechanism to execute custom logic before a request reaches a controller, after the controller has processed the request but before the response is sent.

Interceptors are generally used for operations such as security, authentication, authorization, logging and monitoring

Both Filters and Interceptors serving the same purpose. But what is the difference between them (Refer Question No 368, Spring 116)

Key Points About Interceptors

1. Interceptors are tied to Spring MVC and work with HandlerMapping and HandlerExecutionChain.
2. They can pre-process requests, post-process responses, and handle after-completion logic used for cleanup tasks.
3. Unlike filters, interceptors have access to the controller's handler method and can examine method arguments, model data, and view.

366 Spring 115: How to create an Interceptor?

Step 1: Create a component by Implementing the HandlerInterceptor Interface

Spring provides the HandlerInterceptor interface. It has **three main methods**:

```
@Component
public class MyInterceptor implements HandlerInterceptor {

    // Runs before the controller method
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
        throws Exception {
        System.out.println("Pre-handle logic: Incoming request URL -> " +
request.getRequestURI());
        // Return true to continue to the controller, false to block the request
        return true;
    }

    // Runs after the controller method but before view rendering
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        System.out.println("Post-handle logic: Controller executed, preparing
response");
    }

    // Runs after the complete request is finished and view rendered
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
        System.out.println("After completion logic: Request finished");
        if (ex != null) {
            System.out.println("Exception occurred: " + ex.getMessage());
        }
    }
}
```

- **preHandle():**

When an interceptor is implemented, any request before reaching the desired controller will be intercepted by this interceptor and some pre-processing can be performed like logging, authentication, redirection, etc.

Must return true to continue to the controller; false stops the request.

- **postHandle():**

This method is executed after the request is served but just before the response is sent back to the client. It intercepts the request in the final stage, giving us a chance to make any final trivial adjustments.

- **afterCompletion():**

This method is executed after the request and response mechanism is completed. This method can turn out to be very useful in cleaning up the resources once the request is served completely.

Step 2: Interceptors need to be registered with Spring MVC using WebMvcConfigurer:

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    private final MyInterceptor myInterceptor;

    public WebConfig(MyInterceptor myInterceptor) {
        this.myInterceptor = myInterceptor;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(myInterceptor)
            .addPathPatterns("/**"); // Apply to all endpoints
    }
}
```

Here, we have registered myInterceptor using InterceptorRegistry.

367 Spring 116: Difference between Filter and Interceptor?

| Feature | Filter | Interceptor |
|-----------|--|--|
| API Level | Part of the Servlet API (javax.servlet.Filter) | Part of Spring MVC (HandlerInterceptor) |
| Scope | Works on all HTTP requests, even outside Spring MVC (e.g., static resources, JSPs) | Works only on requests handled by Spring MVC controllers |

| Feature | Filter | Interceptor |
|-----------------------------|--|--|
| Execution Point | Pre-processing before the request enters the servlet and post-processing after response leaves the servlet | Pre-processing before controller, post-processing after controller but before view, and after completion after view rendered |
| Access to Handler | Does not know about controllers or handler methods | Has access to handler (controller method) and ModelAndView |
| Use Cases | Cross-cutting concerns like logging, authentication, compression, CORS, security | Controller-specific tasks like logging request/response, authorization, metrics, exception handling |
| Order of Execution | Controlled by @Order or FilterRegistrationBean | Controlled by order of interceptor registration in WebMvcConfigurer |
| Dependency on Spring | Independent of Spring; works at servlet container level | Dependent on Spring MVC; integrated with Spring lifecycle |
| Example | JWT token validation before reaching any controller | Measuring execution time of controller method |

368 Spring 117: Have you written any custom Annotation, if yes how you have written it?

Annotations are created by using '@' sign, followed by the keyword interface, and followed by annotation name as shown in the below example.

```
public @interface EnableRestCallLogs{
}
```

Generally, each Annotation can contain meta-annotations. Both of these are not mandatory. If not applied, default values will be considered.

- @Retention
- @Target.

Retention policy(@Retention)

It specifies the Retention policy. A retention policy determines at what time annotation should be discarded. Mainly java defined 3 types of retention policies.

Those are **SOURCE**, **CLASS** and **RUNTIME**.

- The **SOURCE** policy will be retained only with source code, and discarded during compile time. It effected on before compile the source code. Besides that very common java annotations like [@Override](#), [@Deprecated](#), [@FunctionalInterface](#), [@SuppressWarnings](#) also uses retention policy as **SOURCE**.

- The retention policy **CLASS** will be retained until compiling the code, and discarded during runtime.
- Retention policy **RUNTIME** will be available to the JVM through runtime. If we not specify the retention policy the default retention policy type is **CLASS**.

Target(@Target)

- @Target annotation definition defines where to apply the annotation .
- It takes ElementType enumeration as its only argument.
- The ElementType enumeration is a constant which specifies the type of the program element declaration (class, interface, constructor, etc.) to which the annotation can be applied.

Type -> Class, interface or enumeration

Field -> Field

Method -> Method

Constructor -> Constructor

@Target(ElementType.METHOD , @Target({ElementType.METHOD, ElementType.TYPE})

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

// Custom annotation
@Retention(RetentionPolicy.RUNTIME) // Annotation should be available at runtime
@Target(ElementType.METHOD) // Can be applied only on methods
public @interface LogExecutionTime { }
```

369 Spring 118: @Configuration vs @Component?

1. @Component

- @Component is a generic stereotype annotation in Spring.
- It tells Spring, "this class is a Spring bean, please register it in the application context."
- We usually use it for service classes, repositories, controllers, or any bean we want Spring to manage.
- Methods inside a @Component class are not special by default means if we add a @Bean method here, Spring will still create a bean, but it won't guarantee singleton behaviour when one @Bean method calls another.

```
@Component
public class AppConfig {

    @Bean
    public UserService userService() {
```

```

        return new UserService(orderService()); // creates new OrderService each time
    }

    @Bean
    public OrderService orderService() {
        return new OrderService();
    }
}

```

Here, userService() will get a **new instance** of OrderService() instead of the Spring-managed one.

@Configuration

- @Configuration is a specialized form of @Component.
- It not only registers the class as a bean but also enables full configuration features.
- Specifically, it uses CGLIB proxies to intercept calls between @Bean methods and ensure singleton semantics which means even if one @Bean method calls another, we still get the same Spring-managed bean instance.

```

@Configuration
public class AppConfig {

    @Bean
    public UserService userService() {
        return new UserService(orderService()); // gets the Spring-managed singleton
    }

    @Bean
    public OrderService orderService() {
        return new OrderService();
    }
}

```

Here, userService() will get the **same OrderService instance** from the container.

Point to remember:

@Component: No proxying means calling one @Bean method from another creates a new object.

@Configuration: Proxying enabled means Spring ensures that all @Bean methods return proper container-managed singletons.

370 Spring 119: What are best techniques to handle exceptions in Spring?

1.Try-Catch Blocks (Local Handling)

- The simplest approach is to use **try-catch** within a method.
- Pros: Quick for simple cases.
- Cons: Leads to **duplicate code** if many methods need similar handling.

Example:

```

try {
    userService.deleteUser(id);
}

```

```

} catch (UserNotFoundException e) {
    // handle exception locally
}

```

2. @ExceptionHandler (Controller Level)

- It is used to handle exceptions **within a specific controller**.
- Annotate a method with `@ExceptionHandler` to intercept exceptions.
- We can return custom response objects for better API consistency.

Example:

```

@RestController
public class UserController {

    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<String> handleUserNotFound(UserNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
    }
}

```

3. @ControllerAdvice (Global Exception Handling)

- It is useful for centralized exception handling across all controllers.
- Annotate a class with `@ControllerAdvice` and define `@ExceptionHandler` methods.
- It is best for REST APIs to send uniform error responses.

```

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleUserNotFound(UserNotFoundException ex) {
        ErrorResponse error = new ErrorResponse("USER_NOT_FOUND", ex.getMessage());
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGenericException(Exception ex) {
        ErrorResponse error = new ErrorResponse("INTERNAL_ERROR", "Something went wrong");
        return new ResponseEntity<>(error, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

```

4. ResponseEntityExceptionHandler

- Spring provides a base class `ResponseEntityExceptionHandler` with default handling for common exceptions.

- We can extend it in the `@ControllerAdvice` to customize default exception handling.
- Useful for exceptions like `MethodArgumentNotValidException` (validation errors).

```
@RestControllerAdvice
public class CustomResponseEntityExceptionHandler extends ResponseEntityExceptionHandler
{

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
        MethodArgumentNotValidException ex,
        HttpHeaders headers,
        HttpStatusCode status,
        WebRequest request) {

        Map<String, Object> body = new LinkedHashMap<>();
        body.put("timestamp", LocalDateTime.now());
        body.put("status", status.value());

        List<String> errors = ex.getBindingResult()
            .getFieldErrors()
            .stream()
            .map(x -> x.getField() + ": " + x.getDefaultMessage())
            .collect(Collectors.toList());

        body.put("errors", errors);

        System.out.println(body);
        return new ResponseEntity<>(body, status);
    }

}
```

5. Custom Exceptions

- We can define our own exception classes to make the code readable and meaningful.
- Combine with `@ExceptionHandler` or `@ControllerAdvice`.

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

Indirect Questions

1. Define `@ExceptionHandler` or `@ControllerAdvice`

`@ExceptionHandler` handles exceptions inside one controller, while `@ControllerAdvice` applies globally across controllers.

2. If I define `@ExceptionHandler` for a custom exception in a controller and also in a `@ControllerAdvice`, which one gets executed and why

If both exist, the local `@ExceptionHandler` in the controller wins, because Spring gives priority to controller-scoped handlers.

3. If you don't define any `@ControllerAdvice` or `@ExceptionHandler`, what kind of response does Spring Boot return for an exception thrown in a REST API?

Without custom handlers, Spring Boot returns a default JSON error response (status, error, message, timestamp, path) using `BasicErrorController`.

4. If multiple `@ControllerAdvice` classes handle the same exception type, how does Spring decide which one to use?

If multiple `@ControllerAdvice` match, Spring picks the most specific one (by exception hierarchy or `@Order` annotation).

5. If a service method annotated with `@Async` throws an exception, will your `@ControllerAdvice` catch it? If not, how would you handle it

Exceptions from `@Async` methods don't reach `@ControllerAdvice`. We need to handle them via `AsyncUncaughtExceptionHandler`

```
public class CustomAsyncExceptionHandler implements AsyncUncaughtExceptionHandler {
    @Override
    public void handleUncaughtException(Throwable ex, Method method, Object...
params) {
        System.err.println(
            "Exception occurred in async method: " + method.getName()
            + " with message: " + ex.getMessage());
        // Log the exception, send notifications, etc.
    }
}
```

6. How do you handle Validation Errors.

Validation errors (from `@Valid`) are caught as `MethodArgumentNotValidException`. Typically, we handle them in `@ControllerAdvice` and return structured error messages (field, rejected value, reason).

```
@ControllerAdvice
public class ValidationHandler {
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>>
handleValidation(MethodArgumentNotValidException ex) {
    Map<String, String> errors = new HashMap<>();
    ex.getBindingResult().getFieldErrors()
        .forEach(e -> errors.put(e.getField(), e.getDefaultMessage()));
    return ResponseEntity.badRequest().body(errors);
}
}
```

7. How can you restrict `@ControllerAdvice` only to controller or exceptions from specific packages

We can restrict @ControllerAdvice with basePackages, basePackageClasses, or annotations so it only applies to specific controllers or packages.

```
//Applies only to controllers in package "com.example.api"
@ControllerAdvice(basePackages = "com.example.api")
public class ApiExceptionHandler {
    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handle(Exception ex) {
        return ResponseEntity.internalServerError().body("API error: " + ex.getMessage());
    }
}
```

8. What is ProblemDetails?

ProblemDetails is a standard format that is defined in [RFC 7807](#) for describing errors and exceptions in HTTP APIs. The format includes a set of predefined fields, such as the error type, error code, error message, and additional details about the error.

```
@RestControllerAdvice
public class CustomExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value = { CustomException.class })
    protected ResponseEntity<Object> handleCustomException(CustomException ex, WebRequest
request) {
        ProblemDetails problemDetails = new ProblemDetails();
        problemDetails.setStatus(HttpStatus.BAD_REQUEST);
        problemDetails.setTitle("Custom Exception");
        problemDetails.setDetail(ex.getMessage());
        return handleExceptionInternal(ex, problemDetails, new HttpHeaders(),
HttpStatus.BAD_REQUEST, request);
    }
}
```

Reference: [Using ProblemDetails in Spring Boot | by Duncan Roydon | Medium](#)

9. Mention 5 Spring Exceptions that you came across.

- **NoSuchBeanDefinitionException:** Thrown when we ask Spring for a bean that doesn't exist in the ApplicationContext.
- **NoUniqueBeanDefinitionException:** Thrown when multiple beans match a type and Spring can't decide which one to inject.
- **BeanCurrentlyInCreationException:** Happens during circular dependency issues.
- **BeanInstantiationException:** Raised when Spring fails to create a bean instance, often due to missing constructor or abstract class.
- **ApplicationContextException:** A generic runtime exception for problems starting or using the Spring ApplicationContext.

371 Spring 120: How Database indexing improves the performance?

An index in a database is a data structure (usually a B-tree or hash table) that allows faster retrieval of rows from a table. It works like an index in a book, rather than scanning the entire book to find a topic, we go to the index, find the page number, and jump directly to it.

```
SELECT * FROM users WHERE email = 'abc@example.com';
```

Without Index:

The database does a full table scan.

It checks every single row, one by one, to see if email matches.

This is very slow for large datasets.

With Index:

The database goes to the index on email, finds the location of the matching row(s) quickly, and jumps directly there.

It is like going straight to the correct shelf in a library rather than checking every book.

This is much faster, often reducing lookup time from seconds to milliseconds.

```
CREATE INDEX idx_email ON users(email);
```

Please note: If we create too many indexes, it may hurt write performance and increase storage usage.

Reference: [How Does Indexing Work | Atlassian](#)

372 Spring 121: What is externalization and how would you achieve it?

When we build applications, one of the fundamental principles is separation of configuration from code. We don't want to hardcode things like database URLs, API keys, file paths, or feature toggles directly into the Java classes. Why? Because those values often change between environments like development, testing, staging, and production.

Externalization is the practice of moving configuration values outside the application code so they can be changed without modifying or recompiling the application. Instead of editing Java classes, we adjust a configuration file, an environment variable, or even a command-line argument.

How Spring Achieves Externalization

1. Using Application Properties/YAML Files: Configurations placed in application.properties or application.yml

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=secret
```

2. Using Command-Line Arguments: we can override properties at runtime without touching the code.

```
java -jar myapp.jar --server.port=9090
```

3. Using Environment Variables: It is useful in cloud or containerized deployments. Example: setting SPRING_DATASOURCE_PASSWORD in Docker or Kubernetes.

How Spring read external configurations?

1. Using @Value annotation

The @Value annotation in Spring Boot is used to inject values into fields, method parameters, or constructors of Spring-managed beans.

```
@Value("${db.url}")
private String dbUrl;
```

Main disadvantage of @value is , It is not type-safe; wrong values can cause runtime errors.

To overcome this, we can use @ConfigurationProperties.

2. @ConfigurationProperties

Instead of injecting individual values like @Value, it binds a whole set of related properties into a strongly typed POJO.

```
@ConfigurationProperties(prefix = "app")
public class AppProperties {
    private String name;
    private int timeout;

    // getters and setters
}
```

application.properties:

```
app:
  name: MyApplication
  timeout: 5000
```

Here, name and timeout are mapped to the fields of AppProperties file.

373 Spring 122: Auditing in Spring JPA?

When we build applications that store data, we often need to keep track of who created a record, when it was created, and who last modified it. This is where *auditing* comes in. Instead of manually setting these fields everywhere in the code, Spring Data JPA provides a neat way to handle it automatically.

Spring Data JPA auditing hooks into the entity lifecycle. When we save or update an entity, Spring automatically fills in fields like:

- **CreatedDate**: when the entity was created
- **LastModifiedDate**: when the entity was updated
- **CreatedBy**: who created the entity
- **LastModifiedBy**: who last updated it

Step 1: Enable JPA Auditing

```
@SpringBootApplication
@EnableJpaAuditing // enables auditing support
public class AuditingApp {
    public static void main(String[] args) {
        SpringApplication.run(AuditingApp.class, args);
    }
}
```

Step 2: Create the Base Audit Class

```
@EntityListeners(AuditingEntityListener.class) // required for auditing to kick in
@Getter
@Setter
public abstract class Auditable {

    @CreatedBy
    @Column(updatable = false)
    private String createdBy;

    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createdDate;

    @LastModifiedBy
    private String lastModifiedBy;

    @LastModifiedDate
    private LocalDateTime lastModifiedDate;
}
```

Step 3: Extend Audit Fields in Entities

Any entity that extends this base class will automatically get audit fields populated.

```
@Entity
public class Product extends Auditable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
}
```

Step 4: Provide an AuditorAware Bean

Spring needs to know who the current user is. We do that by providing a bean of type AuditorAware<T>.

Indirect question: How to get current user logged in Spring?

```
@Configuration
public class AuditConfig {

    @Bean
    public AuditorAware<String> auditorAware() {
        // In real apps, fetch from SecurityContextHolder
        return () -> Optional.of("system_user");
    }
}
```

Step 5: Save

Whenever we save or update a Product, Spring will automatically fill in audit fields.

```
Product product = new Product();
product.setName("Laptop");
productRepository.save(product);

// createdBy = "system_user"
// createdDate = current timestamp
// lastModifiedBy = "system_user"
// lastModifiedDate = current timestamp
```

@EnableJpaAuditing: This annotation is placed on a Spring configuration class (e.g., the main application class). Its purpose is to activate the JPA auditing features provided by Spring Data.

When this annotation is present, Spring Data JPA automatically detects and processes entities that are configured for auditing.

@EntityListeners(AuditingEntityListener.class): This annotation is placed on an entity class (a class annotated with @Entity). It registers the AuditingEntityListener as an entity listener for that specific entity.

The **AuditingEntityListener** is a Spring Data JPA class that listens for JPA lifecycle events (like pre-persist and pre-update).

When these events occur, the AuditingEntityListener automatically populates the auditing fields within the entity, such as: @CreatedDate, @LastModifiedDate, @CreatedBy

Reference: [Database Auditing in Spring boot with spring security context and spring data JPA | by Mayank Yaduvanshi | Medium](#)

374 Spring 123: Explain application flow of the Spring boot or What will happen when we call run ()?

When we create a Spring Boot app, we typically start with something like:

```
@SpringBootApplication
public class MyApp {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}
```

Under the hood, a lot of things will happen. Let's walk through the flow step by step.

1. Run Method.

`SpringApplication.run()`

2. Environment Setup.

Spring Boot sets up the Environment, means it loads all the configurations like `application.properties`, system variables, and command line arguments.

3. ApplicationContext Creation.

Creates the `ApplicationContext`, it is the heart of Spring that manages all the beans.
`applicationContext` as the container that holds all the beans.

4. Bean Definitions

Now Spring boot scans all the beans. Spring detects them and registers it in the context.

`@SpringBootApplication` is the combination of:

`@Configuration`, `@EnableAutoConfiguration` and `@ComponentScan`

@Configuration - This annotation indicates that the class is a source of bean definitions for the Spring application context.

@EnableAutoConfiguration: This enables Spring Boot's auto-configuration mechanism. Spring Boot automatically configures the application based on the dependencies present in the class path.

@ComponentScan: This annotation instructs Spring to scan for components within a specified package (and its sub-packages). It automatically discovers and registers classes annotated with `@Component`, `@Controller`, `@Service`, `@Repository`

5. Auto-Configuration

It is one of best features in Spring boot. Based on classpath and environment, Boot decides what to configure.

For example:

- If `spring-boot-starter-web` is present, Boot auto-configures Tomcat, DispatcherServlet, etc.
- If `spring-boot-starter-data-jpa` is present, it configures Hibernate, DataSource, etc.

6. Context Refresh

After bean definitions are ready

- Spring calls context.refresh().
- This triggers actual bean instantiation, dependency injection, and initialization.

7. Starting Embedded Web Server (if web app):

If this is a web application:

- Boot starts Tomcat (or Jetty/Undertow).
- Binds it to the port from the config (server.port).
- Registers the DispatcherServlet.

8. Runners Execution.

Once the context is fully ready, Spring runs CommandLineRunner or ApplicationRunner

```
@Component
public class MyRunner implements CommandLineRunner {
    @Override
    public void run(String... args) {
        System.out.println("App started with args: " + Arrays.toString(args));
    }
}
```

9. Application Ready

Finally, ApplicationReadyEvent is published.

At this point:

- All beans are ready.
- The server is up.
- And the application is running.

375 Spring 124: What are different strategies of ID generation in Hibernate?

Every entity in Hibernate usually has a primary key. Hibernate needs to know how to generate that primary key when we insert a new record. This is where *identifier generation strategies* come in.

Choosing the right strategy affects not just performance, but also portability across databases, scalability, and even data integrity.

Hibernate supports a number of identifier generation strategies.

1. AUTO (Default Strategy)

JPA allows Hibernate (or the JPA provider) decide which strategy to use based on the underlying database. MySQL uses IDENTITY. Oracle uses SEQUENCE.

2. IDENTITY

The database auto-generates the primary key using an auto-increment column. The ID is assigned only after insert.

```
@Entity
class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
}
```

3. SEQUENCE

Uses a database sequence object to generate IDs. Hibernate fetches the next value from the sequence before the insert. Best for Oracle, PostgreSQL, or any DB that supports sequences.

```
@Entity
class User {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "user_seq")
    @SequenceGenerator(name = "user_seq", sequenceName = "user_sequence", allocationSize
= 1)
    private Long id;
}
```

4. UUID

Generates a globally unique ID (UUID) instead of a numeric value.

```
@Entity
class User {
    @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;
}
```

Point to remember: @GenericGenerator is deprecated from Hibernate 6.5, we will have to use alternatives like @UuidGenerator, @IdGeneratorType

Tricky question:

You have an entity with a @GeneratedValue(strategy = GenerationType.AUTO), but while inserting records, IDs are not generating sequentially. Why?

We are basically telling JPA: “Pick the generation strategy based on the underlying database.”

- On PostgreSQL / Oracle, it might use a sequence.
- On MySQL, it usually maps to AUTO_INCREMENT.
- On H2 / some others, it could choose a table generator.

And here is why we don't see sequential IDs:

- Sequences and identity columns often allocate IDs in blocks for performance. For example, Hibernate might reserve 50 IDs at a time (allocationSize=50 by default). If the app restarts, unused IDs are skipped, so gaps appear.
- In distributed systems, different instances can pre-allocate different ID ranges, leading to non-continuous values.
- AUTO doesn't guarantee *sequential* IDs So the answer: IDs are not sequential because GenerationType.AUTO relies on database-specific mechanisms (sequence, identity, table), and Hibernate optimizes them by allocating IDs in batches, which causes gaps.

If we really want sequential IDs, we have to explicitly use GenerationType.IDENTITY (on MySQL) or a @SequenceGenerator with allocationSize = 1 But it impacts the performance.

376 Spring 125: What is singleton?

A Singleton is a design pattern where only one instance of a class is created, and that single instance is shared across the application.

Think of it like having a single coffee machine in the office. No matter how many employees come to drink coffee, they all use the same machine. we don't build a new one every time.

In programming, this helps in:

- Saving memory (one object instead of many).
- Providing a single point of control.

Singleton in Plain Java Before Spring, we create a singleton in Java using private constructors and static methods.

```
class CoffeeMachine {
    private static CoffeeMachine instance;

    private CoffeeMachine() {} // private constructor

    public static CoffeeMachine getInstance() {
        if (instance == null) {
            instance = new CoffeeMachine();
        }
        return instance;
    }

    public void brew() {
        System.out.println("Brewing coffee...");
    }
}

public class Main {
```

```

public static void main(String[] args) {
    CoffeeMachine c1 = CoffeeMachine.getInstance();
    CoffeeMachine c2 = CoffeeMachine.getInstance();

    System.out.println(c1 == c2); // true (same object)
}
}

```

Here, c1 and c2 point to the same object.

How Spring Handles Singleton

In Spring, we don't have to write all that boilerplate code. By default, every bean we define in Spring is a singleton.

Example:

```

@Component
class CoffeeMachine {
    public void brew() {
        System.out.println("Brewing coffee in Spring...");
    }
}

```

Now, when we inject this bean in multiple places: Even though Employee1 and Employee2 have their own copies of coffeeMachine variable, Spring gives them the same instance of CoffeeMachine bean.

```

@Service
class Employee1 {
    @Autowired
    private CoffeeMachine coffeeMachine;

    public void drink() {
        coffeeMachine.brew();
    }
}

@Service
class Employee2 {
    @Autowired
    private CoffeeMachine coffeeMachine;

    public void drink() {
        coffeeMachine.brew();
    }
}

```

Additional Questions:

1. In Singleton, what problems may occur with serialization and reflection?

Serialization Problem

- When we serialize a singleton object and then deserialize it, Java creates a new instance of the class.
- This breaks the singleton guarantee because now we have two instances.
- Use `readResolve()` method to preserve singleton:

Reflection Problem

- Reflection can bypass the private constructor and create a new instance of the singleton.
- This also breaks the singleton guarantee.

Reference for reflection: [Java Reflection \(With Examples\)](#)

2. How to break Singleton design pattern?

Reference: [How to BREAK and FIX Singleton Design Pattern | Interview Question](#)

3. How singleton work in multithread environment?

By using synchronized keyword, we can create Singleton object in multithread environment.

Reference: [Singleton Design Pattern In Java With All Scenarios Examples](#)

4. How spring's singleton scope is different than GOF singleton?

GoF Singleton (Gang of Four Singleton Design Pattern):

- It is a creational pattern where we make sure only one instance of a class exists in the entire JVM.
- The class itself controls its lifecycle, usually with a private constructor and a `getInstance()` method.

```
public class GoFSingleton {
    private static GoFSingleton instance;

    private GoFSingleton() {}

    public static GoFSingleton getInstance() {
        if (instance == null) {
            instance = new GoFSingleton();
        }
        return instance;
    }
}
```

Spring Singleton (Bean Scope):

- "Singleton" in Spring means one bean instance per Spring container (`ApplicationContext`), not across the JVM.
- If we create multiple Spring contexts, each will have its own bean instance.

```
@Component
public class SpringSingletonBean {
```

```
ApplicationContext context1 = new AnnotationConfigApplicationContext(AppConfig.class);
ApplicationContext context2 = new AnnotationConfigApplicationContext(AppConfig.class);

SpringSingletonBean bean1 = context1.getBean(SpringSingletonBean.class);
SpringSingletonBean bean2 = context1.getBean(SpringSingletonBean.class);
SpringSingletonBean bean3 = context2.getBean(SpringSingletonBean.class);

System.out.println(bean1 == bean2); // true (same context)
System.out.println(bean1 == bean3); // false (different context)
```

377 Spring 126: How Cross Origin works in Spring?

When we open a website, the browser usually only allow us to talk to the same server it came from. That is called the **Same-Origin Policy**. But sometimes we want a page to talk to another server; for example, your frontend at <http://localhost:3000> needs to call a Spring Boot backend at <http://localhost:8080>. That is where **CORS (Cross-Origin Resource Sharing)** comes in. Here is how Spring handles it internally:

1. Browser Sends Origin

Every cross-origin request includes an Origin header. Example:

Origin: http://localhost:3000

Spring reads this header to check where the request came from.

2. Preflight Requests

If the browser wants to send something more complex than a GET or POST (say DELETE or custom headers), it sends a quick OPTIONS request first.

Example preflight request:

```
OPTIONS /api/data HTTP/1.1
Origin: http://localhost:3000
Access-Control-Request-Method: DELETE
```

3. Spring's CORS Configuration

Spring compares the request against the CORS rules we have defined:

With @CrossOrigin on a controller method:

```
@CrossOrigin(origins = "http://localhost:3000")
@GetMapping("/data")
public String getData() {
    return "Hello CORS!";
}
```

Or globally with WebMvcConfigurer:

```

@Override
public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**")
        .allowedOrigins("http://localhost:3000")
        .allowedMethods("GET", "POST", "DELETE");
}

```

4. Spring Adds Response Headers

If the origin is allowed, Spring attaches CORS headers to the response, for example:

- Access-Control-Allow-Origin: http://localhost:3000
- Access-Control-Allow-Methods: GET, POST, DELETE
- Access-Control-Allow-Headers: Content-Type

5. Browser checks It

The browser looks at these headers. If the headers match what it needs, it allows the frontend script to read the response. If not, it silently blocks it, even though the backend replied.

378 Spring 127: How @Retryable works in Spring?

Sometimes, we call an external service or a database, and it fails, maybe because of a network glitch, a timeout, or just bad luck. In such cases, instead of immediately throwing an error, we might want to try again a few times before giving up.

That is exactly where Spring's **@Retryable** comes in. It is part of **Spring Retry**, and it gives us a neat way to retry failed operations automatically without writing boilerplate loops.

How it Works

When we put **@Retryable** on a method:

1. Spring wraps that method in a proxy.
2. If the method throws an exception that we have marked for retry, Spring will call the method again.
3. It repeats this until either:
 - The method succeeds
 - The maximum retry attempts are exhausted
4. If retries are exhausted, we can handle the failure using **@Recover**.

Example:

Step 1: Add Dependency

```

<dependency>
  <groupId>org.springframework.retry</groupId>
  <artifactId>spring-retry</artifactId>
</dependency>
<dependency>

```

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

Step 2: Enable Retry

```
@SpringBootApplication
@EnableRetry
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

Step 3: Use @Retryable

```
@Service
public class PaymentService {

    private int counter = 1;

    @Retryable(
        value = RuntimeException.class, // Which exception to retry
        maxAttempts = 3,               // How many times
        backoff = @Backoff(delay = 2000) // Wait 2s between retries
    )
    public void makePayment() {
        System.out.println("Trying payment... attempt " + counter);
        counter++;
        throw new RuntimeException("Payment failed!");
    }

    @Recover
    public void recover(RuntimeException e) {
        System.out.println("All retries exhausted. Payment failed permanently: " +
e.getMessage());
    }
}
```

Key Points to Remember

- @Retryable works only on Spring-managed beans.
- It needs Spring AOP (that is why we added spring-boot-starter-aop).
- Use @Recover to define a fallback method after retries fail.
- We can control retries with:
 - maxAttempts means how many times
 - backoff means delay, multiplier (exponential backoff)
 - value to which exceptions trigger retry

Reference: [Implementing Method-Level Retry in Spring Boot | Medium](#)

Indirect Question:

1. One of your microservice is down then how do you handle?

Use Retry or circuit breaker

379 Spring 128: How proxyMode works in Spring and how does it help?

Problem: Injecting a Short-Lived Bean into a Long-Lived Bean

Imagine we have a **singleton bean** (default scope) and a **request-scoped bean**. We want to inject the request-scoped bean into the singleton:

Singleton bean:

```
@Service
public class SingletonService {

    @Autowired
    private RequestService requestService; // Problem!

    public void printRequestId() {
        System.out.println(requestService.getId());
    }
}
```

Request-scoped bean:

```
@Component
@RequestScope
public class RequestService {
    private final String id = UUID.randomUUID().toString();

    public String getId() {
        return id;
    }
}
```

What happens: Spring will throw an error or inject the same instance of the request bean into the singleton. That is wrong, because request-scoped beans are supposed to be new for each HTTP request.

The core issue: singleton lives longer than request, so Spring doesn't know which request instance to inject.

Solution: Use proxyMode

Definition: In Spring Framework, proxyMode = ScopedProxyMode.TARGET_CLASS is used in conjunction with the @Scope annotation to define how a scoped bean (like a prototype, request, or session-scoped bean) should be injected into other beans, especially singleton beans.

Spring provides a way to inject a proxy instead of the actual request bean. The proxy acts like a placeholder and resolves the actual bean at runtime based on the current request.

```
@Component
@RequestScope(proxyMode = ScopedProxyMode.TARGET_CLASS)
public class RequestService {
    private final String id = UUID.randomUUID().toString();

    public String getId() {
        return id;
    }
}
```

```
}
```

Reference: [Bean Scopes in Java Springboot. Non-related intro | by Jayamal Jayamaha | Medium](#)

Now, when the singleton calls `requestService.getId()`, Spring fetches the correct request-specific instance behind the scenes.

380 Spring 129: How to deploy Spring boot application using jar or war?

When we finish building a Spring Boot project, the next big step is **deployment**.

This means taking the application out of the IDE and running it in the real world means on a server, or even in the cloud.

Spring Boot makes this surprisingly simple, and it gives, two main ways to package and deploy the application:

- As an **executable JAR**
- As a **traditional WAR**

1. Deploying as a JAR (the modern way)

By default, Spring Boot applications are packaged as JAR files. Because Spring Boot already bundles an embedded web server (Tomcat, Jetty, or Undertow). That means the app carries its own server inside, like a backpack. We don't need to install Tomcat separately.

How to build a JAR

1. Open the pom.xml and make sure the packaging is set to jar:

```
<packaging>jar</packaging>
```

2. Package the app with Maven: `mvn clean package`
3. We will see something like this in the target folder: `myapp-0.0.1-SNAPSHOT.jar`
4. Run it just like any other Java program: `java -jar target/myapp-0.0.1-SNAPSHOT.jar`

2. Deploying as a WAR (the traditional way)

Before Spring Boot came along, Java web apps were typically packaged as **WAR files** and deployed into external servers like Tomcat, JBoss, or WebLogic.

This approach is still used in companies that maintain older infrastructures.

How to build a WAR

Change the packaging type in pom.xml: `<packaging>war</packaging>`

Update the main Spring Boot class to extend `SpringBootServletInitializer`:

```
@SpringBootApplication
public class MyAppApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return builder.sources(MyAppApplication.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(MyAppApplication.class, args);
    }
}
```

Build the WAR: `mvn clean package > myapp-0.0.1-SNAPSHOT.war`

Drop this WAR file into the external server's webapps/ directory (for Tomcat, it is usually /apache-tomcat/webapps/).

Additional Questions:

1. What is JAR and WAR?

A JAR file stands for Java Archive. It is a package file format used to aggregate many Java classes, associated metadata, and resources (like images or text) into one file for easy distribution. JAR files are primarily used for deploying standalone Java applications or libraries that other projects can reference.

A WAR file stands for Web Application Archive. It is a specialized JAR file specifically for web applications. It has a standardized structure for deploying to a Servlet Container (like Tomcat, Jetty) or Application Server (like WildFly, WebSphere).

2. When to choose JAR and WAR?

Use JAR for,

Standalone Applications: Any application that has its own main() method and is run from the command line (e.g., using `java -jar myapp.jar`). This is the standard for:

- Microservices
- Batch jobs (e.g., using Spring Batch)
- CLI tools and utilities
- Desktop applications (though less common now)

Modern Spring Boot Applications: This is a crucial point. Spring Boot uses an "embedded server" approach.

Our web application is packaged as a JAR.

This JAR contains an embedded Tomcat, Jetty, or Undertow server.

We can run it instantly without needing a separately installed Tomcat.

This is the default and preferred approach for most modern Spring Boot development due to its simplicity and portability.

Use WAR for

Traditional Web Application Deployment: When we need to deploy the application to an existing, external Servlet Container or Application Server.

- We have a dedicated Tomcat server where we need to deploy multiple applications.
- And Company's infrastructure team manages the server, and we just provide the .war file.

Reference: [Difference Between JAR and WAR. Java Archive\(Jar\) File vs Web... | by Abdul Kadar | Medium](#)

381 Spring 130: Explain different kind of paging techniques?

Imagine your database has 1 million rows. If your API return all of them in one shot, your client (and your server) cannot handle. How can you implement?

Solution:

This is where Pagination comes into picture, means fetching only a set of records at a time. Means “Reading records like a page where each page contains fixed number of records”.

Pagination Techniques

1. Offset-based Pagination (Page & Size)

This is the most common technique, here we need specify the page number and page size.

- Page number means ‘which page?’
- Page size means ‘how many records per page?’.

Example: GET /products?page=2&size=5

Here we are mentioning, “skip first 5 records, then give me next 5”.

In SQL, it handles with LIMIT and OFFSET.

```
SELECT * FROM products ORDER BY id LIMIT 5 OFFSET 5;
```

LIMIT “tells the database how many rows to return.” In this example 5 will be returned.

OFFSET “tells the database to skip the specified number of rows before applying the LIMIT”. In this example as we mentioned offset 5, it skips first 5 rows and return next 5 rows

```
SELECT * FROM products LIMIT 5 OFFSET 10;
```

In this example, it skips first 10 rows, then return next 5 rows (rows 11 to 15).

Remember like OFFSET is “start point,” LIMIT is “how many to fetch.”

In Spring, Pagination and sorting is handled by PagingAndSortingRepository.

```
Pageable pageable = PageRequest.of(1, 5); // page index starts at 0
Page<Product> products = productRepository.findAll(pageable);
```

The PageRequest.of(1, 2) means page index 1 (remember it starts at 0), with 5 items per page. Spring takes care of building the LIMIT and OFFSET query for us.

Reference: [Spring Boot Pagination and Sorting Example](#)

Offset pagination works for small datasets, imagine if we request for 10,000th page, then Database has to scan first 10000 records to skip them when slows down the process. For very large datasets we use Keyset/Cursor pagination.

2. Keyset Pagination (Also known as Cursor Pagination):

Keyset pagination retrieves records based on a specific key (usually a unique, indexed column like ID or timestamp). This makes it faster and more scalable, especially for large datasets.

Example, Let's say we are using “timestamp” as key:

In the first API call clients send without any key: GET /products?&size=10, so Backend has to first 10 records, SQL for this is

```
SELECT * FROM products ORDER BY timestamp ASC LIMIT 10;
```

The backend returns those 10 products and also includes the last product’s timestamp in the response (let's say 2025-01-01).

From next request, Client to has to send latest timestamp in the key GET /products?key=2025-01-01&size=10. SQL:

```
SELECT * FROM products WHERE timestamp > 2025-01-01 ORDER BY timestamp ASC LIMIT 10;
```

Which means client wants the 10 records after Jan 1st 2025. From there every request should contain the key.

Code Example:

```
public interface OrderRepository extends JpaRepository<Order, Long> {

    // First page (no cursor)
    @Query("SELECT o FROM Order o ORDER BY o.timeStamp ASC")
    List<Order> findFirstPage(Pageable pageable);

    // Next pages (cursor provided)
    @Query("SELECT o FROM Order o WHERE o. timeStamp > :cursor ORDER BY o.createdAt ASC")
    List<Order> findNextPage(@Param("cursor") Instant cursor, Pageable pageable);
}
```

Below service decides which query to call depending on whether a cursor is provided.

```

@Service
public class OrderService {

    @Autowired
    private OrderRepository orderRepository;

    public List<Order> getOrders(Integer size, Instant cursor) {
        Pageable pageable = PageRequest.of(0, size, Sort.by("createdAt").ascending());

        if (cursor == null) {
            return orderRepository.findFirstPage(pageable);
        } else {
            return orderRepository.findNextPage(cursor, pageable);
        }
    }
}

```

REST APIs

382 Rest 1: What is REST API and what are its uses?

REST (Representational State Transfer) API is a way for two systems (like a frontend app and a backend server) to talk to each other over HTTP using a set of rules. Instead of inventing new ways to exchange data, REST relies on standard HTTP methods like:

- **GET**: fetch data
- **POST**: create data
- **PUT/PATCH**: update data
- **DELETE**: remove data

The data is usually sent in formats like **JSON** or **XML** (JSON is the most common).

Example:

GET /users/1: Get details of user with ID 1

POST /users: Create a new user

Uses of REST API

1. **Communication between applications**: Web app or mobile app can fetch data from a backend server via REST APIs.
2. **Platform-independent**: Since it is based on HTTP, any system (Java, Python, React, Android, iOS) can use it.
3. **Scalability**: Each request is stateless (server doesn't store session info), making it easier to scale.
4. **Integration**: REST APIs allow connecting third-party services (payment gateways, maps, social media).

5. **Reusability:** Once an API is built, multiple clients (web, mobile, desktop apps) can use the same endpoints.

Point to Remember: You may be asked to write sample code for any API, so practice GET, POST, and PUT

Indirect Question:

1. What is stateless and stateful in REST?

In REST, stateless means every request from the client must contain all the info needed, and the server doesn't remember past interactions.

Stateful means the server stores session data about the client, so requests can rely on the server's memory of previous calls.

Stateless example:

- A GET /users/123 request always includes authentication (like a token in headers).
- The server doesn't remember the user logged in; every request carries the needed info.

Stateful example:

- we log in once, the server creates a session and stores it in memory.
- On the next GET /users/123, we just send a session ID, and the server uses its stored state to know the user who logged in.

2. How can you maintain a session in a stateless REST API

Use tokens instead

The common approach is JWT (JSON Web Tokens):

- Client logs in then server generates a token with user info and expiry.
- Client sends the token on every request (usually in the Authorization header).
- Server validates the token, without needing to store session state.

3. Why should you handle response timeout while calling any API and how to handle in Spring?

If we don't handle response timeouts when calling an API, our app can just wait there forever if the other service is slow or dead. That means bad user experience, and even system crashes under load.

In Spring, we handle this by setting **timeouts** in the HTTP client. For example, with RestTemplate or WebClient, we can configure connectTimeout and readTimeout. If the API doesn't respond in that time, the call fails fast, and we can retry, show an error, or switch to a fallback.

```
@Configuration
public class AppConfig {

    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder
            .setConnectTimeout(Duration.ofMillis(3000)) // Example: Set connection
timeout
            .setReadTimeout(Duration.ofMillis(3000)) // Example: Set read timeout
            .build();
    }
}
```

```

    }
}
```

4. Difference between Connection Timeout and Read Timeout?

Connection Timeout: How long the client will wait to establish a connection with the server. Example: if the server is down or unreachable, this timeout kicks in.

Read Timeout: How long the client will wait for data after the connection is established. Example: server accepted the connection but is too slow to send the response.

383 Rest 2: What is difference between PUT and PATCH?

PUT: It Replaces the entire resource with the new one. If we don't provide some fields, they can be overwritten with defaults or null.

```

@PutMapping("/{id}")
public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody User newUser)
{
    return userRepository.findById(id)
        .map(user -> {
            // replace all fields
            user.setName(newUser.getName());
            user.setAge(newUser.getAge());
            user.setCity(newUser.getCity());
            return ResponseEntity.ok(userRepository.save(user));
        })
        .orElseGet(() -> {
            // if not found, create new
            newUser.setId(id);
            return ResponseEntity.ok(userRepository.save(newUser));
        });
}
```

PATCH: It is used for Partially updates means only the fields we send are updated, the rest stay unchanged.

```

@PatchMapping("/{id}")
public ResponseEntity<User> patchUser(@PathVariable Long id, @RequestBody Map<String, Object> updates) {
    return userRepository.findById(id)
        .map(user -> {
            updates.forEach((key, value) -> {
                switch (key) {
                    case "name": user.setName((String) value); break;
                    case "age": user.setAge((Integer) value); break;
                    case "city": user.setCity((String) value); break;
                }
            });
            return ResponseEntity.ok(userRepository.save(user));
        })
}
```

```

        .orElse(ResponseEntity.notFound().build());
}

```

Tricky Question:

In both the methods we are using `userRepository.save(user)`, then how does PUT different from PATCH?

Let's see How save() works under the hood

- In JPA/Hibernate, save() doesn't directly do an SQL UPDATE of everything.
- It checks if the entity already exists (id is present).
 - If **new**, it runs INSERT.
 - If **existing**, it compares fields in the managed entity with what's changed and generates an UPDATE ... SET ... WHERE id=?.

Here the thing is it is not about what Hibernate does, it is about semantics of the API contract.

Technically, both may call save()

But from a REST design perspective:

- Use PUT if clients must send the entire object.
- Use PATCH if clients can send only the fields they want to change.

384 Rest 3: Can you tell what are the annotations you used while developing REST APIs?

`@RestController`: Marks the class as a REST controller (returns JSON/XML instead of views).

`@RequestMapping`: Defines the base URL for all methods in the controller.

`@GetMapping`, `@PostMapping`, `@PutMapping`, `@PatchMapping`, `@DeleteMapping`: HTTP-specific shortcuts for CRUD endpoints.

`@PathVariable`: Extracts values from the URI path.

```

@GetMapping("/users/{id}")
public User getUser(@PathVariable Long id) {
}

```

`@RequestParam`: Extracts query parameters from the URL.

```

@GetMapping("/users")
public List<User> findUsers(@RequestParam String city) {
}

```

`@RequestBody`: Maps the request JSON payload to a Java object.

`@ResponseEntity`: Builds custom responses with status codes and headers.

```
 ResponseEntity
    .status(HttpStatus.CREATED)
    .body(savedUser);
```

385 Rest 4: Difference between @RestController and @Controller?

@Controller

- It is a specialization of @Component; it marks class as a web request handler. It is used to declare common web controllers that can return HTTP responses
- By default, methods in a @Controller return views (like JSP, Thymeleaf templates) unless we add @ResponseBody.
- Typically used in web applications where we return HTML pages.

@RestController

- It is the combination @Controller + @ResponseBody. It is used to create controllers for REST APIs that can return JSON responses.
- Every method automatically returns JSON/XML instead of a view.
- Typically used in REST APIs.

Reference: [What is the difference between @Controller vs @RestController in Spring Boot?](#)

Tricky Questions:

1. What happens if we use both @Controller and @RestController on the same class?

- @RestController itself is a combination of @Controller + @ResponseBody.
- If we add @Controller along with @RestController, it doesn't break anything.
- But it is redundant, Means Spring will still treat it as a @RestController.

2. What happens if we don't use @ResponseBody with @Controller?

Without @ResponseBody, Spring will assume the method is returning a **view name** (HTML/JSP/Thymeleaf template).

```
@Controller
public class TestController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello World"; // interpreted as view name "Hello World"
    }
}
```

Spring will look for a template called Hello World.html instead of returning "Hello World" as text.

To return raw data (like JSON, String, etc.), we must use @ResponseBody or just switch to @RestController.

3: Can we use `@RestController` and still return a view (HTML)?

By default, no. `@RestController` forces `@ResponseBody` on all methods, so it always returns JSON/XML.

If we really want to return a view, we must explicitly use `ModelAndView`.

```
@RestController
public class PageController {
    @GetMapping("/page")
    public ModelAndView getPage() {
        return new ModelAndView("home"); // renders home.html
    }
}
```

4. What happens if a method in `@RestController` returns `void`?

- If the method is `void` and we don't write anything to the response, the client just gets an empty 200 OK response.
- Useful in cases like a `DELETE` endpoint where we don't want to return data.

386 Rest 5: Difference between `@RequestMapping` and `@GetMapping`?

`@RequestMapping`

- It is a general-purpose annotation to map HTTP requests to handler methods.
- It can handle **all HTTP methods** (GET, POST, PUT, DELETE, etc.) via the `method` attribute.

```
@RequestMapping(value = "/users", method = RequestMethod.GET)
public List<User> getUsers() {
}
```

`@GetMapping`

- It is the Shortcut (specialization) of `@RequestMapping(method = RequestMethod.GET)`.
- It is Cleaner, more readable when the method is only handling GET requests.

```
@GetMapping("/users")
public List<User> getUsers() {
}
```

Tricky Questions

1. Can `@RequestMapping` handle multiple HTTP methods in a single method?

Yes. Example:

```
@RequestMapping(value = "/users", method = {RequestMethod.GET, RequestMethod.POST})
public String handleUsers() {
    return "Handled GET or POST";
}
```

2. What happens if you use `@RequestMapping` without specifying a method?

It will match **all HTTP methods** (GET, POST, PUT, DELETE, etc.) for that URL.

3. Can you combine `@RequestMapping` at class level with `@GetMapping` at method level?

Yes, that is common. The class-level `@RequestMapping` acts as a **base path**

```
@RestController
@RequestMapping("/api")
public class UserController {

    @GetMapping("/users")
    public List<User> getAllUsers() {
        return List.of(new User(1L, "CodingLyf"));
    }
}
```

Final endpoint = /api/users.

387 Rest 6: Difference between `@PathVariable` and `@RequestParam`?

`@PathVariable`

It is used to extract values from the URL path itself.

Typically used when the value uniquely identifies a resource.

```
@GetMapping("/users/{id}")
public String getUserId(@PathVariable Long id) {
    return "User ID: " + id;
}
```

Request: /users/10 : Response: User ID: 10

`@RequestParam`

- Used to extract values from **query parameters** in the URL.
- Typically used for filtering, searching, sorting, or optional values.

```
@GetMapping("/users")
public String getUsersByCity(@RequestParam String city) {
    return "Users in city: " + city;
}
```

When to use which

Use `@PathVariable` when the value is part of the resource identifier.

/users/1/orders/99 userId = 1, orderId = 99

Use `@RequestParam` for optional or filtering parameters.

/users?city=Hyd&sort=asc

Tricky Questions:

1. What happens if you don't pass a @RequestParam that is required?

By default, it throws an exception (MissingServletRequestParameterException). To avoid this, we can make it optional:

```
@RequestParam(required = false) String city or
```

```
@RequestParam(defaultValue = "Hyderabad") String city
```

2. What happens if you don't pass a @PathVariable that is required?

If the value is required (default behavior) and it is missing from the request URL, we will get a 400 Bad Request error because Spring can't bind it.

If we make it optional (required = false) and it is missing, Spring will pass null to the method parameter.

```
// Optional PathVariable
@GetMapping("/{"/info", "/info/{id}"})
public String getUserInfo(@PathVariable(required = false) Integer id) {
    return id == null ? "No ID provided" : "User Info for ID: " + id;
}
```

388 Rest 7: What is Response Entity and how it will be used?

ResponseEntity is a Spring class that represents the whole HTTP response. It allows us to set not only the body of the response, but also the status code and headers.

In other words, instead of just returning data, we get full control over what the client will receive.

Why to use ResponseEntity?

- To customize HTTP status codes (200 OK, 201 CREATED, 404 NOT FOUND, etc.)
- To set custom headers (like authentication tokens, cache-control, etc.)
- To send data + metadata together

```
@GetMapping("/user/{id}")
public ResponseEntity<User> getUser(@PathVariable int id) {
    User user = userService.findById(id);
    if (user != null) {
        return ResponseEntity.ok(user); // 200 OK with user data
    } else {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build(); // 404
    }
}
```

It returns data with a status code

```
@PostMapping("/login")
public ResponseEntity<String> login(@RequestBody LoginRequest request) {
    String token = authService.authenticate(request);
```

```

    return ResponseEntity.ok()
        .header("Authorization", "Bearer " + token)
        .body("Login successful");
}

```

This login API response includes a header + body + status.

1. Can we return `ResponseEntity` without a body? If yes, what will happen?

Use `ResponseEntity.ok().build()`

2. How do you map exceptions to different HTTP status codes in a RESTful Spring boot application?

1. `@ResponseStatus` on custom exceptions

Attach the status code directly to an exception class.

```

@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}

```

Whenever this exception is thrown, Spring will return **404 Not Found**.

2. `@ExceptionHandler` inside a `@ControllerAdvice`

Centralized place to catch exceptions and return proper status codes.

```

@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String> handleBadRequest(IllegalArgumentException ex) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(ex.getMessage());
    }
}

```

389 Rest 8: Different types of HTTP status codes?

Role of HTTP codes:

HTTP status codes provide a standardized way to indicate the success or failure of an HTTP request. In REST APIs, they inform the client about the result of their request, such as success (2xx codes), client errors (4xx codes), or server errors (5xx codes). Proper use of these codes enhances API usability and debugging.

200 OK : Request succeeded and response contains the expected data

201 Created : A new resource was successfully created (commonly in POST APIs)

204 No Content : Request succeeded but no response body (e.g., DELETE success)

301 Moved Permanently : Resource permanently moved to a new URI

302 Found : Temporary redirect to another URI

304 Not Modified : Resource not changed, client can use cached version

400 Bad Request : Client sent an invalid request (missing params, invalid data)

401 Unauthorized : Authentication required or failed (invalid token/credentials)

403 Forbidden : Authenticated but not allowed to access the resource

404 Not Found : Requested resource doesn't exist

405 Method Not Allowed : HTTP method not supported for the resource

409 Conflict : Request conflicts with server state (e.g., duplicate entry)

415 Unsupported Media Type : Request format not supported (e.g., sending XML instead of JSON)

429 Too Many Requests : Rate limit exceeded

500 Internal Server Error : Generic server-side failure

502 Bad Gateway : Server acting as a proxy/gateway got invalid response

503 Service Unavailable : Server temporarily overloaded or down for maintenance

504 Gateway Timeout : Server acting as a proxy/gateway didn't get a timely response

390 Rest 9: How Spring send JSON by default / How Spring converts Java object to JSON while returning response from API?

Here is what happens behind the scenes:

Spring MVC + @RestController

When we return a Java object from a controller method (inside a @RestController or with @ResponseBody), Spring doesn't just return the object directly. It delegates to a **HttpMessageConverter** to serialize it into the desired format (usually JSON).

Default JSON Conversion

- Spring Boot includes **Jackson** (com.fasterxml.jackson) in its starter dependencies (spring-boot-starter-web).
- Jackson automatically converts the Java object (POJO) into JSON if the request's Accept header includes application/json.

```

@RestController
public class UserController {
    @GetMapping("/user")
    public User getUser() {
        return new User("CodingLyf", 1);
    }
}

class User {
    private String name;
    private int age;

    // getters & setters
}

```

Response JSON:

```
{
    "name": "CodingLyf",
    "age": 1
}
```

391 Rest 10: Different types of headers?

1. Content-Type

Defines the format of the request or response body (JSON, XML, text).

Example: Content-Type: application/json

When a client sends data to a server, the Content-Type header informs the server about the format of the data in the request body.

When a server sends data back to a client, the Content-Type header informs the client about the format of the data in the response body

2. Accept

The Accept header is a request header used by clients to specify which media types they are willing to accept as a response from the server.

Example: Accept: application/json

3. Authorization

It carries credentials like tokens or API keys for authentication.

Example: Authorization: Bearer eyJhbGciOiJIUzI1NilsInR5cCI6...

4. Cache-Control

It is used for caching behavior like no-store, no-cache, or max-age.

Example: Cache-Control: no-cache

392 Rest 11: What is content negotiation?

Content Negotiation is how Spring (or any web framework) decides what format (JSON, XML, HTML, etc.) to send back to the client based on the request.

There are three main ways:

1. Header-based (Accept Header)

Client sends Accept: application/json , Server responds with JSON.

GET /products Accept: application/xml

Response: XML format.

2. Parameter-based (URL query parameter)

Format is specified in URL.

GET /products?format=json

3. Path extension (deprecated in Spring)

o File-like extension in URL.

GET /products.json

Indirect Questions:

1. Can a controller return String instead of JSON?

Yes.

- In a @RestController, we can return a string as plain text
- In a @Controller, a String is usually treated as a view name (we need to annotate the method with @ResponseBody, then we can send as plain text).

2. What if we return a plain string without @ResponseBody?

A String is usually treated as a view name

3. Suppose you need to send a Java object over a network in a distributed application. What challenges might arise, and how would you handle them?

Challenges

1. Serialization: To send an object, it must be converted into bytes. If the class doesn't implement Serializable, it can't be sent directly.
2. Versioning issues: If sender and receiver have different versions of the class (e.g., extra fields), deserialization can fail.
3. Performance overhead: Default Java serialization is slow and creates large payloads.
4. Security risks: Deserialization can be exploited if untrusted data is processed.

How to handle

- Use Serializable or Externalizable for converting objects to bytes.

- Define serialVersionUID to manage version compatibility.
- For performance and portability, prefer protocols like JSON, XML, or Protobuf instead of Java's built-in serialization.
- Validate inputs and avoid deserializing from untrusted sources to reduce security risks.

4. If you want to return both HTML views and JSON responses from a single controller class, how can you achieve this?

@Controller can return HTML views (via Thymeleaf, JSP, etc.) when you return a String (view name).

It can also return JSON if you annotate specific methods with @ResponseBody.

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

@Controller
@RequestMapping("/users")
public class UserController {

    // Returns HTML view
    @GetMapping("/page")
    public String userPage() {
        return "userView"; // mapped to a template like userView.html
    }

    // Returns JSON
    @GetMapping("/json")
    @ResponseBody
    public User getUserJson() {
        return new User(1, "CodingLyf");
    }
}
```

393 Rest 12: How can Spring send XML if default is JSON?

By default, Spring Boot uses Jackson to convert Java objects to JSON. But it can also produce XML if we include the right dependency.

Steps:

1. Add Jackson XML dependency: If we use Maven, add this to pom.xml:

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

2. Controller Example

```
@RestController
@RequestMapping("/api")
public class EmployeeController {
```

```

    @GetMapping(value = "/employee", produces = {MediaType.APPLICATION_XML_VALUE,
    MediaType.APPLICATION_JSON_VALUE})
    public Employee getEmployee() {
        return new Employee(1, "CodingLyf", "Software Engineer");
    }
}

```

If client sends Accept: application/json , returns JSON.

If client sends Accept: application/xml , returns XML.

394 Rest 13: What is Idempotency?

Idempotency means **making the same request multiple times gives the same result without causing side effects.**

In HTTP:

- **GET** is idempotent, calling it multiple times returns the same data.
- **DELETE** is idempotent, deleting the same resource multiple times still results in "deleted".
- **PUT** is idempotent, updating a resource with the same payload gives the same state.

Example:

- DELETE /users/123 : first call deletes, next calls return "user not found", but the system state doesn't change further.

395 Rest 14: Why Patch is not idempotent?

PATCH is not guaranteed idempotent.

Reason: PATCH applies a partial update. If the patch operation is additive (like "increment balance by 100"), sending it multiple times changes the state each time.

But if the patch operation is a direct replacement (like "set name to 'CodingLyf'"), then it behaves idempotent.

PATCH may or may not be idempotent depending on how you design the patch semantics.

By HTTP spec, PATCH is not required to be idempotent (unlike PUT).

396 Rest 15: Can we make POST as idempotent?

Normally, POST is not idempotent because each call creates a new resource (like adding a new record).

But yes, we can design a POST API to behave idempotently using Idempotency Keys.

How it works

- While invoking the API, Client generates a unique key (say Idempotency-Key: abc123).
- Server stores this key with the request and its response.

- If the client retries the same POST with the same key, the server returns the stored response instead of creating a new resource again.

Real-world use case:

- Payment APIs (Stripe, PayPal, Razorpay, etc.)
 - When we POST a payment request, client include an idempotency key.
 - If the client retries due to timeout, the server won't charge the customer twice.

So, POST can be made idempotent through server-side logic + idempotency keys.

397 Rest 16: How do REST APIs handle versioning, and why is it important?

Once we release an API, people use it. If we change the endpoints, request/response formats then clients can break mean they may get errors eventually apps fail.

To avoid failures to the existing clients, we use something called API Versioning.

REST API versioning is all about how we manage changes to the API without breaking existing clients

There are various approaches to versioning a RESTful API, including:

- URL Versioning: Adding a version number to the API endpoint (e.g., /api/v1/resource).
- Query Parameter Versioning: Specifying the version as a query parameter (e.g., /api/resource?version=1).
- Header Versioning: Sending the version information in a custom header (e.g., X-API-Version: 1).

398 Rest 17: How do you validate the Request from client in Spring boot?

Spring Boot provides a clean way to handle validation using **annotations**, **binding**, and **exception handling**.

1. Using Validation Annotations

Spring Boot supports the Jakarta Validation API with annotations like:

- `@NotNull` – ensures the value is not null
- `@Size(min=, max=)` – validates the length of strings or collections
- `@Email` – ensures the value is a valid email address
- `@Min / @Max` – validates numeric ranges

These annotations are added directly to the request DTOs (Data Transfer Objects).

2. Enabling Validation in Controllers

To trigger validation automatically when a client sends a request, we need to use the `@Valid` annotation in the controller method

```
@PostMapping("/users")
public ResponseEntity<String> createUser(@Valid @RequestBody UserRequest request) {
    return ResponseEntity.ok("User created successfully");
}
```

@Valid tells Spring to check the DTO annotations.

If the request is invalid, Spring throws a MethodArgumentNotValidException.

3. Handling Validation Errors

We can handle errors gracefully using @ControllerAdvice and a global exception handler:

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<String> handleValidationErrors(MethodArgumentNotValidException ex) {
        String errorMessage = ex.getBindingResult()
            .getFieldErrors()
            .stream()
            .map(err -> err.getField() + ": " +
err.getDefaultMessage())
            .collect(Collectors.joining(", "));
        return ResponseEntity.badRequest().body(errorMessage);
    }
}
```

This way, we can send a clear message about what went wrong instead of a generic server error.

399 Rest 18: How to create custom validators in spring?

Creating custom validators in Spring involves defining a custom annotation and implementing the validation logic in a separate class.

Let's say we are creating custom validator to check Strong password. Password should contain at least 8 chars, 1 digit, 1 uppercase.

Steps to Create a Custom Validator:

1. Add the Validation Dependency.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

2. Create a Custom Constraint Annotation.

Define an annotation that will be used to mark fields or classes for validation. This annotation needs to specify the validator class and other metadata.

```

import jakarta.validation.Constraint;
import jakarta.validation.Payload;
import java.lang.annotation.*;

@Documented
@Constraint(validatedBy = StrongPasswordValidator.class)
@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface StrongPassword {
    String message() default "Password is too weak"; //Default error message
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}

```

- `@Constraint` declares the class which will implement our interface and define its behaviour, `StrongPasswordValidator` is the class which will implement `StrongPassword` interface
- `@Target` defines the program element types to which our custom annotation will apply,
- `@Retention` defines when our annotation will be available.

3. Implement the Validator Class.

Create a class that implements the `ConstraintValidator` interface. This class contains the actual validation logic.

```

import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;

public class StrongPasswordValidator implements ConstraintValidator<StrongPassword,
String> { // StrongPassword is the annotation, String is the type to validate

    @Override
    public boolean isValid(String password, ConstraintValidatorContext context) {
        if (password == null) return false;
        // simple example: at least 8 chars, 1 digit, 1 uppercase
        return password.length() >= 8 &&
            password.matches(".*[A-Z].*") &&
            password.matches(".*\\d.*");
    }
}

```

4. Apply the Custom Annotation.

Annotate the fields or parameters in the model or DTOs where you want to apply the custom validation.

```

public class UserRequest {
    @StrongPassword
    private String password;

    // getters and setters
}

```

5. Trigger Validation.

Ensure that validation is triggered, typically by using @Valid or @Validated annotations on method parameters in the controllers or services.

```
import org.springframework.web.bind.annotation.*;
import jakarta.validation.Valid;

@RestController
@RequestMapping("/users")
public class UserController {

    @PostMapping
    public String createUser(@Valid @RequestBody UserRequest request) {
        return "User created with password: " + request.getPassword();
    }
}
```

That is it. Now if the password doesn't meet the rule, Spring Boot will throw a MethodArgumentNotValidException with the custom message.

Reference: [Spring Boot Custom Validation. Creating custom validation in a Spring... | by Bereket Berhe | Medium](#)

[Creating Spring Boot Custom Validators | by Catherine Edelveis | Medium](#)

400 Rest 19: How do you implement caching in REST APIs?

Implementing caching in REST APIs involves various strategies across different layers to improve performance and reduce server load.

1. Client-Side Caching (Browser Caching):

HTTP Headers:

- We can achieve using HTTP headers, Cache-Control, Expires, ETag, and Last-Modified headers in the API responses.
- Cache-Control: Specifies caching policies (e.g., public, private, max-age, no-cache).
 - Cache-Control: max-age=3600, public: Means the response can be cached by anyone for 1 hour.
- Expires: Provides a date/time after which the response is considered stale.
 - Expires: Wed, 23 Aug 2025 12:00:00 GMT
- ETag: A unique identifier for the resource version, allowing clients to send If-None-Match to check for updates.
- Last-Modified: The last modification date of the resource, used with If-Modified-Since.

2. Server-side Caching

- Store frequently accessed responses in Redis, Memcached, or in-memory caches.

- Example: Cache all product details for 10 minutes in Redis, so repeated API calls don't hit the database.

401 Rest 20: How can you handle rate limiting in REST API?

Rate limiting is the process of restricting the number of requests a client can make to an API within a given time frame. It helps prevent abuse, protects server resources, and ensures fair usage. Rate limiting can be implemented by tracking the number of requests per client and enforcing limits using techniques like token bucket or fixed window algorithms.

Refer Question No 435 , Spring Scenario 23 for the implementation

402 Rest 21: Rate limiting vs API throttling?

Both these are used to protect the misuse or overload of the APIs

1. Rate Limiting

Rate limiting is about setting a maximum number of requests a client can make within a fixed time window.

If the limit is 100 requests per minute, the 101st request is immediately rejected. If the limit is crossed, the API usually returns 429 Too Many Requests.

2. Throttling

Throttling is about controlling the speed at which requests are processed, rather than blocking them.

If the system allows 5 requests per second and we send 20 at once, only 5 are processed immediately. The rest are queued, delayed, or dropped based on policy.

403 Rest 22: What are different types of authentications used in REST APIs?

Basic Authentication: The client includes the username and password in the request headers. This method is simple but not recommended for sensitive data as it transmits credentials in plaintext.

Token-based Authentication: The server generates and returns a token (e.g., JSON Web Token or JWT) upon successful login. The client includes this token in subsequent requests to access protected resources.

OAuth: A protocol that allows users to authorize third-party applications to access their resources without sharing credentials. It involves obtaining an access token from an authorization server and using it to make authenticated requests.

404 Rest 23: Difference between URI and URL?

A URI (Uniform Resource Identifier) is a general term that identifies a resource. It can identify a resource by its name, its location, or both. Think of a URI as a way to uniquely identify something.

A URL (Uniform Resource Locator) is a specific type of URI that identifies a resource by its location and also describes the means of accessing it. In essence, a URL tells where a resource is and how to get it.

Key Differences:

- **Scope:** All URLs are URIs, but not all URIs are URLs. A URI can be a URN (Uniform Resource Name) which identifies a resource by its name without specifying its location (e.g., urn:isbn:0451450523 identifies a book by its ISBN).
- **Purpose:** URIs are for identification, while URLs are for location and access.
- **Components:** URLs always include a protocol (like http://, https://, ftp://), a domain name, and potentially a path and file name, all of which are essential for locating and accessing the resource. URIs can be simpler, focusing solely on identification.

405 Rest 24: How do you implement pagination in REST API?

In a REST API, pagination is often used to limit the amount of data returned in a single request. There are different approaches to handling pagination.

One common approach is to use query parameters, such as "page" and "limit", to specify the desired page number and the number of items per page.

The API can then use this information to retrieve and return the appropriate data subset.

Pagination is important for improving performance and ensuring that large data sets are processed efficiently.

406 Rest 25: What is the difference between synchronous and asynchronous communication in RESTful web services?

When building RESTful APIs, how the client and server communicate matters. Communication can either be synchronous or asynchronous.

Synchronous Communication

In synchronous communication, the client sends a request and **waits for the server** to respond before doing anything else. The client is essentially “blocked” until the response arrives.

For example, when a client requests user information with GET /users/123, it waits until the server returns the user data. This approach works well for operations where the result is needed immediately, such as retrieving, updating, or deleting a resource.

Asynchronous Communication

In asynchronous communication, the client sends a request but **does not wait for the server** to respond. Instead, the client continues with other tasks. Later, it can check back for the response or receive a callback when the server has completed processing.

For instance, if a client requests generating a large report with POST /generate-report, the server might immediately return a job ID. The client can continue with other work and later query the status of the report or get notified when it is ready.

407 Rest 26: What are the best practices while designing the REST API?

Designing a RESTful API involves following several best practices. These practices ensure a reliable, scalable, and easily understandable RESTful API design:

- Use meaningful and consistent resource names for the API endpoints.
- Use HTTP verbs (GET, POST, PUT, DELETE) correctly to perform the corresponding actions on the resources.
- Ensure that the API is stateless and authentication is handled using tokens or OAuth.
- Use descriptive error messages and correct HTTP status codes for responses.
- Versioning the API can help manage changes over time.
- Lastly, document the API comprehensively, including endpoint details, request and response formats, and examples.

408 Rest 27: How do you document the APIs and their details?

When building RESTful APIs, proper documentation is crucial. It helps developers understand how to use the API, what endpoints are available, what parameters are required, and what responses to expect. One of the most popular tools for this purpose is **Swagger**, now known as **OpenAPI**.

Swagger allows developers to describe APIs in a structured JSON or YAML format. This description includes all the endpoints, request parameters, response formats, authentication methods, and even error codes. Once the API is described, Swagger can automatically generate human-readable documentation that is easy to navigate.

409 Rest 28: SOAP VS REST?

When designing web services, two popular approaches are SOAP (Simple Object Access Protocol) and REST (Representational State Transfer). While both allow applications communicate over the web, they differ in philosophy, design, and use cases.

1. SOAP (Simple Object Access Protocol)

- **Protocol-based:** SOAP is a strict protocol with specific rules and standards.
- **Message Format:** Uses **XML** for both requests and responses.
- **Features:** Built-in error handling, security (WS-Security), transactions, and ACID compliance.
- **Transport:** Usually HTTP, but can also use SMTP, TCP, or JMS.
- **Use Case:** Enterprise applications requiring high security, reliable messaging, and formal contracts (like banking, payment gateways).

2. REST (Representational State Transfer)

- **Architectural style:** REST is not a protocol; it is a set of principles for designing APIs.
- **Message Format:** Uses **JSON** (most common) or XML.

- **Features:** Lightweight, stateless, supports CRUD operations via standard HTTP methods (GET, POST, PUT, DELETE).
- **Transport:** Only HTTP/HTTPS.
- **Use Case:** Web and mobile applications requiring fast, scalable, and flexible APIs (like social media apps, e-commerce platforms)

Additional Question:

Why do financial services prefer SOAP over rest?

Strong Contracts: SOAP uses WSDL (Web Services Description Language) which is like a strict contract between client and server. In finance, this ensures no ambiguity in request/response formats.

Built-in Security: SOAP supports WS-Security (signing, encryption, authentication) out of the box. REST relies on external mechanisms like OAuth, JWT, or HTTPS.

Transaction Support: SOAP has standards for things like ACID transactions (WS-AtomicTransaction), which is critical for money transfers where operations must succeed or fail together.

410 Rest 29: How do you invoke the other APIs or Inter-service communication in a microservice in Spring?

In modern applications, it is common for any service to call other APIs, whether internal microservices or third-party services. Spring provides simple, flexible ways to make these HTTP requests and handle responses.

1. Using RestTemplate

RestTemplate is the classic way to call REST APIs in Spring. It is synchronous, meaning the call will block until a response is received.

```
@Autowired
private RestTemplate restTemplate;

public String getUserData() {
    String url = "https://api.example.com/users/123";
    return restTemplate.getForObject(url, String.class);
}
```

2. Using WebClient

WebClient is part of Spring WebFlux and supports reactive, non-blocking calls. It is the preferred approach for modern, scalable applications.

```
@Autowired
private WebClient webClient;

public Mono<String> getUserData() {
    return webClient.get()
        .uri("https://api.example.com/users/123")
```

```

        .retrieve()
        .bodyToMono(String.class);
}

```

Mono<String> represents a single asynchronous value.

retrieve() handles the HTTP response.

Point to remember:

Spring has announced RestTemplate is in maintenance mode. That means:

- It will still work in current and future Spring versions but No new major features are planned.
- Spring recommends using WebClient (from Spring WebFlux) for new projects because it supports non-blocking, reactive calls and is more flexible for modern applications.

411 Rest 30: How do you implement search functionality?

1. Using Query Parameters

The most common way. We pass search criteria as query parameters in the URL.

Example: Search users by name and city:

```

@GetMapping("/users")
public List<User> searchUsers(
    @RequestParam(required = false) String name,
    @RequestParam(required = false) String city) {
    return userService.searchUsers(name, city);
}

```

Request looks like: GET /users?name=CodingLyf&city=Hyderabad

2. Using Request Body (for complex searches)

For more advanced filters (ranges, multiple fields, or nested conditions), we can send a POST with a search payload.

```

@PostMapping("/users/search")
public List<User> searchUsers(@RequestBody UserSearchCriteria criteria) {
    return userService.searchUsers(criteria);
}

```

Request payload:

```
{
    "name": "CodingLyf",
    "city": "Hyderabad",
    "ageMin": 25,
    "ageMax": 35
}
```

Spring Scenario Based Questions

412 Spring Scenario 1: Get by User ID or Email.

You are designing an API to fetch the user details. How would you like to fetch, will you use user-id or user-email and why?

Explanation: User ID is preferred.

Why?

- Every user has a unique ID, usually an auto-generated integer or UUID. There's no risk of collision.
- Database queries by primary key (like userId) are extremely fast.
- IDs don't reveal personal information, unlike emails.

413 Spring Scenario 2: Debug the slow API.

Imagine there is an existing API but it takes too long to respond. How you debug it?

1. Check if it is really slow

First, confirm the API is slow. Use **Postman** or **cURL** and measure the response time. If it is only slow in the browser, it might be a frontend issue. If Postman also shows slowness, then it is the backend.

2. Check Latency.

- Network latency (time taken to reach the server).

We can use “curl -w” command to check the network latency

- Server processing time (time spent inside the backend).

We place loggers to check the server processing time.

```
@GetMapping("/users")
public List<User> getUsers() {
    long start = System.currentTimeMillis();

    List<User> users = userService.getUsers();

    long end = System.currentTimeMillis();
    log.info("Server processing took {} ms", (end - start));

    return users;
}
```

- Database or external calls (time spent waiting for other services).

We place loggers at repository to check the DB processing time.

```
long dbStart = System.currentTimeMillis();
List<User> users = userRepository.findAll();
long dbEnd = System.currentTimeMillis();

log.info("Database call took {} ms", (dbEnd - dbStart));
```

We can use commons-lang3 **StopWatch** instead of loggers

4. Look at server-side profiling

- We can use an APM (Application Performance Management) tool (like New Relic, Spring Boot Actuator).
- These tools show a detailed breakdown: how much time was spent in controller, service, repository, etc.

5. If server processing is slow, we might need to optimize the code.

6. If DB processing is slow, we need to optimize the query.

This is how we debug the API.

414 Spring Scenario 3: Performance optimization.

You have found an API is very slow and you need to optimize it, what are the steps you are going to take?

1. Check loops: Inefficient loops can slow down the process for example, looping over thousands of records in Java when the database could filter them.

```
//Bad: filtering in Java
List<User> users = userRepository.findAll();
List<User> active = new ArrayList<>();
for (User u : users) {
    if (u.isActive()) {
        active.add(u);
    }
}
```

Better approach: push filtering to DB

2. Creating objects inside a loop slows things down

```
for (int i = 0; i < 100000; i++) {
    String s = new String("Hello"); // unnecessary new object
}
```

3. Use Asynchronous Processing: Use asynchronous processing for long-running tasks to improve responsiveness

4. Caching: Implement caching to reduce database load and improve response times

5. Pagination: For large datasets implement the pagination (Refer Question No.)

6. Optimize the Queries

```
-- Bad
SELECT * FROM users;  
  
-- Better (only fetch what you need)
SELECT id, name FROM users;
```

7. Have proper indexes: If we query on a column without an index, DB scans the whole table (Refer Question No. 372, [Spring 120](#))

8. Address the N+1 Problem: Fetching the data inside a loop can make N number of queries. Try to fix using JOIN Fetch (Refer Question No 303, 304, Spring 51,52).

9. Connection Pooling: Use connection pooling to manage database connections efficiently.

415 Spring Scenario 4: DB Migrations

You need features to an existing Spring Boot application, which involves changing the database schema. How do you ensure smooth database migrations?

Steps:

1. To handle database migrations, we can use migrations tools like Flyway or Liquibase.
2. Write SQL changes in separate files and place them in `src/main/resources/db/migration`:
 - V1__create_users_table.sql
 - V2__add_email_to_users.sql

The numbering (V1, V2, V3) ensures order, and the tool keeps track of what's already been applied.

3. Spring Boot can run Flyway or Liquibase migrations at startup. That means whenever the app boots, it checks which migrations are pending and applies them. This keeps development and test environments in sync automatically.
4. Always run them in a staging database first. This helps catch performance issues (like adding an index on a huge table) or mistakes (like dropping a column that is still used).
5. It's always good to have a rollback mechanism in place. Tools like Liquibase are useful to define rollback scripts. Flyway doesn't encourage rollbacks but suggests creating a new "fix" migration if needed.

416 Spring Scenario 5: File upload and download

You are designing an APIs for upload and download the files; how would you achieve this?

Explanation:

For file upload, we need to use `MultipartFile`. It represents the file sent in a multipart/form-data request.

- Client sends file via HTTP POST request.
- Server receives it as `MultipartFile`.
- Server processes and stores the file (in DB, cloud, or local storage).
- Return a success response.

```
@PostMapping("/upload")
public ResponseEntity<String> uploadFile(@RequestParam("file") MultipartFile file) {
    try {
        // Save to disk (demo purpose)
        Path path = Paths.get("uploads/" + file.getOriginalFilename());
        Files.write(path, file.getBytes());

        return ResponseEntity.ok("File uploaded successfully: " +
file.getOriginalFilename());
    } catch (IOException e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                           .body("Could not upload file");
    }
}
```

For File downloads, two headers matter:

- Content-Disposition: tells the browser to download instead of preview.
- Content-Type : defines file type (application/pdf, application/vnd.ms-excel, etc.).

```
@GetMapping("/download/{fileName}")
public ResponseEntity<Resource> downloadFile(@PathVariable String fileName) {
    try {
        Path path = Paths.get("uploads/" + fileName);
        Resource resource = new UrlResource(path.toUri());

        if (!resource.exists()) {
            return ResponseEntity.notFound().build();
        }

        return ResponseEntity.ok()
                           .contentType(MediaType.APPLICATION_OCTET_STREAM)
                           .header(HttpHeaders.CONTENT_DISPOSITION,
                                  "attachment; filename=\"" + resource.getFilename() + "\"")
                           .body(resource);

    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}
```

Sample URL downloads the file: <http://localhost:8080/download/report.pdf>

Additional Questions.

1. What is the maximum file size that spring accepts?

By default, Spring Boot accepts file uploads up to **1MB** (per file) and **10MB** (per request).

2. How do you increase the acceptable file size ?

You can increase this by setting properties like `spring.servlet.multipart.max-file-size` and `spring.servlet.multipart.max-request-size` in `application.properties`.

```
spring.servlet.multipart.max-file-size=10MB  
spring.servlet.multipart.max-request-size=10MB
```

3. What is the maximum file size that Spring can accept?

Spring itself doesn't impose any maximum size, but very large uploads are limited by memory, disk, and timeout constraints.

417 Spring Scenario 6: Asynchronous Processing

Suppose you need to a large number of data and save data in DB. It is very long running task. How would you handle it?

Use a background worker

This is where tools like **Spring Batch** or **@Async with ExecutorService** comes into picture.

- Spring Batch is built exactly for this kind of job like chunked reading, processing, and writing to DB.
- If we don't want the overhead of Spring Batch, we can write our own async task using **@Async** (Refer Question no.351 Spring 99) or even put it on a **queue system** (like RabbitMQ, Kafka).

418 Spring Scenario 7: Handling Configuration

Suppose you need to develop an APP for different environments where each environment has its own configurations like database and endpoints. How would you configure it?

Explanation:

When the app runs in different environments (development, testing, production), each one has its own settings: different databases, different API endpoints.

Spring Boot provides a flexible way to manage configuration properties using **application.properties** or **application.yml** files.

1. Create Environment-Specific Property Files

For each environment, you create a dedicated file inside src/main/resources.

application-dev.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/devdb
spring.datasource.username=devuser
spring.datasource.password=devpass
```

application-prod.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/proddb
spring.datasource.username=produser
spring.datasource.password=prodpass
```

2. Activate the Right Profile

We tell Spring Boot which environment to use by setting the **active profile**.

This can be done in different ways:

Option A: Inside application.properties, use this spring.profiles.active=dev

Option B: java -jar app.jar --spring.profiles.active=prod (for prod)

3. Inject Properties.

Use **@ConfigurationProperties**, this annotation helps us to map the entire content of the properties file to a POJO (Plain Old Java Object). A property file can be either application.properties or application.yml.

```
@Configuration
@ConfigurationProperties(prefix = "spring.datasource")
public class DataSourceConfig {
    private String url;
    private String username;
    private String password;

    // Getters and Setters
}
```

Spring will automatically map the right values depending on the active profile.

4. @Profile for Environment-Specific Beans

Sometimes, you want different beans for different environments.

Example: a dummy email service in dev, and a real one in prod.

```
public interface EmailService {
    void send(String to, String msg);
```

```

}

@Profile("dev")
@Service
public class DummyEmailService implements EmailService {
    public void send(String to, String msg) {
        System.out.println("DEV mode: Email not really sent to " + to);
    }
}

@Profile("prod")
@Service
public class RealEmailService implements EmailService {
    public void send(String to, String msg) {
        // actual implementation
    }
}

```

419 Spring Scenario 8: Apply configuration without reboot

You are running a Spring Boot application in production. One day, your manager comes and says: "We need to change the log level to DEBUG for one application/service to investigate an issue, but you should not restart the service because it is handling live traffic."

In other words, you must **apply configuration changes without rebooting** the application.

Explanation:

Normally, Spring Boot reads its configuration from application.properties or application.yml files at startup. So, if we change them, we need to restart the app

To solve this, Spring provides ways to dynamically refresh configurations without restarting.

To do this we need to follow these things:

1. Store configurations centrally in a separate Git repo.
2. Set up a Spring Cloud Config Server
3. Connect your main applications to Config Server
4. Enable actuator > refresh API

1. Store configurations centrally in a separate Git repo.

Create a Git repository (say config-repo). Inside it, place environment-specific property files:

```

application-dev.properties
application-test.properties
application-prod.properties

```

2. Set up a Spring Cloud Config Server

Create a new Spring Boot app with `@EnableConfigServer` and add

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Point it to the Git repo (config-repo) in application.properties:

```
spring.cloud.config.server.git.uri=https://github.com/org/config-repo
```

```
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer
public class ConfigServer {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServer.class, args);
    }
}
```

`@EnableConfigServer`. This annotation tells Spring Boot to enable the Config Server functionality for the application.

3. Connect your main applications to Config Server

- Add `spring-cloud-starter-config` dependency
- To connect the application to config server, we need to give below property pointing to the config server URL.

```
spring.cloud.config.uri=http://config-server:8888
spring.profiles.active=prod
```

4. Enable actuator > refresh API:

- Add `spring-boot-starter-actuator` dependency.
- Annotate beans that should reload on config change with `@RefreshScope`. Example:

```
@RestController
@RefreshScope
public class ConfigController {
    @Value("${featureX.enabled}")
    private boolean featureEnabled;

    @GetMapping("/feature-status")
    public String featureStatus() {
        return "Feature X is " + (featureEnabled ? "ENABLED" : "DISABLED");
    }
}
```

```

    }
}
```

The `@RefreshScope` annotation in Spring Cloud is used to enable dynamic refreshing of configuration properties within a Spring Boot application without requiring a full application restart.

How it works?

- Suppose you want to disable the feature in prod.
- Edit `application-prod.properties` in config-repo branch: `featureX.enabled=false`
- Commit and push changes to Git.
- Go to main running app.
- Call the actuator refresh endpoint:
- POST `http://main-app:8080/actuator/refresh`
- Spring reloads properties from Config Server into your beans without reboot.

Reference: [Complete Guide to Spring Cloud Config | Medium](#)

420 Spring Scenario 9: Large file handling.

Client might upload a large file through the API. How would you handle this to avoid memory issues?

Option 1: Streaming Uploads

Instead of loading the whole file in memory, stream the data chunk by chunk.

```

@PostMapping("/upload-large")
public ResponseEntity<String> uploadLargeFile(HttpServletRequest request) {
    try (ServletInputStream inputStream = request.getInputStream();
        OutputStream outputStream = new FileOutputStream("uploads/largefile.bin")) {

        byte[] buffer = new byte[8192]; // 8KB buffer
        int bytesRead;
        while ((bytesRead = inputStream.read(buffer)) != -1) {
            outputStream.write(buffer, 0, bytesRead);
        }
        return ResponseEntity.ok("File uploaded successfully (streamed).");
    } catch (IOException e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Upload failed.");
    }
}
```

Option 2: Direct-to-Cloud Uploads

- In many production systems, we don't use REST API handle GBs of data directly.
- Instead, we issue a pre-signed URL from S3, GCS, or Azure.
- Client uploads file directly to cloud while your API just coordinates.

Sample code for GCS (Google cloud storage):

```

import com.google.cloud.storage.*;
import org.springframework.web.bind.annotation.*;
import java.net.URL;
import java.util.concurrent.TimeUnit;

@RestController
@RequestMapping("/files")
public class FileController {

    private final Storage storage;

    public FileController(Storage storage) {
        this.storage = storage;
    }

    @GetMapping("/presigned-url")
    public String getPresignedUrl(@RequestParam String fileName) {
        BlobInfo blobInfo = BlobInfo.newBuilder("my-bucket-name", fileName).build();

        URL signedUrl = storage.signUrl(
            blobInfo,
            15, TimeUnit.MINUTES, // URL valid for 15 minutes
            Storage.SignUrlOption.httpMethod(HttpMethod.PUT),
            Storage.SignUrlOption.withV4Signature()
        );

        return signedUrl.toString(); // return signed URL to client
    }
}

```

- The API generates a signed URL valid for 15 minutes.
- Clients use this URL to upload directly to GCS with an HTTP PUT request.
- Our server never touches the file bytes.

Client Upload (Example with cURL)

```

curl -X PUT --upload-file ./bigfile.zip \
"https://storage.googleapis.com/my-bucket-name/bigfile.zip?X-Goog-Algorithm=GOOG4-RSA-
SHA256&X-Goog-Credential=..."

```

421 Spring Scenario 10: Caching.

Your application is having performance issues due to frequent queries to Database. How would you cache the data

Explanation: Spring boot provides various caching solutions such as EhCache, Redis, Hazelcast and In-memory (for small data).

We need to use these Annotations: @EnableCaching, @Cachable, @CachePut, @CacheEvict.

@EnableCaching:

This annotation is placed on a configuration class or the main application class to enable Spring's annotation-driven cache management.

@Cacheable:

This annotation marks a method whose result can be cached. When a method annotated with @Cacheable is invoked, Spring checks if the result for the given arguments is already present in the cache. If found, the cached value is returned, avoiding method execution. If not found, the method is executed, and its result is stored in the cache for future use.

@CachePut:

This annotation is used to update the cache with the new result of a method execution without skipping the method execution itself.

Unlike @Cacheable, @CachePut always executes the method and then places its return value into the cache, ensuring the cache is refreshed with the latest data.

@CacheEvict:

This annotation is used to remove one or more entries from a cache. It is typically applied to methods that modify data, ensuring that stale cached data is removed after an update or deletion operation. This forces subsequent requests for that data to fetch it from the underlying source (e.g., database), ensuring data consistency.

422 Spring Scenario 11: Validations.

Imagine you have a POST API that accepts user details. If the client sends bad data (like missing required fields or too-short passwords), your API should validate, reject gracefully, and return meaningful errors.

Explanation: Spring Boot makes this simple with **Bean Validation (JSR-380)** + annotations like @NotNull, @Size, @Email.

Step 1: Add Validation Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Step 2: Create a DTO with Validation Annotations

```
import jakarta.validation.constraints.*;

public class UserRequest {
```

```

@NotBlank(message = "Name is required")
private String name;

@email(message = "Email must be valid")
@NotBlank(message = "Email is required")
private String email;

@Size(min = 8, message = "Password must be at least 8 characters")
private String password;

// getters and setters
}

```

Step 3: Use @Valid in Controller

```

@RestController
@RequestMapping("/users")
public class UserController {

    @PostMapping
    public ResponseEntity<String> createUser(@Valid @RequestBody UserRequest
userRequest) {
        return ResponseEntity.ok("User created: " + userRequest.getName());
    }
}

```

Here:

- @Valid triggers validation on the incoming JSON.
- If validation fails, Spring throws MethodArgumentNotValidException.
- Then catch the exception with @ExceptionHandler

423 Spring Scenario 12: Scheduling the task.

Imagine you are building an e-commerce application. Every midnight, you need to generate a sales report and send it to management. Doing this manually is painful and error-prone. How would you achieve this?

Explanation:

Spring has a mechanism to schedule the tasks using @EnableScheduling and @Scheduled(Refer Question no. 347,348, Spring 95,96)

1. We need tell Spring to turn on the scheduler by using @EnableScheduling.
2. Now, let's see the job that will run every night at midnight.

```

@Component
public class SalesReportScheduler {

    @Scheduled(cron = "0 0 0 * * ?") // Every day at midnight
    public void generateDailySalesReport() {
        System.out.println("Generating sales report... " + LocalDateTime.now());
    }
}

```

```
// logic: fetch sales data, generate PDF, send email
}
```

Cron: [Cron expression generator by Cronhub](#)

3. Spring will handle them in the background using a default thread pool. If we need a custom thread pool, configure a TaskScheduler bean.

424 Spring Scenario 13: Dependency Injection.

Imagine you have a **UserService** that handles all the logic for fetching users, creating users, updating users, etc. Now, multiple controllers in your application (say AdminController and CustomerController) need to use this same service. How do you make it work cleanly in Spring?

1. Mark the service as a Spring Bean

- We do this with `@Service` (or `@Component`).
- This tells Spring to manage the lifecycle of this service and make it available in the application context.

```
@Service
public class UserService {
    public String getUserDetails() {
        return "User details from service";
    }
}
```

2. Inject the service into controllers

- Since the service is now a bean, we don't need to new it manually.
- We just declare a dependency using `@Autowired` (or constructor injection, which is the better practice).

Example in AdminController:

```
@RestController
@RequestMapping("/admin")
public class AdminController {

    private final UserService userService;

    public AdminController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping("/user")
    public String getUser() {
        return userService.getUserDetails();
    }
}
```

We don't have to write any extra code.

Spring sees UserService in the context, sees that both controllers need it, and injects the same bean instance into both places (because by default, Spring services are **singleton scoped**).

425 Spring Scenario 14: Custom Bean.

Imagine You are building an **e-commerce application**. You need a **utility class** to calculate discounts. It is just a reusable piece of business logic, Like DiscountCalculator.

How would you inject into the Spring context?

Explanation:

Since it is a utility class, it would be idle to use @Bean in Configuration class.

```
public class DiscountCalculator {

    public double applyDiscount(double price, double percentage) {
        return price - (price * percentage / 100);
    }
}
```

This is a plain Java class without any annotation. And then we can inject like this

```
@Configuration
public class AppConfig {

    @Bean
    public DiscountCalculator discountCalculator() {
        return new DiscountCalculator();
    }
}
```

@Configuration tells Spring this class provides bean definitions.

@Bean tells Spring “whenever someone needs DiscountCalculator, use this method to create and manage it.”

Additional Question:

Why don't you use @Service or @Component to DiscountCalculator?

Service and @Component are meant for general Spring-managed beans. Generally, @Service class is to maintain the business logic. DiscountCalculator is more like a utility/helper without business logic. So, annotating it with @Service or @Component adds no real value here.

Common examples are

- RestTemplate: we usually create it as a @Bean in a config class so we can reuse the same instance across the app.

- DataSource : Defined as a @Bean to configure database connections (URL, username, pool size).
- ObjectMapper: Registered as a @Bean when we want custom JSON serialization/deserialization rules.

426 Spring Scenario 15: Inject bean on condition.

You have two payment services, one is for CreditCardService and one is for UPIService. You want to inject only service one depending on config.

Explanation: We need to use @ConditionalOnProperty to achieve this (Refer Question No 278, Spring 26).

Example:

```
public interface PaymentService {
    void pay(double amount);
}

@Service
@ConditionalOnProperty(name = "payment.type", havingValue = "card")
class CardPaymentService implements PaymentService {
    public void pay(double amount) {
        System.out.println("Paid by Card: " + amount);
    }
}

@Service
@ConditionalOnProperty(name = "payment.type", havingValue = "upi")
class UpipaymentService implements PaymentService {
    public void pay(double amount) {
        System.out.println("Paid by UPI: " + amount);
    }
}
```

Only one bean is created at runtime, depending on the condition (payment.type in application.properties file). This avoids conflicts and makes bean injection flexible.

Additional Question:

1. Why can't you use @Profile.

@Profile will be used for environment specific like to load beans based on environment (Dev or Prod)

427 Spring Scenario 16: Inject bean on multiple conditions.

Suppose your app needs to connect to different message brokers (like Kafka or RabbitMQ) but only under certain conditions:

- Broker type should match the property value from application.properties(broker.type).

- The app should also be running on a specific OS (e.g., Linux only).

So, we want the bean created only when multiple conditions are satisfied.

Explanation:

We need use `@ConditionalOnExpression` or custom Condition class

```
@Bean
@ConditionalOnExpression("${broker.type} == 'kafka' and
 '${os.name}'.contains('Linux')")
public MessageBroker kafkaOnLinux() {
    return new KafkaBroker();
}
```

Suppose if we have more complex logic that can't be resolved in `@ConditionalOnExpression` and same condition is being used in multiple times , then we need to go with custom Condition class

```
public class LinuxAndPropertyCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        String os = System.getProperty("os.name").toLowerCase();
        String brokerType = context.getEnvironment().getProperty("broker.type");
        return os.contains("linux") && "kafka".equalsIgnoreCase(brokerType);
    }
}
```

And we will have to use `@Conditional` to trigger the custom Condition class.

```
@Bean
@Conditional(LinuxAndPropertyCondition.class)
public MessageProducer kafkaProducer() {
    return new KafkaProducer();
}
```

428 Spring Scenario 17: Dynamic bean selection, based on parameter from client.

Imagine you are building a payment service. Your app supports multiple payment gateways like PayPal, Stripe, and Razorpay.

- At runtime, the client request parameter (say `paymentMode=paypal`) should decide which bean (payment processor) to use.
- You cannot solve this with `@Profile` or `@ConditionalOnProperty` because that work only at application startup based on config/environment.

Means, when client send `paypal` (POST /pay?type=paypal&amount=100), paypal service should be invoked.

Explanation:

In this case, we inject all candidate beans and pick one dynamically at runtime. Spring won't create/destroy beans per request, but we can route requests to the right bean.

We can achieve this using, Strategy Pattern + @Autowired Map or List

- Spring injects all beans implementing an interface.
- You select the right one using the client parameter.

Strategy pattern means: instead of writing many if-else for different logics, we keep each logic in a separate class and pick the one we need at runtime. This makes code clean, flexible, and easy to change without touching the main class.

Step 1: Define Interface:

```
public interface PaymentProcessor {
    void processPayment(double amount);
}
```

Step 2: Implement multiple beans

```
@Component("paypal")
class PaypalProcessor implements PaymentProcessor {
    public void processPayment(double amount) {
        System.out.println("Paid " + amount + " via PayPal");
    }
}

@Component("stripe")
class StripeProcessor implements PaymentProcessor {
    public void processPayment(double amount) {
        System.out.println("Paid " + amount + " via Stripe");
    }
}
```

Step 3: Use Map Injection for dynamic selection

```
class PaymentService {
    private final Map<String, PaymentProcessor> processors;

    public PaymentService(Map<String, PaymentProcessor> processors) {
        this.processors = processors;
    }

    public void pay(String type, double amount) {
        PaymentProcessor processor = processors.get(type.toLowerCase());
        if (processor == null) {
            throw new IllegalArgumentException("Invalid payment type: " + type);
        }
        processor.processPayment(amount);
    }
}
```

```
}
```

Step 4: Controller

```
@RestController
class PaymentController {
    private final PaymentService paymentService;

    public PaymentController(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    @PostMapping("/pay")
    public void pay(@RequestParam String type, @RequestParam double amount) {
        paymentService.pay(type, amount);
    }
}
```

Now On POST call /pay?type=paypal&amount=100 , Spring routes to PaypalProcessor.

Reference: [Dynamic Bean Switching in Spring Boot ! No If-Else Hell](#)  @Javatechie

429 Spring Scenario 18: Load initial cache data.

Suppose, you have a CacheService which will load some data into cache. How would you achieve this, will you use `@PostConstruct` or `CommandLineRunner`.

Explanation:

@PostConstruct is useful when we want to run some initialization logic right after Spring creates and injects the bean, without needing a `CommandLineRunner` or `ApplicationRunner`. It Runs right after bean creation and dependency injection is done. Best when we need to initialize resources

CommandLineRunner: Runs after the whole Spring context is ready. Best when we need application-wide startup logic. Example: inserting the database

Why can't we use `@Postconstruct` to insert into the DB?

- `@PostConstruct` runs right after the bean is created, before the full Spring Boot context is ready.
- At that time, all beans (like `EntityManager`, `TransactionManager`, `Repositories`) may not be fully available. This can cause errors or missing transaction issues when we try inserting data.
- On the other hand, `CommandLineRunner` or `ApplicationRunner` run after the entire Spring context is ready, meaning our DB connections, repositories, and transactions are fully initialized. Perfect for inserting.

430 Spring Scenario 19: Logging in Spring boot

You need to add logging to your application to track its behaviour and troubleshoot issues. How do you implement logging in Spring Boot?

Spring Boot provides support for various logging frameworks such as Logback, Log4j2, and SLF4J.

SLF4J

SLF4J stands for Simple Logging Facade for Java.

It is not a logging implementation. It is a facade means a common API that routes all the logging calls to a real logging backend like Logback or Log4j.

Think of SLF4J as: “The common interface the code uses to log, while the actual logging engine does the actual work.”

Logback

It is the default logging implementation in Spring Boot.

Created by the same author as Log4j.

Features: faster startup, good async support, easy configuration via logback-spring.xml.

Log4j2

Successor of Log4j (fixed performance & security issues).

Offers advanced features like asynchronous logging (much faster in high-load apps), garbage-free logging (less memory churn).

Additional Questions:

1. Why can't you use simple System.out.println()?

System.out.println() is slow because it writes directly to the console (standard output) synchronously, meaning the main thread waits for each print to complete before moving on.

No log levels: We can't distinguish INFO, DEBUG, WARN, or ERROR easily, with logging frameworks, we can filter messages by level.

2. How would you ensure logging is a singleton in multithreaded programming

In practice, we use existing logging frameworks (SLF4J, Log4j, Logback) since they already guarantee singleton behaviour internally.

Example:

```
public class Logger {
    private Logger() {}
    private static class Holder {
        private static final Logger INSTANCE = new Logger();
    }
    public static Logger getInstance() {
```

```
        return Holder.INSTANCE;
    }
    public void log(String msg) {
        System.out.println(msg);
    }
}
```

431 Spring Scenario 20: Monitoring the Spring application

Your application is in production, and you need to monitor its health, metrics, and logs to ensure it is performing optimally.

Explanation:

Spring Boot Actuator provides a powerful set of tools for monitoring and managing applications.

Add Actuator Dependency.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Enable Actuator Endpoints: management.endpoints.web.exposure.include=*

Access Actuator Endpoints.

- Health Check: <http://localhost:8080/actuator/health>
- Metrics: <http://localhost:8080/actuator/metrics>
- Environment: <http://localhost:8080/actuator/env>

We can also integrate with external monitoring tools like Prometheus and Grafana for more advanced monitoring capabilities.

432 Spring Scenario 21: Post idempotency

A user clicks Place Order multiple times quickly. If your backend just accepts every request and creates an order record each time, you will get duplicate orders.

But HTTP POST is not idempotent by design (every call can cause a new resource).

So you need to make POST “behave idempotently” in your scenario. How do you achieve it?

Solutions

1. Idempotency Key

- The client (e.g., frontend or mobile app) generates a unique key (like UUID) for every “intent to place order”.
- It sends this key in the request header or body:

- POST /orders Headers: Idempotency-Key: abc123
- The server stores this key.
 - If the same key comes again, it returns the same order response instead of creating a new one.

2. Database Constraints (Transactional Safety)

- Put a unique constraint in DB (e.g., user_id + cart_id must be unique).
- If user clicks multiple times, the DB prevents duplicates.
- The backend can catch the DB exception and return the existing order.

433 Spring Scenario 22: Custom Annotation + Interceptor

Imagine in your application there was a custom annotation called @Adminonly. Now You need to make sure, if any delete/* API is not annotated with, should reject the request.
Kind of centralized check for delete APIs.

Solution:

We have already had custom annotation.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface AdminOnly {

}
```

1. Create Interceptor

In Pre Handle method.

```
@Component
public class DeleteApiInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) {
        if (handler instanceof HandlerMethod handlerMethod) {
            AdminOnly adminOnly = handlerMethod.getMethodAnnotation(AdminOnly.class);
            if (adminOnly == null) {
                // Check if user has ADMIN role
                response.setStatus(HttpServletRequest.SC_FORBIDDEN);
                return false;
            }
        }
        return true;
    }
}
```

2. Register the Interceptor

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    private final DeleteApiInterceptor deleteApiInterceptor;

    public WebConfig(DeleteApiInterceptor deleteApiInterceptor) {
        this.deleteApiInterceptor = deleteApiInterceptor;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(deleteApiInterceptor)
            .addPathPatterns("/delete/*"); // Apply only to /delete/*
    }
}
```

3. Controller

```
@RestController
@RequestMapping("/api/v1/users")
public class UserController {

    // This delete API requires @AdminOnly
    @DeleteMapping("/{id}")
    @AdminOnly
    public ResponseEntity<String> deleteUser(@PathVariable Long id) {
        // Imagine delete logic here
        return ResponseEntity.ok("User with ID " + id + " deleted successfully");
    }

    // This endpoint does not have @AdminOnly
    @GetMapping("/{id}")
    public ResponseEntity<String> getUser(@PathVariable Long id) {
        // Just dummy return
        return ResponseEntity.ok("Fetched user with ID " + id);
    }
}
```

We have registered the DeleteApiInterceptor so that it runs for every request under delete/*.

Inside the interceptor, we use HandlerMethod to check if the controller method handling this request is annotated with @AdminOnly.

- If the method **has the annotation**, the request continues as normal.
- If the method **does not have the annotation**, we block the request and return a 403 Forbidden.

This way, even though the interceptor is invoked for all delete/* APIs, only the methods explicitly marked with @AdminOnly are allowed to execute.

434 Spring Scenario 23: Rate Limiting

You have a public API `/api/search` that clients use to query your system. Some clients start abusing it, sending hundreds of requests per second, which puts unnecessary load on your backend. You want to ensure:

1. Each client (based on IP or API key) can only make 5 requests per second.
2. If they exceed this limit, they should receive an HTTP 429 Too Many Requests.
3. This should not affect other clients.

We can achieve this using Custom Filter.

Code Link: [Spring-Samples/RateLimiting at main · CodingLyf-Fullstack/Spring-Samples](#)

Steps

- Create a custom filter called RateLimitingFilter
- The RateLimitingFilter class implements the Filter interface to create a custom filter for rate limiting.
- A ConcurrentHashMap is used to store request counts per IP address for thread-safe operations.
- The doFilter method increments the request count and checks if it exceeds the defined limit. If so, it responds with a 429 Too Many Requests status.

FilterRegistrationBean

When we write a filter in Spring Boot, we usually extend OncePerRequestFilter or implement Filter. That filter can intercept every HTTP request and response.

But here is the problem:

By default, Spring Boot will pick up the filter and apply it globally to all the URLs. But when we want more control like deciding the order, Apply the filter to specific URL patterns (like /api/*) then we will use FilterRegistrationBean

```
@Bean
public FilterRegistrationBean<CustomFilter> customFilter() {
    FilterRegistrationBean<CustomFilter> reg = new FilterRegistrationBean<>();
    reg.setFilter(new CustomFilter());
    reg.addUrlPatterns("/api/*"); // only apply to /api
    reg.setOrder(1);           // set execution order
    return reg;
}
```

435 Spring Scenario 24: Custom Query method

You are building a Spring Data JPA repository and you want to fetch all users that belong to a certain email domain. For example, get everyone with an email ending in @gmail.com but without JPQL.

How would you do this?

Solution

Spring Data JPA has a feature called **Derived Query Methods**. Instead of writing the whole query, we just follow a naming convention in our repository method. Spring automatically interprets it into SQL/JPQL for us.

Example:

Repository:

```
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByEmailEndingWith(String domain);
}
```

When we call the method

```
List<User> gmailUsers = userRepository.findByEmailEndingWith("@gmail.com");
```

Spring Data JPA translates it into something like:

```
SELECT * FROM user WHERE email LIKE '%@gmail.com';
```

436 Spring Scenario 25: Custom Query

You are building an application to manage Orders placed by Users. Now, you need to fetch all orders placed by users older than 25, where the order status is 'DELIVERED'.

Solution

Here we need to write query that involves joins, filtering, or multiple conditions that we cannot achieve with Derived Query Method.

In Spring Data JPA, we can write custom queries in two ways:

1. JPQL (object-oriented query, works with entity names and fields), using @Query
2. Native SQL (direct database query)

Example:

Our Scenario involves 2 tables (User, Orders)

```
@Entity
public class User {
    @Id
    private Long id;
    private String name;
    private int age;
}
```

```
@Entity
public class Order {
    @Id
    private Long id;
    private String status;
    private double productPrice;

    @ManyToOne
    private User user;
}
```

So, a User can have many Orders. That is why Order holds a reference to User.

Now, derived queries (like findByStatus) won't help here, because we need conditions across two tables. This is where `@Query` comes into play

```
public interface OrderRepository extends JpaRepository<Order, Long> {

    @Query("SELECT o FROM Order o JOIN o.user u " +
           "WHERE u.age > 25 AND o.status = 'DELIVERED'")
    List<Order> findDeliveredOrdersByUsersOlderThan25();
}
```

- JOIN o.user u: joins the Order table with its related User.
- u.age > 25: filters by user age.
- o.status = 'DELIVERED': filters by order status.

437 Spring Scenario 26: Lazy vs Eager / N+1 problem

You have two entities:

- User
- Order

A user can have many orders. Sometimes you just want the user's details (name, age, etc.). Other times, you also want their orders.

The question is: how do you design it so you don't always drag orders out of the database unnecessarily, but can still fetch them when you actually need them?

Solution:

First let's understand the relationships.

1. A User can have many orders (OneToMany)

2. Many Orders can be placed by a user (ManyToOne)

Entities:

```
@Entity
public class User {
    @Id
    private Long id;
    private String name;
    private int age;

    @OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
    private List<Order> orders;
}
```

Notice fetch = FetchType.LAZY.

- This means: When I load a User, don't immediately pull in their Orders.
- Orders will only load when we call user.getOrders().

Now we can get the user, only user will be returned. But when we want to fetch the orders, a separate query will be made to fetch orders.

```
User user = userRepo.findById(1L).get();
System.out.println(user.getName()); // Only User query runs

List<Order> orders = user.getOrders(); // triggers a separate query
```

If there are 100 users, for each user one order will be made, which creates N+1 problem (Refer Question No. 303,304 Spring 51,52);

To solve this, we have two approaches.

1. Use JOIN FETCH: This fetches all the data in one query (Like a join query in SQL)

```
@Query("SELECT u FROM User u JOIN FETCH u.orders WHERE u.id = :id")
User findUserWithOrders(@Param("id") Long id);
```

2. Use @EntityGraph: This is cleaner, no custom query needed:

```
@EntityGraph(attributePaths = "orders")
@Query("SELECT u FROM User u WHERE u.id = :id")
User findUserWithOrders(@Param("id") Long id);
```

438 Spring Scenario 27: save() or saveAll()

You are building a system where you need to persist data into the database. Sometimes it is just 1 record (like a single order). Other times, it is 10000 records at once (say, importing bulk Orders from a file).

Do you use save() or saveAll()?

Solution:

- save() is used to save single record.
- saveAll() is used to save list of the records.

```
List<User> users = List.of(
    new User(2L, "Coding", 30),
    new User(3L, "Lyf", 25)
);

userRepository.saveAll(users);
```

But here is the catch. saveAll() internally runs for loop and save() each record. If there are 10000 records, it will loop 10000 times. Best technique to persist N number of records is by **batching**.

Hibernate can group those inserts into batches. All we need is to configure it:

```
spring.jpa.properties.hibernate.jdbc.batch_size=500
spring.jpa.properties.hibernate.order_inserts=true
spring.jpa.properties.hibernate.order_updates=true
```

Now, if we call saveAll() with 10000 records:

- Instead of 10000 separate INSERTs, Hibernate will do them in batches of 500.
- That means only 20 round trips to the database. Much faster.

439 Spring Scenario 28: Partial Update

You are working with a user entity that has multiple fields:

```
@Entity
public class User {
    @Id
    private Long id;
    private String name;
    private int age;
    private String status;
}
```

Now imagine the requirement:

You want to update only the status of a user (say from INACTIVE to ACTIVE) without fetching the entire User object first.

Step 1: The Problem

Generally, In JPA, first we get the record and update it. Which means two queries will run to update single field.

```
User user = userRepo.findById(1L).get(); // 1 query
user.setStatus("ACTIVE");
userRepo.save(user); // another query (update)
```

Step 2: Solution.

To avoid select query we can @Modifying + @Query for partial updates.

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Modifying
    @Query("UPDATE User u SET u.status = :status WHERE u.id = :id")
    int updateUserStatus(@Param("id") String status, @Param("id") Long id);
}
```

- @Modifying: tells Spring this is an update/delete, not a select.
- JPQL UPDATE query updates only the status field.
- int return value: number of rows updated.

Point to remember: We can achieve Delete also with @Modifying. We can also use it for bulk updates where can update the record without fetching it first

440 Spring Scenario 29: Soft Deletes

Instead of deleting, you need to mark a record as inactive How to implement soft delete in JPA?

Solution:

Step1: Add a field in the entity.

```
@Entity
public class User {
    @Id
    private Long id;
    private String name;
    private int age;
    private boolean deleted = false; // soft delete flag
}
```

Now instead of delete we can run update using @Modifying.

Or we can use @SQLDelete

The **@SQLDelete** annotation is a Hibernate-specific annotation (not part of the standard JPA specification) which is useful to **override the default SQL statement** that Hibernate uses to delete an entity from the database.

Instead of the standard DELETE FROM table_name WHERE id = ?, we can define a custom SQL query to be executed when entityManager.remove() is called.

We define @SQLDelete in our entity like this:

```
@Entity
@SQLDelete(sql = "UPDATE user SET deleted = true WHERE id = ?")
@Where(clause = "deleted = false")
public class User {
    @Id
    private Long id;
    private String name;
    private int age;
    private boolean deleted = false;
}
```

When we run userRepo.deleteById(1); hibernate actually runs the Update query.

```
update user set deleted = true where id = 1;
```

441 Spring Scenario 30: Auditing

You are building an Orders System. Every order has details like product, price, and status. We need to know when each order was created and when it was last updated. How would you achieve this?

Solution:

The naive way is to add two fields: createdDate and updateDate and then, every time we save an entity, set them manually:

But Spring Data JPA has built-in support for auditing using annotations:

- @CreatedDate: set once when the entity is created.
- @LastModifiedDate: updated every time the entity changes.

And all of this works by simply enabling auditing (Refer Question No. 361 Spring 109).

Example:

```
@SpringBootApplication
@EnableJpaAuditing // switch ON auditing
public class App {
```

```
}
```



```
@Entity
@EntityListeners(AuditingEntityListener.class)
public class Order {

    @Id
    @GeneratedValue
    private Long id;

    private String product;
```

```

    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createdDate;

    @LastModifiedDate
    private LocalDateTime updatedDate;
}

```

Now when we call `orderRepo.save(order);` `createdDate` and `updatedDate` will be saved automatically.

442 Spring Scenario 31: Handling null values

User is trying to fetch the details of the book that is not exist in the database. how would you gracefully handle this in Spring?

Approach:

- JPA uses `Optional` when fetching by ID.
- If the book is not found, throw a **custom exception**.
- Use `@ControllerAdvice` and `@ExceptionHandler` to return a proper error response.

Sampel code

Create Entity:

```

@Entity
public class Book {
    @Id
    private Long id;
    private String title;
    private String author;
}

```

Create Repository:

```

public interface BookRepository extends JpaRepository<Book, Long> {
}

```

Custom Exception:

```

public class BookNotFoundException extends RuntimeException {
    public BookNotFoundException(Long id) {
        super("Book with id " + id + " not found");
    }
}

```

Service

```
@Service
public class BookService {
    @Autowired
    private BookRepository repository;

    public Book getBook(Long id) {
        return repository.findById(id)
            .orElseThrow(() -> new BookNotFoundException(id));
    }
}
```

Global Exception handler:

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(BookNotFoundException.class)
    public ResponseEntity<String> handleBookNotFound(BookNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
    }
}
```

In Service, we are invoking `findById()` which returns `Optional`, if not found we are throwing a custom exception. That custom exception will be caught by Global exception handler and returns `NOT_FOUND` message to User.

443 Spring Scenario 32: Transaction rollback failed

Suppose, you have a service method that saves Employee data and Department data. Both need to be saved in one transaction so that if anything goes wrong, neither is saved (rollback).

But what happened in your case?

- Employee was saved successfully.
- Department failed due to an exception.
- Rollback didn't happen means Employee record is still in the database.

Why rollback did not happen in this case?

Solution:

Springs `@Transactional` only rolls back automatically on **unchecked exceptions** (runtime exceptions).

By default:

- `RuntimeException`: triggers rollback
- Errors: triggers rollback
- Checked exception (like `IOException`, `SQLException`): does **not** trigger rollback

For checked Exceptions we need to use **rollbackFor** explicitly:

The rollbackFor attribute in Spring's @Transactional annotation is used to specify which Exception types should trigger a transaction rollback.

```
@Transactional(rollbackFor = {MyCheckedException.class, AnotherCheckedException.class})
public void doSomethingTransactional() throws MyCheckedException,
AnotherCheckedException {
    // ... business logic ...
    if (someCondition) {
        throw new MyCheckedException("Error condition met, rolling back.");
    }
}
```

When we use rollbackFor, it adds to the default rollback behaviour, rather than replacing it. This means that if we specify rollbackFor = {MyCheckedException.class}, the transaction will still roll back on RuntimeExceptions and Errors, in addition to MyCheckedException.

444 Spring Scenario 33: Read-only transaction

Imagine You are building an **Employee Management System**. You have a method to fetch employees from the database. Since You are only reading data (no inserts, no updates),

How will make fetchEmployees() read-only?

Solution:

We need to use `@Transactional(readOnly = true)`

```
@Transactional(readOnly = true)
public List<Employee> getAllEmployees() {
    return employeeRepository.findAll();
}
```

The `@Transactional(readOnly = true)` annotation in Spring is used to declare that a method or class's operations within a transaction will only read data and will not modify it.

`@Transactional(readOnly = true)` has a lot of advantages.

1. performance improvement: read-only entities are not dirty-checked
2. memory saving: snapshots of persistent state are not maintained
3. data consistency: the changes of read-only entities are not persisted
4. database load: when we use master-slave , or read-write replica set(or cluster), `@Transactional(readOnly = true)` makes us enable to connect to read-only DB.

445 Spring Scenario 34: Event driven

You have a POST API:

- When a new reel gets posted, you need to send notifications to N users (could be hundreds, thousands, or even millions).
- Key challenge: if you try to notify all users synchronously inside the request, the API call will be slow or even timeout.

So how do you design it?

Solution: Event-Driven Approach

We need a system to handle notifications asynchronously. Using,

Message Queues: RabbitMQ, Kafka, Google Pub/Sub, AWS SQS.

446 Spring Scenario 35: Improve initial loading time or Cold Start Optimization Techniques

Your Spring Boot application is taking too much time during the initial load. Users notice that the application takes 10–15 seconds before the first API call responds after startup.

Solution.

When we start the application, following things will happen:

- Loading JVM
- Initializing the Spring context.
- Initiating the beans.
- Connecting to DB
- Setting up security, filters etc

Steps to improve.

Use **actuator/startup** endpoint to analyse which beans are taking time.

Enable lazy initialization using @Lazy:

In Spring Boot, all beans get initialized at startup by default.

If we have complex beans (like DataSources, caches, external API clients), startup time will increase. So use **@Lazy** to load non-critical beans only when needed.

Tune the auto-configuration:

Spring Boot auto-configures does many things we may not use (JDBC, JPA, Security, Actuator, etc.). This adds to startup.

Use **@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})** for things that are not needed.

Use Spring AOT + Native image.

Instead of compiling bytecode into native code at runtime (like the JVM with JIT does), AOT compiles the code ahead of time into a native executable. In traditional Java applications, the Just-in-Time (JIT)

compiler compiles code during runtime, which can introduce some delays. With AOT, Spring 6 aims to overcome these delays by performing the compilation ahead of time, which improves startup performance and resource usage.

447 Spring Scenario 36: Public some APIs

You are building an e-commerce application with Spring Boot.

- Pages like Home, Product Listing, and Contact Us should be accessible to everyone (public).
- Pages like Add Product, Delete Product, and User Management should only be accessible to Admins.

Solution:

Step 1: Define roles

In the application, we will have two main roles:

- ROLE_USER: can view products, browse, buy.
- ROLE_ADMIN: can do everything a user can, plus manage products and users.

Step 2: Secure endpoints with Spring Security

We use Spring Security (SecurityFilterChain) to configure which URL endpoints are public and which require authentication.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeHttpRequests()
            // Public pages
            .requestMatchers("/", "/home", "/products", "/contact").permitAll()

            // Admin-only pages
            .requestMatchers("/admin/**").hasRole("ADMIN")

            // Everything else requires login
            .anyRequest().authenticated()

            .and()
            .formLogin() // default login page
            .and()
            .logout();

        return http.build();
    }
}
```

Using requestMatchers we can mention which APIs are public and which are should be accessed by admin.

448 Spring Scenario 37: Only ADMIN operation

Imagine You are building an Employee Management System for a company. The system has two main types of users:

1. Regular employees: They can log in, update their own profile, and view their payslip.
2. Admins (HR team): They can add new employees, update details of any employee, and delete employees if needed.

Now, deleting a user is a sensitive operation.

How you implement this example Delete operation should be done only by Admins?

Solution:

Spring Security gives a clean way to enforce this kind of restriction at the method level use PreAuthorize or PostAuthorize.

```
@PreAuthorize("hasRole('ADMIN')")
public void deleteUser(Long userId) {
    // delete logic here
}
```

- When anyone calls deleteUser(...), Spring Security checks the logged-in user.
- If that user doesn't have the role ADMIN, Spring throws an AccessDeniedException before the method body is even executed.
- If the user does have the role, then and only then the deletion happens.

Note: We can also achieve it using requestMatchers

```
http.authorizeHttpRequests()
.requestMatchers(HttpMethod.DELETE, "/employees/**").hasRole("ADMIN");
```

This works fine as long as delete is always triggered via the REST endpoint /employees/{id}. Any call to that endpoint requires the ADMIN role.

But, Since deleting employees is a sensitive business action, method-level security is the safer choice.

Why? Because:

1. Security should be with the business logic (the delete method), not just with one endpoint.
2. If later we add another controller, a scheduled task, or an internal service call deleteEmployee(), the security rule should still apply.

449 Spring Scenario 38: Monitoring the Application

The operation teams want to monitor the application. how can you give APIs to them to monitor?

Solution:

Spring Boot provides **Actuator** which gives production-ready monitoring APIs. By enabling it, we can expose endpoints that the Operations team can consume (Refer Questions No.336 Spring 84)

450 Spring Scenario 39: Inter-service communication

Imagine you have a big e-commerce platform. It is not just one big application. Instead, you have split it into multiple services (microservices). For example:

- User Service: manages users, profiles, authentication
- Order Service: manages orders, cart, checkout
- Payment Service: processes payments
- Inventory Service: keeps track of stock

How you make communication between the services?

Approaches:

1. Synchronous Communication:

Example: The Order Service makes a REST API call to the Payment Service (POST /payments) and waits until it gets a success or failure response.

Tools we can use:

- REST (HTTP/JSON): simple and widely used
- gRPC: faster, binary protocol, good for high-performance communication

In Spring, we can use RestTemplate or WebClient(Refer Questions No. 411 Rest 29)

2. Asynchronous Communication (Message driven)

Example: The Order Service publishes an event like OrderCreated to a message broker. The Payment Service listens for that event, processes payment

Tools we can use:

- Kafka
- RabbitMQ
- Google Pub/Sub / AWS SQS

451 Spring Scenario 40: Searching Million Records

You are building a REST API for an e-commerce app. The dataset contains millions of products. The client asks for search functionality where users can quickly find products by name, category, or description.

Explanation:

If we use simple LIKE %keyword% queries on such a huge dataset, performance will degrade badly because SQL has to look at every single row, one by one.

For large datasets, we cannot rely on plain database queries. You need a search optimization strategy.

Database indexing (Refer Question No. 372 Spring 120)

- Create indexes on frequently searched fields (e.g., product name, category).
- This makes queries like WHERE name LIKE 'abc%' much faster.

Full-Text Search

- Use native DB support (FULLTEXT INDEX in MySQL, tsvector in PostgreSQL).

External Search Engines (for very large scale)

- Use tools like **Elasticsearch** or **OpenSearch**.
- They're built for text search and can handle millions of records efficiently.

Caching

- Cache frequently searched queries with Redis to avoid hitting DB repeatedly.

Full-Text Search:

FULLTEXT is a special index designed for text search. It helps you search big text fields (like names, descriptions, articles, blogs) much faster and smarter.

When we add a FULLTEXT index, SQL creates its own “search-friendly” copy of the text data. So instead of checking row by row, SQL jumps directly to matching results.

Reference:

[MySQL FULLTEXT Indexes: Usage & Examples](#)

[MySQL and PostgreSQL for Advanced Full-Text Search.](#)

452 Spring Scenario 41: Process Million Records (**NOTE: It is about processing not searching**)

How would you handle a situation where a Java application needs to process millions of records efficiently?

Explanation:

If we just load everything into memory and loop over it, the JVM will blow up with OutOfMemoryError. So, we need to design this smartly.

1. Implement pagination (Refer Question No. 382 Spring 130)

Instead of pulling all records into memory, fetch them in chunks (pagination).

2. Use Spring Data JPA Stream APIs

Instead of fetching everything at once. Fetch records little by little, and process them as they come. SQL databases can return results in a **streaming fashion** instead of dumping all rows at once

Example:

```
@Repository
interface PostRepository extends JpaRepository<Post, Long> {

    @Query("""
        select p
        from Post p
        where date(p.createdOn) >= :sinceDate
        """
    )
    @QueryHints(
        @QueryHint(name = AvailableHints.HINT_FETCH_SIZE, value = "25")
    )
    Stream<Post> findPosts(@Param("from") LocalDate fromDate);
}
```

- This repository method fetches posts created on or after a given date using JPQL.
- The @QueryHint sets the fetch size to 25, so JPA loads rows in chunks instead of all at once.
- It returns a Stream<Post>, allowing lazy, memory-efficient processing of results.

Reference: [Spring Data JPA — batching using Streams | by Patrik Hörlin | Predictly on Tech | Medium](#)

[The best way to use Spring Data JPA Stream methods - Vlad Mihalcea](#)

3. Parallelize the work

Millions of records usually mean *independent tasks*. We can opt for parallel processing. Or use a thread pool (ExecutorService) if we want more control.

```
listOfRecords.parallelStream()
.forEach(this::processRecord);
```

4. Batch processing

If the job involves writing back to DB, instead of writing them row by row better to collect and write in batches.

```
List<Customer> batch = new ArrayList<>();
for (Customer c : records) {
    batch.add(c);
    if (batch.size() == 1000) {
        customerRepository.saveAll(batch);
        batch.clear();
    }
}
```

5. Asynchronous / Queue-based processing

We can use Kafka, RabbitMQ, etc. for high scalability. Push the records into Queue and let consumers process them in parallel.

453 Spring Scenario 42: OutOfMemory due to huge data.

How did you reduce memory consumption in a high-load system? A system with 500,000 users had frequent OutOfMemoryError. How would you prevent it?

1. Use Streams and Pagination instead of Loading Everything

Instead of fetching all 500,000 users into memory, always use pagination (Pageable in Spring Data JPA) or Java Streams with @QueryHint(FETCH_SIZE) where supported.

This way we only hold a fraction of the dataset in memory at a time.

2. Reduce Object Retention with Weak/Soft References

Often memory leaks happen because objects stay referenced longer than needed (e.g., static caches, thread locals).

Use WeakReference or SoftReference for caches where data can be reloaded instead of holding strong references forever.

```
private Map<String, WeakReference<Product>> cache = new ConcurrentHashMap<>();
```

3. Tune Collections and Object Sizes

Replace heavy collections with memory-efficient ones:

- Use ArrayList instead of LinkedList for large lists.
- Use EnumSet or EnumMap instead of HashMap if keys are enums.

- Avoid unnecessary String duplication , use intern() where safe, or use StringBuilder instead of + in loops.
- Prefer primitives instead of Wrapper classes when possible . Use int[], long[], etc. instead of List<Integer>, List<Long> when ever applicable .

```
EnumMap<Category, List<Product>> productsByCategory = new EnumMap<>(Category.class);
```

454 Spring Scenario 43: Writing to DB by two threads

How did you handle a situation where two threads were updating the same record in the database?

Solution:

When two threads trying to update the same record in the database at the same time. This creates a race condition. Sometimes one thread's changes would overwrite the other's, leading to inconsistent data.

We need to use a technique called “**Optimistic Locking**.”

In Hibernate/JPA, this is simple to set up. We add a @Version field to the entity. Hibernate automatically manages this field whenever an update happens.

How It Works

1. Each record has a version number (integer or timestamp).
2. When Thread A reads the record, it sees version = 1. Thread B gets the same version = 1.
3. Thread A updates the record, Hibernate increments version : 2.
4. Thread B tries to update based on version = 1. Hibernate detects mismatch (because DB now has version = 2).
5. Thread B's transaction fails with an OptimisticLockException.
6. Instead of silently losing data, we catch this exception and retry the operation with fresh data.

Whenever we get OptimisticLockException, we can retry

Example:

```
@Entity
public class Product {
    @Id
    private Long id;

    private String name;

    @Version
    private Integer version; // Hibernate manages this
```

```
}
```

```
@Transactional
public void updateProduct(Long productId, String newName) {
    boolean success = false;
    int retries = 3;

    while (!success && retries > 0) {
        try {
            Product product = productRepository.findById(productId).orElseThrow();
            product.setName(newName);
            productRepository.save(product);
            success = true;
        } catch (OptimisticLockException e) {
            retries--;
            if (retries == 0) {
                throw e; // give up after retries
            }
        }
    }
}
```

455 Spring Scenario 44: Parallel API call

Suppose you have to make N number of APIs. for example, there could be 1,00,000 users for each user you need to make API call. how would you do that?

Solution:

1. Async with CompletableFuture

We can do parallel calls with the combination of @Async and CompletableFuture.

Async method to run in the background.

```
@Async
public CompletableFuture<String> syncUser() {
    return CompletableFuture.supplyAsync(() ->
        restTemplate.getForObject("https://api.external.com-sync/", String.class)
    );
}
```

Making API calls by iterating users.

```
List<CompletableFuture<String>> futures = users.stream()
    .map(this::syncUser)
    .collect(Collectors.toList());

// Wait for all to complete
CompletableFuture.allOf(futures.toArray(new CompletableFuture[0])).join();
```

2. Reactive Approach (WebClient + Flux)

If we use on Spring WebFlux, WebClient is non-blocking and much more efficient for massive N.

```
Flux<User> users = Flux.fromIterable(userList)
    .flatMap(user -> webClient.get()
        .uri("https://api.external.com-sync/{id}", user.getId())
        .retrieve()
        .bodyToMono(User.class))
    .collectList()
    .block(); // waits for all results
```

456 Spring Scenario 45: Secure the App

How would you secure the Spring application that should be accessed only to registered users?

Solution:

1. Add `spring-boot-starter-security` starter dependency to the project.
2. Implement Authentication like BasicAuth or Token Based Authentication
3. Implement Authorization, means Apply role-based (RBAC) for access control. Example: Admin can delete users; a normal user cannot.
4. **Encrypt the sensitive data:** For example, passwords must be encrypted before saving to DB. Use BCryptPasswordEncoder

5. Transport Security (TLS): To protect data in transit, always use HTTPS instead of plain HTTP. Redirect all HTTP traffic to HTTPS so no user accidentally connects over an insecure channel. In Spring Boot, enable SSL in the application:

```
server:
  ssl:
    enabled: true
    key-store: classpath:keystore.p12
    key-store-password: yourpassword
    key-store-type: PKCS12
```

6. Input Validation & Sanitization

- Validate all inputs (length, type, allowed values).
- Escape or sanitize data before storing/using.
- Protect against SQL Injection, XSS, Command Injection.

- Use parameterized queries with JPA/JDBC instead of string concatenation.

7s. Rate Limiting & Throttling

- Protect APIs from brute-force or denial-of-service attacks.
- Implement rate limiting (e.g., 100 requests/min per IP).
- Use tools like Bucket4j, Redis, or API Gateway throttling.

457 Spring Scenario 46: Prevent SQL Injection

How do you prevent SQL injections in Spring boot App?

Let's understand how SQL Injection Happens, Causes and Prevention.

SQL Injection (SQLi) is a type of security vulnerability where an attacker manipulates your SQL queries by injecting malicious input. Attacker can read, modify, or delete the database without proper authorization

This can happen when user input is inserted directly into SQL queries without any safety checks. This allows an attacker to manipulate the query and run any SQL they want.

Example:

Let's say we have written query like this to accept a username from Frontend and return user details based on username

```
String username = request.getParameter("username");
String query = "SELECT * FROM users WHERE username = '" + username + "'";
```

If an attacker enters this as the username: '`;` `DELETE FROM users; --`'

Then the resulting SQL becomes, where attacker can delete users table

```
SELECT * FROM users WHERE username = ''; DELETE FROM users; -- '
```

Common Causes

1. String concatenation in queries : building SQL using raw input.
2. No input validation : allowing unexpected characters like `', ;, --`.

Prevention Techniques

a) Use Parameterized Queries / Prepared Statements

Always let the database handle user input safely.

```
String sql = "SELECT * FROM users WHERE username = ? AND password = ?";
PreparedStatement ps = conn.prepareStatement(sql);
```

```

ps.setString(1, username);
ps.setString(2, password);
ResultSet rs = ps.executeQuery();

```

Here, even if the user types ' ; DELETE FROM users; --, it is treated as a literal value, not SQL code.

b) Use ORM Properly (JPA / Hibernate)

Avoid concatenating strings in queries. Use parameters:

```

@Query("SELECT u FROM User u WHERE u.email = :email")
User findByEmail(@Param("email") String email);

```

458 Spring Scenario 47: Prevent XSS attacks

How do you prevent SQL injections in Spring boot App?

Let's understand how XSS Injection Happens, Causes and Prevention.

Cross-Site Scripting (XSS) happens when an attacker manages to inject malicious scripts into a website so the other users will execute in their browsers. Essentially, the attacker tricks the browser into running their JavaScript.

Impact: leading to data theft, session hijacking, or redirection to malicious sites

How XSS Happens

The attacker looks for a web page where users can submit input, like a comment box, without proper validation or escaping.

They write a small JavaScript snippet to do something harmful, like stealing cookies or session info.

The attacker submits the script in the comment box. The backend stores it in the database without cleaning it.

When other users visit the page and see the comment, their browsers automatically run the attacker's script, letting the attacker steal information or manipulate the page.

Causes of XSS:

No input validation

- The app doesn't check or clean what users type before showing it.
- Example: allowing <script> in a comment.

No output escaping

- Special characters like < and > aren't converted, so the browser treats them as code instead of plain text.

Using innerHTML unsafely

- Setting innerHTML with raw user input can run scripts directly on the page.

Prevention

Input Validation

Always validate and escape all user input to ensure it is treated as text, not executable code.

Sanitize Output

Encode HTML entities when displaying user input.

```
String safeComment = StringEscapeUtils.escapeHtml4(userComment)
```

Content Security Policy (CSP):

Implement a CSP header to instruct the browser on which sources are allowed to execute scripts, effectively blocking injected, untrusted scripts

Enable XSS Protection and prevent clickjacking.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .headers(headers -> headers
                // 1. Content Security Policy
                .contentSecurityPolicy(csp ->
                    csp.policyDirectives("default-src 'self'; script-src 'self'"))

                // 2. XSS Protection
                .xssProtection(xss -> xss.block(true))

                // 3. Clickjacking prevention
                .frameOptions(frame -> frame.sameOrigin())
            );
        return http.build();
    }
}
```

- default-src 'self': only allow scripts from your own domain.
- xssProtection(true): blocks some reflected XSS attacks.
- frameOptions sameOrigin: prevents your pages from being loaded in a malicious iframe.

459 Spring Scenario 48: Best Practices while designing the REST API

When we design REST APIs, the first thing to keep in mind is **clarity**. Our endpoints should tell a story. For example, if we want all users, use `/users`. If we want a specific user, use `/users/{id}`. Avoid vague names like `/getUserInfo`.

Next comes **HTTP methods**. Use them for their true purpose. GET for fetching data, POST for creating, PUT or PATCH for updating, and DELETE for removing. Don't mix them up. A GET request should never modify data.

Status codes are the language back to the client. If something is created successfully, say 201 Created, not just 200 OK. If input is wrong, return 400 Bad Request. These codes make debugging and integration easier.

Now, **validation**. Never trust incoming data blindly. Use Spring's validation annotations (@Valid, @NotNull, etc.) to ensure the request is clean before processing it.

For **error handling**, don't scatter try-catch blocks everywhere. Centralize them with @ControllerAdvice. This way, clients get consistent error responses instead of random stack traces.

Think about **security**. Use authentication (JWT, OAuth2, BasicAuth if simple), and always validate who is accessing the endpoint. Also, never return sensitive details from the responses.

Finally, consider **versioning**. APIs evolve, and breaking old clients is a bad move. Start with /api/v1/... and move to /api/v2/... when needed.

And one last thing, document **the APIs**. Whether it is Swagger or OpenAPI, make sure anyone can understand and test the endpoints without guessing.

460 Spring Scenario 49: Design the Entity Relationships

Imagine you are implementing a system where Students and Departments need to be saved in DB. How would you design Entity and relationships

Solution

Entities

- Department: Represents an academic department (like Computer Science, Mechanical, etc). One department can have many students.
- Student: Represents an individual student. Each student belongs to exactly one department.

So, the relationship here is:

One Department can contain Many Students (One-to-Many). At the same time

Many Students can be there in One Department (Many-to-One).

Department.java

```
@Entity  
public class Department {  
  
    @Id
```

```

@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String name;

// One Department -> Many Students
@OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
private List<Student> students;

// getters and setters
}

```

Student.java

```

@Entity
public class Student {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String name;

// Many Students -> One Department
@ManyToOne
@JoinColumn(name = "department_id") // foreign key column
private Department department;

// getters and setters
}

```

In Department Entity, In @OnetoMany, we have used mappedBy, mappedBy tells “Don’t create a separate join table or foreign key for this collection. The ownership is on the Student entity, look at its department field for the actual mapping.”

On the Student side, we define @JoinCloumn which tells “In the students table, create a column department_id and use it as a foreign key referencing the department table.”

This is how tables look like:

department Table

| department_id | name | location |
|---------------|------------------|----------|
| 1 | Computer Science | Block A |
| 2 | Mechanical Engg | Block B |

| department_id | name | location |
|---------------|------------|----------|
| 3 | Civil Engg | Block C |

student Table

| student_id | name | roll_number | email | department_id |
|------------|--------|-------------|---------------------|---------------|
| 101 | Ram | CS001 | alice@college.edu | 1 |
| 102 | Charan | CS002 | bob@college.edu | 1 |
| 103 | Mahesh | ME001 | charlie@college.edu | 2 |
| 104 | Diana | CE001 | diana@college.edu | 3 |
| 105 | Raja | CS003 | ethan@college.edu | 1 |

Explanation of Relationship

- Each student has a department_id foreign key that links to the department table.
- A department can have many students, but each student belongs to only one department.

So, in DB terms:

- department_id in student is a foreign key referencing department.department_id.

461 Spring Scenario 50: End to End High level Design

Build a system like **Swiggy or Uber Eats**, where users can order food from restaurants, and restaurants manage their menu.

Interview Tip: While designing the systems:

- Start by clearly defining the problem and scope. Identify core entities and their relationships.
- Choose technologies for backend, frontend, database, cloud and include best practices.
- Design the APIs and clearly mention HTTP methods that are going to be used. Explain architecture layer by layer, including async processing if needed.
- Cover scalability, reliability, CI/CD, logging and monitoring.

- Optionally, mention extra features like real-time notifications or analytics, Disaster recovery and failure alerts.

1. Backend (Server-side)

- **Framework:** Spring Boot
- **Responsibilities:**
 - Manage **Users, Restaurants, Menu Items, Orders, Delivery Agents.**
 - Business logic like calculating delivery ETA, order status updates, and payment processing.
 - Authentication & authorization for customers, restaurants, and admins.
- **Architecture:**
 - Controller > Service > Repository layers.
 - DTOs for requests/responses.
- **Best Practices:**
 - Implement Pagination for restaurant listings.
 - Have proper status codes for order creation, cancellation, etc.
 - Handling Exceptions with @ControllerAdvice.

2. Frontend (Client-side)

- **Framework:** React or Angular for web; React Native or Flutter for mobile.
- **Responsibilities:**
 - Show restaurant menus, allow searching/filtering, place orders.
 - Customer dashboard to track orders.
 - Restaurant dashboard to manage menu and order queue.
- **Integration:**
 - Communicate with backend via REST APIs or GraphQL.
- **UX Considerations:**
 - Lazy loading restaurant images and menu items.

3. Database (Storage Layer)

- **Choice:** SQL (PostgreSQL/MySQL)
- **Why SQL?**
 - Structured data: Users, Orders, Menu, Restaurants.
 - ACID compliance for payments and order status.
 - Relationships:

- User > Orders (One-to-Many)
 - Restaurant > Menu Items (One-to-Many)
 - Order > Menu Items (Many-to-Many)
- **Optional NoSQL:** MongoDB for storing analytics, logs, or user activity streams.

4. Cloud Deployment

- **Provider:** AWS / GCP / Azure
- **Services:**
 - Backend: Spring Boot on Kubernetes / App Engine.
 - Frontend: React hosted on Kubernetes.
 - Database: Managed RDS (SQL) or Cloud SQL.
- **Benefits:** Scalability, automatic backups, monitoring, and resilience.

5. Containerization

- **Docker** for backend, frontend, and worker services (like sending notifications).
- **Kubernetes** to manipulate multiple services and scale dynamically.

6. CI/CD Pipeline

- **Tools:** GitHub Actions, Jenkins,
- **Pipeline:**
 1. Code pushed > run unit tests.
 2. Build Docker images.
 3. Push images to container registry.
 4. Deploy to staging > run integration tests.
 5. Promote to production automatically or manually.

7. DevOps & Monitoring

- Logging: Cloud logging.
- Metrics & Monitoring: Prometheus + Grafana or Cloud monitoring.
- Alerts: Email, Slack, or SMS on errors.
- Backup: Have backup plans for Data.

8. Optional Extras for Scalability

- **Message Queues:** Kafka or RabbitMQ for handling notifications, order processing, and payment updates asynchronously.
- **Caching:** Redis or Memcached for popular restaurants, menu items, or user sessions.

462 Spring Scenario 51: Measuring Execution Time

Point to Note: Before going through the Scenarios on AOP, first go through AOP questions. Refer question no. 342 -344 Spring 90-92

Scenario: Your API performance is degrading, and you want to know which method is slow. Question: How can you capture execution time automatically?

Solution:

Spring AOP gives us a cleaner way: write the logic once, and apply it across multiple methods using an **aspect**.

We need to use `@Around`, it defines an advice that executes both before and after a matched method execution (join point)

```
@Aspect
@Component
public class ExecutionTimeAspect {

    @Around("execution(* com.example.service..*(..))")
    public Object measureTime(ProceedingJoinPoint joinPoint) throws Throwable {
        long start = System.currentTimeMillis();

        Object result = joinPoint.proceed(); // run the target method

        long end = System.currentTimeMillis();
        long duration = end - start;

        System.out.println(joinPoint.getSignature() + " executed in " + duration +
"ms");

        return result;
    }
}
```

The pointcut expression `execution(* com.example.service..*(..))` says:

apply this advice to any method inside the `com.example.service` package (and subpackages).

`joinPoint.proceed()` is where the original method actually executes.

```
@Service
public class OrderService {

    public void placeOrder() throws InterruptedException {
        Thread.sleep(500); // simulate slow DB call
        System.out.println("Order placed!");
    }
}
```

When we call `orderService.placeOrder()`, the aspect intercepts it, and we will get the output:

```
void com.example.service.OrderService.placeOrder() executed in 501ms
```

463 Spring Scenario 52: Centralized Exception Handling

Scenario: You want to log or take action whenever any service methods throws an exception Question:
How would you do it?

Solution:

We can use Spring AOP's `@AfterThrowing` advice. This advice runs only if a matched method throws an exception, which makes it perfect for centralized exception logging, monitoring, or even sending alerts.

```
@Aspect
@Component
public class ExceptionLoggingAspect {

    private static final Logger logger =
LoggerFactory.getLogger(ExceptionLoggingAspect.class);

    // Pointcut: match all methods in service package
    @Pointcut("execution(* com.example.service..*(..))")
    public void serviceMethods() {}

    // Advice: runs only when exception is thrown
    @AfterThrowing(pointcut = "serviceMethods()", throwing = "ex")
    public void logServiceException(Exception ex) {
        logger.error("Exception caught in service method: {}", ex.getMessage(), ex);
        // Here we can also add custom monitoring/alerting logic
    }
}
```

464 Spring Scenario 53: Transaction Management

Scenario: You want all service methods annotated with `@Transactional` to roll back on failure.

Question: How AOP helps here?

Solution:

When Spring sees `@Transactional`, it doesn't change the method code directly. Instead, it creates an AOP proxy around that bean. This proxy intercepts method calls, starts a transaction before the actual method runs, and then decides what to do afterwards, means it commit if everything goes fine, or roll back if an exception occurs.

So, we write plain business logic in the service methods, while the transactional concerns (`begin`, `commit`, `rollback`) are handled transparently by the AOP proxy.

465 Spring Scenario 54: Validate Tenant for all service methods

Scenario: Every service method should validate a tenant ID for multi-tenant architecture.

Question: How do you enforce it without duplicating code in every method?

Solution:

If we add tenant checks manually in every service method, we end up duplicating code and developers will forget it in some place. That is exactly the kind of cross-cutting concern AOP is meant to solve.

Custom Annotation:

```
@Aspect
@Component
public class TenantValidationAspect {

    @Before("within(@org.springframework.stereotype.Service *)")
    public void validateTenant() {
        String tenantId = TenantContext.getTenantId(); //Read tenant from Context

        if (tenantId == null || tenantId.isBlank()) {
            throw new IllegalStateException("No tenant ID found in context!");
        }

        System.out.println("Validated tenant: " + tenantId);
    }
}
```

The @Before advice runs before any service method is executed. If the tenant is missing, it blocks the call.

466 Spring Scenario 55: Custom Annotations

Scenario: You need to mark methods with @TrackExecution and only those should be logged.

Question: How would you achieve this with AOP??

Solution: Create custom annotation

Step 1: Create the Annotation

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TrackExecution { }
```

@Target(ElementType.METHOD): only valid on methods

@Retention(RetentionPolicy.RUNTIME): must be available at runtime for AOP to pick it up

@Aspect:

```
@Aspect
@Component
public class ExecutionTrackerAspect { }
```

```
private static final Logger Log =  
LoggerFactory.getLogger(ExecutionTrackerAspect.class);  
  
@Around("@annotation(com.example.TrackExecution)")  
public Object logExecutionTime(ProceedingJoinPoint pjp) throws Throwable {  
    long start = System.currentTimeMillis();  
  
    Object result = pjp.proceed();  
  
    long duration = System.currentTimeMillis() - start;  
    Log.info("Method {} executed in {} ms",  
            pjp.getSignature().toShortString(),  
            duration);  
  
    return result;  
}  
}
```

`@Around("@annotation(com.example.TrackExecution)")` tells Spring to only execute when a method is annotated with `@TrackExecution`

467 Spring Scenario 56: Load huge data where pagination is not possible

Here we need all the data at once. And it takes multiple tables to join and fetch the data. how you implement it and how will you reduce the latency?

Solution:

Backend aggregation

Don't push raw massive datasets to the frontend. Instead, fetch and aggregate them on the backend . Return the graph-ready format (series, categories, values).

Use optimized queries

Write a single optimized SQL with proper joins instead of firing multiple queries. Ensure you only fetch the required columns.

Materialized views / pre-aggregation

For very large datasets, create materialized views.

Database level

- Add proper indexes on join keys and filter columns.
 - Partition tables if data is huge (time-based partitioning is common).
 - Avoid SELECT *. Only fetch what we need for the graph.

Caching

- Cache heavy results (in Redis or application cache).

- If the data changes slowly, you don't need to hit the DB every time.

Compression & serialization

- Compress large JSON responses (gzip).

Unit and Integration Testing

468 Testing: Questions on Testing

1. What are different types of testing you do as developer?

Unit Testing: Testing individual classes or methods in isolation. Example: testing a service method that calculates discount.

Integration Testing: Testing how multiple components work together. Example: making sure the Spring Service talks correctly to a Repository and database.

Functional/End-to-End Testing: Testing the actual flow from API layer to DB. Example: hitting a REST endpoint and checking the full response.

2. What is Unit Testing and how would write Junit testcases?

Unit testing means testing a small piece of code (like one method) without external systems (DB, network).

With **JUnit 5** and **Mockito** (Mockito is a popular open-source mocking framework for Java used in automated unit tests), a simple test looks like this:

```
//Service we want to test
public class CalculatorService {
    public int add(int a, int b) {
        return a + b;
    }
}

//Test Class
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class CalculatorServiceTest {
    private final CalculatorService calculator = new CalculatorService();

    @Test
    void testAdd() {
        int result = calculator.add(2, 3);
    }
}
```

```

        assertEquals(5, result); // expected = 5
    }
}

```

Key points:

- Use `@Test` annotation to mark test methods.
- Assertions like `assertEquals`, `assertTrue`, etc., check the result.
- For Spring components, we often mock dependencies using **Mockito**.

3. What is Integration Testing and what tools you have used?

Integration testing checks how layers work together, like Controller > Service > Repository > Database.

In Spring Boot, we often use:

- **`@SpringBootTest`**: loads full application context.
- **`@DataJpaTest`**: tests JPA layer with an in-memory DB like H2.
- **`MockMvc`**: to test REST APIs.

```

@SpringBootTest
@AutoConfigureMockMvc
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void test GetUser() throws Exception {
        mockMvc.perform(get("/users/1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.name").value("John"));
    }
}

```

4. What is difference TDD and BDD?

TDD: Test Driven Development is an iterative development process where tests for new functionality are written *before* the actual code. The standard TDD cycle is often summarized as “Red-Green-Refactor.”

1. **Red:** Write a failing test. This ensures that the test is valid.
2. **Green:** Write just enough code to make the test pass.
3. **Refactor:** Optimize the code, ensuring that the test still passes after refactoring.

BDD (Behaviour-Driven Development)

- Extension of TDD, but focuses on behaviour of the system.
- Uses natural language (Given, When, Then).
- Example with Cucumber in Java:

```
Scenario: Add two numbers
Given I have a calculator
When I add 2 and 3
Then the result should be 5
```

5. Explain the lifecycle of a JUnit test.

Lifecycle includes:

1. @BeforeAll (setup for all tests, runs once).
2. @BeforeEach (setup before each test).
3. Test methods (@Test).
4. @AfterEach (cleanup after each test).
5. @AfterAll (cleanup for all tests, runs once).

6. What is @Test annotation and what its difference in JUNIT4 vs JUNITS

@Test tells the JUnit framework that a particular method should be treated as a test and executed when tests are run

In JUnit 4, @Test came from the org.junit package. Test methods had to be declared as public, and if we wanted to check for an exception, we used a special syntax inside the annotation itself

```
@Test(expected = IllegalArgumentException.class)
public void shouldThrowException() {
    // code that throws exception
}
```

JUnit 5 changed the experience by introducing a new architecture called Jupiter. The @Test annotation moved to org.junit.jupiter.api, and we no longer had to make them public. Exception handling became cleaner too, because instead of writing expected exceptions in the annotation, we could simply use an assertion like assertThrows, which made the test more readable

```
@Test
void shouldThrowException() {
```

```
assertThrows(IllegalArgumentException.class, () -> {
    // code that throws exception
});
}
```

7. How do you approach testing in Spring boot applications

- We usually begin with unit tests. These tests focus on individual classes, like a service method or a utility function. They don't load the full Spring context, and dependencies are mocked.
- REST APIs can be tested in two main ways. If we want to test only the controller , we use @WebMvcTest, which loads just the web layer.
- If we want to test the whole flow like controller, service, repository, and even database then we use @SpringBootTest combined with MockMvc.
- For database-specific testing, Spring Boot gives @DataJpaTest, which brings up an in-memory database like H2 so that we can test the JPA repositories without touching a real database.

8. How do you perform unit testing in Spring Boot?

Performing unit testing in Spring Boot usually involves JUnit 5 and Mockito. The idea is to test only one class at a time while mocking out its dependencies.

Suppose we have a UserService that depends on a UserRepository. In a unit test, we don't connect to the real database; instead, we tell Mockito to return fake results when the repository is called. That way, we can verify if the service logic works as expected.

```
@SpringBootTest
class UserServiceTest {

    @MockBean
    private UserRepository userRepository;

    @Autowired
    private UserService userService;

    @Test
    void shouldReturnUserById() {
        when(userRepository.findById(1L)).thenReturn(Optional.of(new User(1L, "John")));

        User user = userService.getUser(1L);

        assertEquals("John", user.getName());
    }
}
```

9. Which starter package do you need to test the spring boot application?

We simply add spring-boot-starter-test in the pom.xml, and we get access to all the popular testing libraries without having to add them one by one.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

10. What dependencies does the spring-boot-starter-test brings to the class-path?

- JUnit as the main testing framework
- Mockito for mocking dependencies,
- AssertJ for fluent and human-readable assertions
- Hamcrest for matcher-based assertions.
- For applications that deal with JSON and XML, JsonPath and XmlUnit are also included.
- Spring Test itself is part of the package, enabling integration with the Spring Boot test framework.

11. How do you perform Integration testing in Spring boot?

Integration testing in Spring Boot is all about verifying how different layers of the application work together.

For example, suppose we have a UserController that talks to a UserService, which in turn talks to a UserRepository. An integration test makes sure that the flow works correctly end to end. In Spring Boot, the simplest way to do this is with @SpringBootTest. This annotation starts the entire application context and allows us to send HTTP requests to the controllers using MockMvc or TestRestTemplate.

```
@SpringBootTest // Loads the full application context
@AutoConfigureMockMvc // Configures and injects MockMvc
class UserControllerIT {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void shouldReturnUserById() throws Exception {
        mockMvc.perform(get("/users/1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.name").value("John"));
    }
}
```

12. How is @ContextConfiguration used in Spring Boot?

@ContextConfiguration comes from the older Spring testing framework. It is used to specify which configuration classes or XML files should be loaded to create the application context for a test. It is used for configuring the ApplicationContext for integration tests, For instance, before Spring Boot, we would write something like this:

```
@ContextConfiguration(classes = AppConfig.class)
class LegacySpringTest {
}
```

In Spring Boot, we rarely need @ContextConfiguration directly because @SpringBootTest usually handles context loading. However, we might still see it in cases where we want fine-grained control over which beans are loaded, or if we are testing a very specific slice of the application without starting the full Boot context.

13. What does @SpringBootTest do? How does it interact with @SpringBootApplication and @SpringBootConfiguration?

When we annotate a test class with @SpringBootTest, we are telling Spring Boot to start up the full application context, almost as if we run the application with @SpringBootApplication. That means all the beans, properties, and auto-configurations are loaded.

Under the hood, @SpringBootTest builds on top of @SpringBootConfiguration and @SpringBootApplication. While @SpringBootApplication is what we put on the main application class to bootstrap the app, @SpringBootTest finds that class and uses it to initialize the context in the test environment

14. What is the difference between @ContextConfiguration and @SpringBootTest?

@ContextConfiguration belongs to the core Spring framework and is mainly used to load a minimal or custom context, typically pointing to specific configuration classes. It doesn't know about Spring Boot's auto-configuration or starters.

@SpringBootTest, on the other hand, is Boot related. It not only loads the application's configuration but also applies all the auto-configuration and Boot-specific features.

15. How can you test Spring Boot REST controllers?

If we only want to test the controller layer in isolation, without loading services or repositories, we use @WebMvcTest. This annotation creates a minimal context that includes only the web components. Dependencies like services can be mocked with @MockBean.

```
@WebMvcTest(UserController.class)
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;
```

```

@MockBean
private UserService userService;

@Test
void shouldReturnUserById() throws Exception {
    when(userService.getUser(1L)).thenReturn(new User(1L, "John"));

    mockMvc.perform(get("/users/1"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.name").value("John"));
}
}

```

If we prefer to test the full flow, including database calls, then we use `@SpringBootTest` with `MockMvc` or `TestRestTemplate`. This way we are not mocking the service or repository; instead, we are checking if everything works together.

```

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class UserControllerFullStackTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void shouldReturnUserById() {
        User user = restTemplate.getForObject("/users/1", User.class);
        assertEquals("John", user.getName());
    }
}

```

16. `@AutoConfigureMockMvc` vs `@WebMvcTest`

When we are testing REST controllers in Spring Boot, two annotations often come up: `@AutoConfigureMockMvc` and `@WebMvcTest`. At first glance, they might look like they serve the same purpose, but they actually live at different levels.

`@WebMvcTest` only work on Web layer like controllers, JSON converters, filters, and so on. Nothing else is loaded. Services and repositories don't exist in this slice unless we explicitly mock them. This is perfect when the goal is to check whether a controller maps endpoints correctly, validates input, or serializes output.

```

@WebMvcTest(UserController.class)
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean

```

```

private UserService userService;

@Test
void shouldReturnUserById() throws Exception {
    when(userService.getUser(1L)).thenReturn(new User(1L, "John"));

    mockMvc.perform(get("/users/1"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.name").value("John"));
}

}

```

Notice what happens here: the web context is bootstrapped, but UserService doesn't exist until we create a mock with @MockBean. That is because @WebMvcTest doesn't load the service or repository layer at all.

On the other side, @AutoConfigureMockMvc comes into play when are already using @SpringBootTest. While @SpringBootTest starts the full application context (controllers, services, repositories, database connections), adding @AutoConfigureMockMvc wires up the MockMvc bean so that we can test our REST endpoints without starting a real HTTP server. In other words, we are testing the controller in the context of the whole application.

Example:

```

@SpringBootTest
@AutoConfigureMockMvc
class UserControllerIT {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void shouldReturnUserFromDatabase() throws Exception {
        mockMvc.perform(get("/users/1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.name").value("John"));
    }
}

```

In this setup we don't create fake services or repositories. The test uses the real database (often an in-memory one like H2 or sometimes Testcontainers), runs the actual service code, and then calls the controller.

17. How do you test @Repository?

Repositories in Spring are gateway to the database. They handle queries, inserts, updates, and deletes. Testing them isn't about mocking, because mocking defeats the purpose, we want to be sure that the queries we write actually work against a real database.

The usual way is to use an **in-memory database** like H2 or a lightweight containerized database (with Testcontainers). Spring Boot makes this super easy.

```
@DataJpaTest
class UserRepositoryTest {

    @Autowired
    private UserRepository userRepository;

    @Test
    void shouldFindUserByEmail() {
        User saved = userRepository.save(new User("macha@test.com"));
        User found = userRepository.findByEmail("macha@test.com");

        assertThat(found.getEmail()).isEqualTo(saved.getEmail());
    }
}
```

Here, we didn't spin up the whole application. Just the JPA part and an in-memory DB. This test checks if the query method in the repository is actually returning what we expect.

18. When do you want `@DataJpaTest` for? What does it auto-configure?

We use `@DataJpaTest` for writing tests for repositories or JPA logic. It auto-configures everything we need for JPA:

- An in-memory database (by default, H2).
- Scans only for `@Entity` classes and Spring Data JPA repositories.
- Configures Hibernate and JPA mappings.
- Rolls back each test by default, so the DB stays clean.

So instead of loading the entire Spring context, You are just testing the persistence layer in isolation. It is lightweight and fast.

19. How would write unit testcases for services?

Services are where our business logic lives. Unlike repositories, here we don't want a real DB most of the time. Instead, we mock the repository layer and just test the logic.

```
class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
}
```

```

void shouldReturnUserWhenEmailExists() {
    when(userRepository.findByEmail("test@mail.com"))
        .thenReturn(new User("test@mail.com"));

    User result = userService.findByEmail("test@mail.com");

    assertThat(result.getEmail()).isEqualTo("test@mail.com");
}
}

```

Here, we don't test how the repository works (that is tested elsewhere). we only test if the service is working correctly or not.

20. What is Mockito?

Mockito is a Java testing framework that helps to create mocks. A mock is a fake object that simulate the behaviour of real objects, so we can test our class without worrying about dependencies.

Imagine the service depends on a repository. Instead of hitting the real DB, Mockito gives a fake repository. We tell it how to behave (like "if someone calls findByEmail, return this dummy user"). That way, we test only the logic in the service, not the repository itself.

21. How do you mock external services / external APIs?

When the code talks to the outside world like a payment gateway, weather API, or an email service, we don't want our tests to actually call them. That would be slow and might even cost money. Instead

In Spring Boot, we usually wrap the external API in a **service class** (say, WeatherClient). Then in your tests, we mock that client.

```

class WeatherServiceTest {

    @Mock
    private WeatherClient weatherClient;

    @InjectMocks
    private WeatherService weatherService;

    @Test
    void shouldReturnSunnyWeather() {
        when(weatherClient.getWeather("Hyderabad"))
            .thenReturn("Sunny");

        String result = weatherService.fetchWeather("Hyderabad");

        assertThat(result).isEqualTo("Sunny");
    }
}

```

```
    }
```

The actual HTTP call never happens. We just “pretend” the client gave a response. This keeps our tests fast and focused only on how our code handles the response.

If we want to test the whole HTTP layer itself, we can also use tools like **WireMock** or **MockWebServer** (These act like fake servers that the code hits during the test.)

22. What is Spy annotation

The `@Spy` annotation in Mockito is used to create a spy instance of a real object. Spies are useful to perform partial mocking, where some methods of the object are mocked while others retain their original behavior. This is particularly useful when we want to test a class with some real method calls and some mocked method calls.

Example:

Let's say we have calculator class.

```
class Calculator {  
    int add(int a, int b) { return a + b; }  
    int multiply(int a, int b) { return a * b; }  
}
```

Test class with `@Spy`.

```
class CalculatorTest {  
  
    @Spy  
    private Calculator calculator;  
  
    @Test  
    void spyExample() {  
        // Real add method will be called  
        assertThat(calculator.add(2, 3)).isEqualTo(5);  
  
        // Mock multiply to return a fake value  
        doReturn(100).when(calculator).multiply(2, 3);  
  
        assertThat(calculator.multiply(2, 3)).isEqualTo(100);  
    }  
}
```

Reference: [Mockito @Spy Annotation Tutorial](#)

23. What are the differences between @MockBean and @Mock annotations?

Both create mocks, but they are different.

- @Mock is from Mockito. It is used in **unit tests**, outside the Spring context. Its a plain JUnit + Mockito.
- @MockBean is from Spring Boot Test. It replaces a bean in the **Spring ApplicationContext** with a mock.

For example, if we are testing a controller with @WebMvcTest:

```
@WebMvcTest(UserController.class)
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService; // replaces bean in context

    @Test
    void shouldReturnUser() throws Exception {
        when(userService.findByEmail("test@mail.com"))
            .thenReturn(new User("test@mail.com"));

        mockMvc.perform(get("/users/email/test@mail.com"))
            .andExpect(status().isOk());
    }
}
```

@MockBean tells Spring: “Don’t load the real UserService, use this fake one instead.”

If we use @Mock, the Spring context won’t know about it, and the test would fail because the controller still expects a bean.

24. How do you Mock the beans in Spring boot?

The easiest way is exactly what we saw above: @MockBean.

Whenever the Spring-managed class depends on another bean, and we don’t want the real one, we just replace it with @MockBean.

25. @InjectMock vs @Mock

When we start writing tests with Mockito, these two annotations often sit next to each other. At first, they look similar, but they actually serve two very different purposes.

Let’s take an example. Imagine we have an OrderService that depends on a PaymentService:

```
class PaymentService {
    public String process() {
```

```

        return "real payment done";
    }

}

class OrderService {
    private final PaymentService paymentService;

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

    public String placeOrder() {
        return paymentService.process();
    }
}

```

Now, suppose we want to test OrderService. You don't really care about whether money is transferred , we just want to check that OrderService behaves correctly when the payment goes through. This is where `@Mock` and `@InjectMocks` come in.

@Mock: When we put `@Mock` on a class, Mockito creates a fake version of it. None of the real logic runs unless we tell it to. Instead, we control its behaviour in the test.

```

@Mock
private PaymentService paymentService;

```

Here, `paymentService` is no longer real. It is just a dummy object

@InjectMocks: Now comes `@InjectMocks`. This tells Mockito: "*Create the real object of this class, but if it has dependencies, inject the mocks into it.*"

```

@InjectMocks
private OrderService orderService;

```

Mockito sees that `OrderService` needs a `PaymentService`, and since we already created a mock of it, it wires the fake into the real `OrderService`.

Complete sample:

```

class OrderServiceTest {

    @Mock
    private PaymentService paymentService; // fake dependency

    @InjectMocks
    private OrderService orderService;      // real class, mock injected

    @Test
}

```

```
void shouldReturnApprovedWhenPaymentSucceeds() {
    when(paymentService.process()).thenReturn("Approved");

    String result = orderService.placeOrder();
    assertThat(result).isEqualTo("Approved");
}
```

In this test, OrderService is real, but the PaymentService inside it is fake. That is exactly what we want in a unit test: test one class in isolation, while replacing its dependencies with mocks.

When to use each

- Use **@Mock** whenever we want to create a fake version of a class and fully control its behaviour in tests. We don't want the real logic at all.
- Use **@InjectMocks** when we are testing a class that has dependencies. It creates the real object and injects the mocks automatically, so we don't need to manually set up the class.

Coding Questions for Practice

469 Array Coding Problems

1. Reverse the array

Given `int[] arr = {2,5,6,4,1,3}; reverse: [3, 1, 4, 6, 5, 2]`

2. Find Second Smallest and Second Largest element in an array

Given `int[] arr = {2, 5, 6, 4, 1, 3, 10, 8, 7}; Second largest is 8 and second smallest is 2`

Code link: <https://github.com/CodingLyf-Fullstack/Arrays/edit/main/SecondSmallestAndLargest.java>

3. Find the repeating elements / duplicates from an array

Given `int[] arr = { 2, 5, 6, 2, 1, 3, 1, 8, 3 };` In this array {2,1,3} are duplicates

Write a function that should return {2,1,3}

Code link: <https://github.com/CodingLyf-Fullstack/Arrays/edit/main/FindDuplicatesInArray.java>

4. Given two arrays, merge them into a single sorted array.

Input: `int arr1[] = {2,3,5,4,1}; int arr2[] = {8,7,6,10,9};`

Excepted output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Code link: [Arrays/MergeArraysIntoSingleSortedArray.java at main · CodingLyf-Fullstack/Arrays](#)

5. Count of number of occurrences of each number in an array.

Given `int arr[] = {9, 8, 4, 9, 5, 4, 9, 1}`. Here 9 repeated 3 times and 4 repeated 2 times etc.
Expected output: {1=1, 4=2, 5=1, 8=1, 9=3}

Code Link: [Arrays/CountNumberofOccurrences.java at main · CodingLyf-Fullstack/Arrays](#)

6. Find all non-repeating or unique elements in an array.

Given `int[] arr = {4, 5, 2, 4, 2, 1}`; 5 and 1 and not repeated.

Expected output : {5,1}.

7. Check if Array is a subset of another array or not

Input1:

`int[] arr1 = {1, 2, 3, 4, 5};`

`int[] arr2 = {2, 4, 5};`

Here 2,4,5 are subset of arr1. Function should return true

Input1:

`int[] arr1 = {1, 2, 3, 4, 5};`

`int[] arr2 = {2, 6};`

Here 2,6 are subset of arr1. Function should return false

Code Link: [Editing Arrays/FindArrayisSubset.java at main · CodingLyf-Fullstack/Arrays](#)

8. Find majority element in an array.

A majority element in an array is an element that appears more than $n/2$ times, where n is the size of the array.

Given `int[] nums = { 2, 2, 1, 1, 1, 2, 2 }`; where 2 is majority element

Code Link: [Arrays/FindMajorityElement.java at main · CodingLyf-Fullstack/Arrays](#)

9. Move all zeroes to the end of the array

Given `int[] arr = { 0, 2, 3, 0, 0, 1, 4, 0, 0 }`; Expected output: {2, 3, 1, 4, 0, 0, 0, 0, 0}

Code link: [Arrays/MoveallZerostoEnd.java at main · CodingLyf-Fullstack/Arrays](#)

We can also use two pointer technique to solve this problem. Refer question no. 21

10. Find Leaders in an Array

A leader in an array is an element that is greater than all the elements to its right.

- The rightmost element is always a leader because there are no elements to its right.
- You need to check each element and see if it is bigger than all the elements that come after it in the array.

Input [16, 17, 4, 3, 5, 2] and the leaders are [2,5,17]

Code link: <https://github.com/CodingLyf-Fullstack/Arrays/edit/main/FindLeadersInArray.java>

11. Merge Two Sorted Arrays Using streams

Given `int[] arr1 = { 2, 4, 6, 7, 9 }; int[] arr2 = { 5, 8, 10, 12 };`

Hint: Use Streams.concat() then .sorted()

12. Find Subarray with Given Sum

Given an array of integers and a target sum: `int arr[] = {1, 2, 3, 7, 5}; int target = 12.`

The task is to find a continuous part of the array (subarray) whose elements add up exactly to the given sum.

Important: Subarray means the elements must be contiguous (next to each other in the array). It is not just picking any random elements.

Key point: We need use Sliding window with variable window size.

Reference: [Sliding Window in 7 minutes | LeetCode Pattern](#)

Code link: [Editing Arrays/FindSubArrayWithGivenSum.java at main · CodingLyf-Fullstack/Arrays](#)

13. Find number of subarrays with Sum.

In this problem, we need to find number of sub array, in the above only one sub array will be there. But we can have multiple sub arrays .

Given an array of integers and a target sum: `int arr[] = {1,2,2,3,1}; int target = 5.`

We can see two sub arrays [1,2,2], [2,3]. So the output should be 2.

Code link: [Editing Arrays/FindNumberofSubArrayswithGivenSum.java at main · CodingLyf-Fullstack/Arrays](#)

14. Find Maximum Subarray Product

Given an integer array, find the contiguous subarray (containing at least one number) which has the largest product.

Input: `int[] arr = { 2, 3, -2, 4 }`, Expected output is 6 which maximum product of a sub array.

Code Link:

Similar question: Find Maximum Subarray Sum (Kadane's algorithm)

Code link: [Arrays/MaxSubarrayProduct.java at main · CodingLyf-Fullstack/Arrays](#)

15. Find Maximum sub array whose length is K

Key point: We need use Sliding window with fixed window size(K).

Reference: [Sliding Window in 7 minutes | LeetCode Pattern](#)

Given an array `int[] arr = {2, 1, 5, 1, 3, 2}`; and `int k = 3`; Means. Given a fixed sized. We need to get maximum sum of a sub array with length 3.

Example:

List all subarrays of length 3

- [2, 1, 5] > sum = 8
- [1, 5, 1] > sum = 7
- [5, 1, 3] > sum = 9 (Maximum)
- [1, 3, 2] > sum = 6

Code Link: [Arrays/MaxSubArrayWhoseLengthisK.java at main · CodingLyf-Fullstack/Arrays](#)

16. Sort the Map by values.

Suppose you have a `HashMap<String, Integer>` that stores names and their scores, and you need to sort it by values.

```
Map<String, Integer> scores = new HashMap<>();
scores.put("Mahesh ", 45);
scores.put("NTR", 30);
scores.put("Chiru", 75);
scores.put("Balu", 60);
```

Code link: [Arrays/SortMapByValues.java at main · CodingLyf-Fullstack/Arrays](#)

Try to solve the same problem Java 8

17. Rotate the Array by K steps.

Given an array `int arr[] = { 1, 2, 3, 4, 5, 6, 7 }` and an integer `k = 3`. Rotate the array to the right by k steps.

That means, each element moves k steps to the right, and if it goes past the end, it comes back around to the start.

Example:

Input int arr = [1, 2, 3, 4, 5, 6, 7] and k = 3

Ouput [5, 6, 7, 1, 2, 3, 4]

Code link: [Arrays/RotateArraybyKSteps.java at main · CodingLyf-Fullstack/Arrays](#)

18. Move all zeros to the left.

In question 9 we have seen moving all zeros to the end of the array, now it is move to the left means start of an array

Given **int[]** arr = { 0, 2, 3, 0, 0, 1, 4, 0, 0 }; Expected output: {0, 0, 0, 0, 0, 2, 3, 1, 4}

Code link: [Arrays/MoveAllZerosToLeft.java at main · CodingLyf-Fullstack/Arrays](#)

Using Two pointer: [Arrays/MoveZerosToLeftUsingTwoPointer.java at main · CodingLyf-Fullstack/Arrays](#)

19. Top K frequent elements

Given an integer array nums and an integer k, return the k most frequent elements. You may return the answer in any order.

Input: nums = [1,1,1,2,2,3], k = 2

Output: [1,2]

Code link: [Arrays/TopKFrequentElements.java at main · CodingLyf-Fullstack/Arrays](#)

20. Check if a Pair with Given Sum Exists in Array (Two Sum Problem)

You are given an array of integers and a target sum. You need to check if there exists any pair of numbers in the array whose sum is equal to the target.

Input: arr = [8, 4, 1, 6], target = 10

Output: true because there is pair 4 and 6 whose sum is 10

Code Link: [Arrays/CheckPairWithGivenSumExistsInArray.java at main · CodingLyf-Fullstack/Arrays](#)

Additional Question: Same question can be asked to the pair instead of true/false.

Hint: **return new int[]{numToIndex.get(complement), i};**

Note: we can also solve with two pointer technique, but we need to sort array (Refer Question 21)

21. Triplet with Zero Sum – 3 Sum Problem

Given an **unsorted array**, check if there exists a triplet (a, b, c) such that a + b + c = 0.

Input: arr = [-1, 0, 1, 2, -1, -4]; there are two triplets with sum 0 [(-1, -1, 2), (-1, 0, 1)]. So our function should return true.

Approach:

Here we use **Two pointer technique**. Reference to know more about two pointer: [Two Pointers in 7 minutes | LeetCode Pattern](#)

Code link: [Arrays/TripletWithSumZero.java at main · CodingLyf-Fullstack/Arrays](#)

22. Remove Duplicates from Sorted Array – Two pointer

You are given a sorted array. You need to remove duplicates *in-place* so that each unique element appears only once.

Input: arr = [1, 1, 2] **output** [1,2]

Code link: [Arrays/RemoveDuplicates.java at main · CodingLyf-Fullstack/Arrays](#)

23: Longest Consecutive Sequence

Given an unsorted array of integers nums, return the length of the longest consecutive elements sequence.

Input: nums = [100,4,200,1,3,2]

Output: 4

Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore, its length is 4.

Code link: [Arrays/LongestConsecutiveSequence.java at main · CodingLyf-Fullstack/Arrays](#)

24. Create a function that takes input as 7 and returns 11. if we give input 11, it should return 7. Don't use if-else conditions.

```
public class Main {

    public static void main(String[] args) {
        System.out.println(toggle(7));
        System.out.println(toggle(11));
    }

    public static int toggle(int n) {
        return 7 + 11 - n;
    }
}
```

25. Print all subset of a given array – Recursion

You are given an array, and you want to **print all possible subsets**.

- Subsets can be of any size (0 to n).

- Example: for [1,2] and subsets are: [], [1], [2], [1,2].

Code Link [Arrays/SubsetsRecursive.java at main · CodingLyf-Fullstack/Arrays](#)

Additional problems for practice

26. Sum of natural numbers using recursion

27. Factorial of a number using recursion

28. Find out the Sum of Digits of a Number.

29. Find Common elements between two arrays int[] arr1 = {1, 2, 4, 5, 6}; int[] arr2 = {2, 3, 5, 7};

30. Add and subtract two matrices

470 String Coding Problems

1. Check two strings anagrams or not

An anagram means both strings contain the same characters with the same frequency, but possibly in a different order.

Example:

"silent" and "listen" are anagrams.

"hello" and "world" are not.

Code link: [Strings/CheckAnagrams.java at main · CodingLyf-Fullstack/Strings](#)

2. Check if String has all unique characters

Input: "silent"

- Characters: s, i, l, e, n, t
- All are different return **true**

Input: "hello"

- Characters: h, e, l, l, o
- l repeats return **false**

Code Link: [Strings/CheckAllUniqueChars.java at main · CodingLyf-Fullstack/Strings](#)

3. Find all Unique Characters

Given a String you need to fetch all the all-unique characters means characters that appear only once.

Input: "success", Unique characters are u, e.

Code Link: [Strings/FindAllUniqueCharacters.java at main · CodingLyf-Fullstack/Strings](#)

4. Find the First Non-Repeating Character

Input: "swiss", First non-repeating character is w.

Code Link: [Strings/FirstNonRepeatingCharacter.java at main · CodingLyf-Fullstack/Strings](#)

5.String to Camel Case

Input: "hello from coding lyf"; output: "Hello From Coding Lyf".

Code Link: [Strings/StringtoCamelCase.java at main · CodingLyf-Fullstack/Strings](#)

6. Check if one String is rotation of another String in java

You need to check if one string is a **rotation** of another.

Example:

- s1 = "abcd"
- s2 = "cdab"

Here s2 is a rotation of s1 because if you rotate "abcd" left by 2, you get "cdab".

Code Link: [Strings/CheckStringRotation.java at main · CodingLyf-Fullstack/Strings](#)

7. Count Number of Words in String without using split().

Input: "This is coding lyf"; Output: 4

Code Link: [Strings/CountNumberOfWords.java at main · CodingLyf-Fullstack/Strings](#)

8. Remove duplicates from the String.

Given a String remove all the duplicate characters from the given String.**Example 1:**

Input: s = "bcabc"

Output: "bca"

Code Link: [Strings/RemoveDuplicates.java at main · CodingLyf-Fullstack/Strings](#)

9. Find the Max repeating character

Input: str = "apple"

Output: p

Explanation: The character 'p' have the maximum occurrence i.e 2.

Code Link: [Strings/FindMaxRepeatingCharacter.java at main · CodingLyf-Fullstack/Strings](#)

10. Create a new String by removing all occurrences of given character

Input: "banana", remove 'a' so output: bnn.

Code Link: [Strings/RemoveChars.java at main · CodingLyf-Fullstack/Strings](#)

11. Find the largest word in a given string

Input: "Java makes coding enjoyable and challenging"; Output: "challenging"

Code Link: [Strings/FindTheLargestWord.java at main · CodingLyf-Fullstack/Strings](#)

12: Remove characters from first string that are present in the second string

You are given two strings:

- **First string:** the base string
- **Second string:** contains characters that need to be removed from the first string

First string = "computer"

Second string = "cat"

Remove 'c' and 't' from first String

Code link: [Strings/RemoveCharactersFromFirstString.java at main · CodingLyf-Fullstack/Strings](#)

13. Implement a method to compare two Strings for equality (don't use .equals)

Code Link: [Strings/StringEqualityCheck.java at main · CodingLyf-Fullstack/Strings](#)

14. Isogram

An isogram is a word or phrase where no letter repeats. Every character appears only once.

Example 1:

Word: machine

- Letters: m, a, c, h, i, n, e
- None of them repeat > isogram.

Example 2:

Word: programming

- Letters: p, r, o, g, r, a, m, m, i, n, g
- Here r, m, and g repeat > Not an isogram.

Code Link: [Strings/Isogram.java at main · CodingLyf-Fullstack/Strings](#)

15. Replace all instances of a Substring with another sub string. – Sliding Window

```
String input = "abc123abc456abc";
```

```
String target = "abc";
```

```
String replacement = "XYZ";
```

Then all the “abc” should be replaced with “XYZ”

Code Link: [Strings/ReplaceAllInstances.java at main · CodingLyf-Fullstack/Strings](#)

16. Merge Two Strings Alternatively

Input: String s1 = "Coding"; String s2 = "Lyf";

Output: CLoydfng

Code Link: [Strings/MergeTwoStringsAlternatively.java at main · CodingLyf-Fullstack/Strings](#)

17. Check if a String is a subsequence of another string – Two pointer

A string s1 is a subsequence of string s2 if all characters of s1 appear in s2 **in the same order**, but not necessarily contiguously.

s1 = "abc", s2 = "axbycz" > subsequence, "abc" is a subsequence.

s1 = "abc", s2 = "acb" > Not subsequence, order breaks.

18. Count the number of times one string occurs in another - Sliding Window

Given two strings, s1 (the pattern) and s2 (the text), count how many times s1 occurs contiguously in s2.

- Contiguously means the characters of s1 appear together in order, without any gaps.

Example 1:

s1 = "abc"; s2 = "abcabcabc"; Output: 3

Explanation:

- "abc" appears starting at index 0, 3, and 6.
- Total occurrences = 3

Example 2:

s1 = "xyz"; s2 = "xyabcz"; Output: 0

Explanation:

- "xyz" is not there "xyabcz".

Code Link: [Strings/CheckSubsequence.java at main · CodingLyf-Fullstack/Strings](#)

19. Find the longest substring without repeating the characters – Siding Window

Given a string s, find the length of the longest substring and substring without duplicate characters.

Input: s = "pwwkew"

Output: 3

Explanation: The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Code Link: [Strings/LongestSubString.java at main · CodingLyf-Fullstack/Strings](#)

20. Find the longest common prefix

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

Example:

Input: strs = ["flower", "flow", "flight"]

Output: "fl"

Code Link: [Strings/LongestCommonPrefix.java at main · CodingLyf-Fullstack/Strings](#)

21. Reverse individual words

Given a string containing multiple words, reverse each word individually without changing their order in the sentence.

Input: "Hello World"

Output: "olleH dlroW"

Code Link: [Strings/ReverseIndividualWords.java at main · CodingLyf-Fullstack/Strings](#)

22. Reverse words in String

Given a string containing multiple words, reverse the order of the words, but keep the characters in each word intact.

- Input: "Hello World from Java"
- Output: "Java from World Hello"

Hint: Split the sting with space “ ”. Iterate the array and apply reverse logic.

23. Rotate String both left and right by K steps.

Rotate a string left or right by k positions.

- Left Rotation: Move the first k characters to the end.
- Right Rotation: Move the last k characters to the beginning.

Example:

Input String: "abcdef"

- Left rotation by 2: "cdefab"
 - "ab" moves from the start to the end.
- Right rotation by 2: "efabcd"
 - "ef" moves from the end to the start.

Code Link: [Strings/RotateStringLeftAndRight.java at main · CodingLyf-Fullstack/Strings](#)

Note: You can also do with substring. Give a try.

24. String Compression

Given a string, compress it by replacing consecutive repeated characters with the character followed by its count.

- If the compressed string is not shorter than the original, you can keep the original.

Example 1:

Input: "aabbbc"

Compression Steps:

- 'a' repeats 3 times, "a3"
- 'b' repeats 2 time , "b2"
- 'c' repeats 1 time, "c"

Output: "a3b2c"

Original: "aabbbc": length 6

Compressed: "a3b2c": length 5

Code Link: [Strings/StringCompression.java at main · CodingLyf-Fullstack/Strings](#)

25: Find all permutations of a String – Recursive

Given String “ABC”, Print all the permutations ABC, ACB, BAC, BCA, CAB, CBA

Code Link: [Strings/FindAllPermutations.java at main · CodingLyf-Fullstack/Strings](#)

Reference: [L12. Print all Permutations of a String/Array | Recursion | Approach - 1](#)

Additional questions:

26. Write a program to sort characters in a string - Use any sorting technique like Bubble sort
27. Write a program to find a substring within a string. If found display its starting position - Sliding Window ()

Hint: Refer question 18. Write a function and instead of incrementing the count. return i and break the loop.

```
if(i == s1.length) > index not found  
else index found at i
```

28. Change case of each character in a string

29. String Palindrome

30. Reverse the String

471 Functional/Behavioural Questions

1. Explain your project folder structure
2. What is spring version you are using, what are major issues you have seen when updated from 2.x to 3.x
3. What are the third-party libraries you have used in your project?
4. What are the exceptions you got in your project?
5. How Agile model works in your project?
6. How do you start when you were assigned a task?
7. What if your product owner expecting a task in 1 sprint but it takes 2 sprints? how would you convey this?
8. How do you test before releasing it to QA?
9. How do you handle if you receive any bug from QA?
10. How would you debug the priority issue from the Prod?
11. Explain Architecture of your Application you are working.
12. What is the Java version you are using in your project? - Prepare the features of the version you are using
13. What is the Spring version you are using in your project? - Prepare the features of the version you are using

14. What is the complicated task you did recently?
15. What's the role of design patterns?
16. What are the design patterns you have used in your project?
17. Have you used SOLID Principles in your project?
17. What is code coverage and do you use sonar cube?
18. How build automation works in your project? Using Jenkins or GitHub actions?
19. How do you start working on some tasks after it was assigned to you?
20. How security is implemented and what checks you do in your project?
21. What is code review pattern you follow in your project?
22. What is your approach to improve the code quality?
23. what is your day-to-day task?
24. How will you ensure compatibility when upgrading the dependencies?
25. What are the testing tools you used in your project?
26. You started working on a task but after couple days new requirements are been added. How would you make sure to complete within the Sprint?
27. How do you learn new concepts or new things?
28. What is SDLC?
29. If you were assigned a task, how you proceed it, right from story points to hand over it to QA.
30. How do you make sure all the acceptance criteria are working fine.
31. How do you handle a situation where it takes 2 sprints but product owner is expecting to be finished in 1 sprint?

Questions by Experience Level

Please Note: These questions are segregated based on fundamental knowledge but in real time its highly depends on company, role and interview process. So please don't stick to these it is just for an idea.

472 Junior Level Developer

Core Java Fundamentals

1. Explain about JDK, JRE, and JVM.
2. What are the core features of Java that make it object-oriented?
3. Can you list all primitive data types in Java and their sizes?
4. How does autoboxing/unboxing work? Can you give an example where it might cause a bug?
5. Explain access modifiers (private, protected, public, default) with simple code.
6. What is a package in Java, and why do we use it?
7. How would you explain the static keyword (variables, methods, blocks)?
8. What is a constructor in Java? What are its types?

OOP Concepts

9. What is inheritance? Can you explain with real-world examples?
10. How do you explain polymorphism to a non-technical person? Give both compile-time and runtime examples.
11. What's the difference between abstraction and encapsulation?
12. Method overloading vs overriding, when do you use which?
13. Difference **interface** vs **abstract class**?
14. What is a Marker Interface and example?
15. Difference between final, finally, and finalize()?

Strings

16. What's the difference between String, StringBuilder, and StringBuffer?
17. Why is String immutable?
18. How does the String Pool work internally? Why do we need it?
19. Difference between creating a string using new and using a literal?
20. == vs .equals() ?

Collections

21. What's the difference between Array and ArrayList?
22. ArrayList vs LinkedList , which one would you choose for frequent insertions?
23. Explain HashMap and its common operations.
24. Difference between HashMap and TreeMap.
25. What's the difference between HashSet and TreeSet?
26. why use List over Set, or vice versa?
27. What is an Iterator? Can you show how to use it?
28. What's the difference between break and continue?

29. Fail safe vs Fail fast?
30. Difference between Vector and ArrayList.

Exception Handling

31. What is an exception? Checked vs unchecked?
32. Explain try-catch-finally. Can you use try without catch?
33. What is the super class of Exception?
34. What is the difference between throw and throws?
35. Why would you create a custom exception?
36. What are the exceptions you have got in your project? How do you prevent them?
37. Difference between Error and Exception.

Multithreading & Concurrency

38. Difference between Thread and Runnable.
39. Lifecycle of a thread in Java.
40. Difference between wait() and sleep().
41. What is the role of synchronized keyword?

Streams

42. What is the Stream API in Java 8?
43. Can you explain map, filter, and forEach with a small example?
44. Why are streams preferred over loops in some cases?
45. Differences Intermediate and terminal operators.
46. Practice the coding questions on filter, map, groupigBy, max, sort
47. Difference between sequential and parallel streams.
48. How does reduce() work?
49. Explain Collectors.groupingBy().
50. How does lazy evaluation work in streams?
51. Difference between map() and peek().
52. How to remove duplicates using streams?
53. How to join a list of strings into one string with delimiter?
54. How do streams work with primitive types (IntStream, LongStream)?

Spring Core

55. What is Spring Framework and why is it used?
56. What is Dependency Injection (DI)?
57. Difference between @Component, @Service, and @Repository.
58. What are Spring bean scopes?
59. Difference between ApplicationContext and BeanFactory.
60. How does @Autowired work?

61. What is @Qualifier in Spring?
62. What is the purpose of @Configuration?
63. Explain Spring bean lifecycle.
64. How do you define beans in XML vs using annotations?

Spring Boot

65. What is Spring Boot and its advantages?
66. Explain @SpringBootApplication.
67. What is dependency injection, and how does @Autowired help?
68. What are Spring Boot starters? and how do they simplify configuration ?
69. Difference between Spring and Spring Boot.
70. What is the role of application.properties?
71. What is an IOC container in Spring?

REST

72. What is HTTP and which methods do you commonly use?
73. What are common HTTP status codes (200, 404, 500)?
74. What is REST API?
75. Explain the difference between @RestController and @Controller.
76. Can you explain how you would create a simple REST API in Spring Boot?
77. Difference between PUT and PATCH.
78. What is Idempotency?

473 Mid-Level Developer

1. Collections

1. When you override equals(), why must you override hashCode() as well?
2. How does HashMap work internally? Explain hashing and collisions.
3. What happens when two keys have the same hash?
4. What is load factor and resizing in HashMap?
5. Difference between HashMap, LinkedHashMap, and TreeMap.
6. Difference between HashMap and ConcurrentHashMap (thread safety).
7. What's the difference between ArrayList, LinkedList, Vector? Which one would you use for frequent random access?
8. Difference between fail-fast and fail-safe iterators (with example).
9. How do you prevent ConcurrentModificationException?
10. How does Arrays.asList() work?
11. What are WeakHashMap and its use cases?

12. How do you synchronize a collection?

Java 8+ Features in Practice

13. What is a Functional Interface? Can you give examples from Java libraries?
14. Difference between Lambda expressions and Anonymous classes.
15. Stream API ; explain intermediate vs terminal operations.
16. map() vs flatMap() , when do you use each?
17. How would you find the 2nd highest salary in a list of employees using Streams?
18. Optional, what problems does it solve? When should you NOT use it?
19. Stream pipelines, how do they work under the hood?
20. Parallel Streams, when should you use them, and when should you avoid them?
21. What is flatMap() used for in real scenarios?
22. Explain how stream pipeline works internally.

OOPs

23. What are Sealed Classes (Java 17)? Why are they introduced?
24. Shallow copy vs Deep copy , when do you need each?
25. How does cloning work (Cloneable)? Why is it tricky?
26. Stack vs Heap memory , what goes where?
27. What is Garbage Collection? Can you explain different types of GCs (briefly)?
28. How do you handle OutOfMemoryError in production?

Multithreading & Concurrency Basics

29. What is the role of synchronized keyword?
30. Thread-local variables, where do they help in real-world projects?
31. Explain volatile vs synchronized.
32. Difference between Callable and Runnable.
33. How does ExecutorService work?
34. Explain producer-consumer problem.
35. What are deadlocks and how to prevent them?
36. What is a thread pool?
37. How do you stop a thread safely?
38. Difference between Future and CompletableFuture.
39. How do you achieve inter-thread communication in Java?

Spring Core

40. Explain AOP (Aspect Oriented Programming).
41. How does @Transactional work?
42. What are proxy objects in Spring?
43. Explain bean lifecycle in detail.
44. What are @Primary and @Qualifier?
45. How do you implement custom annotations in Spring?
46. How does Spring manage circular dependencies?

47. What is the difference between Singleton and Prototype beans?
48. Explain constructor injection vs setter injection.

Spring Boot

49. How does auto-configuration in Spring Boot work?
50. What is @Configuration vs @Bean?
51. How do profiles work in Spring Boot (@Profile use case)?
52. How do you handle global exception handling in Spring Boot (@ControllerAdvice)?
53. application.properties vs application.yml , when to prefer which?
54. What is the use of CommandLineRunner or ApplicationRunner?
55. What happens if two beans of the same type are created? How do @Qualifier and @Primary help?
56. How does the embedded Tomcat server work in Spring Boot?

REST API

57. Difference between @RequestMapping and @GetMapping.
58. @PathVariable vs @RequestParam.
59. How do you validate request inputs in Spring Boot REST APIs?
60. How would you secure a REST endpoint?
61. How do you handle exceptions in REST APIs and return proper status codes?
62. Difference between @RequestBody and @ResponseBody.
63. How to implement global exception handling in REST?
64. How do you secure REST APIs with Spring Security?
65. What are Spring Boot Actuator endpoints?
66. How do you enable pagination and sorting in Spring Data REST?

Database & JPA

67. What is ORM and why do we use it?
68. What is JPA vs Hibernate?
69. Explain @Entity, @Id, @GeneratedValue.
70. Explain lazy vs eager loading with an example.
71. How do you handle relationships like OneToMany in JPA?
72. What's the difference between save() and saveAndFlush() in Spring Data JPA?

Testing

73. What is JUnit, and what are its key annotations?
74. What is Mockito? When would you use @Mock vs @Spy?
75. What is @SpringBootTest vs @WebMvcTest?
76. How do you test REST controllers with MockMvc?

Security

77. Authentication vs Authorization, explain with examples.
78. What is Spring Security, and how does it integrate with REST APIs?

79. How would you secure an endpoint with JWT tokens?
80. Difference between 401 and 403 HTTP codes.\
81. Git commands, Git rebase vs Merge

474 Senior Developer 7+

1. JVM Internals & Performance

1. Explain JVM memory areas (Heap, Stack, Metaspace, Code Cache).
2. Different GC algorithms (Serial, Parallel, G1, ZGC). When would you choose one over the other?
3. What is a memory leak in Java despite having GC? How would you detect and fix it?
4. How would you design a thread-safe Singleton?
5. Explain Liskov Substitution Principle with an example.
6. What are anti-patterns you've seen in OOP design?
7. How do you handle large-scale refactoring in legacy OOP systems?
8. How does polymorphism help in designing extensible systems?
9. How do you design an immutable class for multi-threaded environment?
10. Explain Dependency Inversion Principle in real-world code.
11. How do you decide between inheritance and interfaces in large projects?
12. How would you apply design patterns (Factory, Strategy, Observer) in real-world apps?

Advanced Concurrency & Multithreading

13. What is the Executor Framework? Why use it instead of manually creating threads?
14. Explain Future, CompletableFuture, and use cases for async programming.
15. Difference between synchronized and ReentrantLock. When would you choose one?
16. volatile vs atomic variables, when to use?
17. Explain the Producer-Consumer problem and solve it with BlockingQueue.
18. What is ForkJoinPool? How is it different from normal thread pools?
19. What is the Java Memory Model (JMM)? How does it affect visibility of shared variables?
20. How do you make a singleton thread-safe in Java?

Advanced Spring Boot

21. Explain the Spring bean lifecycle. How can you hook into it?
22. What are proxies in Spring (JDK vs CGLIB)? Where do they matter?
23. How do you inject a prototype bean into a singleton?
24. How does Spring Boot Actuator help in production monitoring?
25. How would you build a custom health check endpoint?
26. What is @ConditionalOnProperty, and where have you used it?
27. How do you implement custom metrics & monitoring with Micrometer/Prometheus?
28. What are Scopes in Spring? How do they affect microservices apps?
29. How does Spring Boot auto-configuration work internally?

30. How does Spring handle circular dependencies?
31. Explain difference between JDK dynamic proxies and CGLIB proxies.
32. How would you profile and optimize a Spring application?
33. How do you implement distributed caching in Spring (Redis/Hazelcast)?
34. How does @Transactional work under the hood with proxies?

Advanced JPA & Databases

35. What is the N+1 select problem in JPA? How do you fix it?
36. When would you use @Query vs Criteria API vs method naming?
37. Explain optimistic locking (@Version) vs pessimistic locking.
38. How do you handle pagination and sorting in Spring Data JPA?
39. What is connection pooling? Why is HikariCP the default in Spring Boot?
40. How do you implement DB migrations in Spring Boot (Flyway/Liquibase)?
41. How do you analyze and optimize a slow SQL query?

Advanced Security

42. Explain OAuth2 + JWT flow in Spring Boot.
43. What's the difference between stateless and stateful authentication?
44. How do you implement role-based access control (RBAC) in Spring Security?
45. What's the difference between 401 Unauthorized vs 403 Forbidden?
46. How does CSRF protection work in Spring Security?
47. How do you secure microservice-to-microservice communication?
48. How does Spring Security filter chain work internally?
49. How do you design API versioning in REST?
50. How do you implement rate limiting in REST APIs?
51. Explain JWT authentication in detail.
52. How would you implement service-to-service authentication in microservices?
53. How do you design REST APIs for high scalability?
54. How do you handle distributed transactions across microservices?
55. How do you debug and profile slow REST endpoints?

Microservices & Architecture

56. Monolith vs Microservices?
57. How do Microservices communicate? (REST, gRPC, Messaging, Event-driven).
58. What is an API Gateway? Why use it?
59. What is Service Discovery and how does it work (Eureka/Consul)?
60. What is the Circuit Breaker pattern (Resilience4j/Hystrix)? When would you use it?
61. How do you implement API rate limiting?
62. How do you design for eventual consistency in microservices?
63. How do you handle a distributed transaction (Saga pattern)?

64. How do you implement distributed logging & tracing (Zipkin, ELK)?
65. What happens when one service becomes slow? How would you isolate failures?
66. One of your microservice is down then how do you handle?

System Design & Scalability

67. How would you design an e-commerce system (catalog, orders, payments)?
68. SOLID Principles and Design patterns
69. How would you design scalable notification system?
70. How do you handle millions of users without crashing the DB?
71. What is CQRS pattern? When to use it?
72. What is database sharding? When is it required?
73. How do you ensure high availability and fault tolerance?
74. How do you design a notification system (email/SMS/push)?
75. What are the different caching strategies in distributed systems? (Redis, Hazelcast)?

Production Debugging Scenarios

76. Your Spring Boot app takes 45s to start in prod, but 5s locally. How do you debug?
77. After deployment, REST APIs are responding in >5s. What's your debugging approach?
78. Your service crashes with OutOfMemoryError when traffic spikes. How do you fix it?
79. A microservice returns 503 during peak load. What's happening?
80. Users are logged out every 30 min despite JWT being valid for 24h. Root cause?

Java Cheat Sheet

These cheat sheets helpful for quick reference before interviews.

475 Java Cheat Sheet

1. Fundamentals

- **Variables & Data Types:** Primitives (int, double, boolean, char) store raw values; References (String, arrays, objects) store memory addresses. Default values differ for primitives and objects (null for references).
- **Operators:** Arithmetic (+,-,*,/,%), Relational (<,>,==), Logical (&&, ||, !), Assignment (=,+=), Unary (++,-+), Ternary (condition ? x : y), Bitwise (& , |, ^, >>, <<).
- **Conditionals:** if-else handles branching; switch-case supports multi-path execution (supports String & enum since Java 7).
- **Loops:** for, while, do-while for repetition; Enhanced-for (for-each) for collections/arrays.
- **Type Casting:** Widening (automatic, safe: int to long), Narrowing (manual, may lose data: double to int).

- **Keywords:** Reserved words like final, static, this, super, transient, volatile, synchronized.

2. Object-Oriented Programming (OOP)

- **Classes & Objects:** Class = blueprint with fields + methods, Object = runtime instance of the class.
- **Constructors:** Used to initialize objects; can be overloaded; if none defined, Java provides a default.
- **Inheritance:** extends lets child reuse parent behaviour. Single inheritance for classes, multiple inheritance via interfaces.
- **Polymorphism:** Overloading = same method name with different signatures; Overriding = subclass redefines parent method (runtime dispatch).
- **Encapsulation:** Hide variables with private and expose via getters/setters. Helps maintain control & validation.
- **Abstraction:** Abstract classes (partial abstraction) and Interfaces (full abstraction till Java 7; default methods allow behaviour from Java 8).
- **this & super:** this: reference current object, super: call parent's methods/constructors.
- **final:** Used to create constants, prevent inheritance, prevent method overriding.
- **static:** Belongs to class, not object. Used for utility methods, static blocks, static nested classes.

3. Core Concepts

- **Methods:** Modularize code; have parameters, return types, and may throw exceptions.
- **Overloading & Overriding:** Overloading = compile-time resolution; Overriding = runtime resolution (polymorphism).
- **Arrays:** Fixed-length container for same-type elements; multi-dimensional arrays for matrices.
- **Strings:** Immutable (String), mutable alternatives are StringBuilder (not thread-safe, fast) and StringBuffer (thread-safe).
- **Wrapper Classes:** Provide object representation of primitives (useful for collections, autoboxing/unboxing).
- **Enums:** Represent constant sets with methods and fields (can have constructors too).

4. Collections Framework

- **List:** Ordered, duplicates allowed (ArrayList = dynamic array, LinkedList = doubly linked).
- **Set:** No duplicates (HashSet = unordered, LinkedHashSet = insertion order, TreeSet = sorted).
- **Map:** Key-value pairs (HashMap = unordered, LinkedHashMap = ordered, TreeMap = sorted keys).
- **Queue/Deque:** FIFO (Queue), double-ended (Deque, e.g., ArrayDeque). PriorityQueue orders by comparator.
- **Collections Utility:** Helper methods like sort, binarySearch, max, unmodifiableList.

- **Stream API:** Declarative functional style with map, filter, reduce, collect. Parallel streams enable concurrency.

5. Java Streams & Functional Programming

- Functional Interfaces: Predicate, Function, Consumer, Supplier.
- Lambda expressions
- Method references (Class::method)
- Stream API: intermediate operators (map, filter, sorted) and terminal operators (collect, reduce, forEach).
- Parallel streams.
- Optional: of, empty, ofNullable, orElse, ifPresent.

6. Input/Output (I/O)

- **Console I/O:** Read with Scanner, BufferedReader; output with System.out.println().
- **File I/O:** Classes: FileReader, FileWriter, BufferedReader, BufferedWriter. Support line-by-line processing.
- **Serialization:** Persist object state to a file/network via Serializable. Mark sensitive fields transient.
- **NIO (New I/O):** Channels, Buffers, Selectors for non-blocking, scalable I/O operations.

7. Exception Handling

- **Checked vs Unchecked:** Checked must be declared or handled (IOException); Unchecked are runtime (NullPointerException).
- **try-catch-finally:** Catch exceptions gracefully; finally always executes (commonly for resource cleanup).
- **throw & throws:** throw creates an exception; throws declares that method may pass exceptions up.
- **Custom Exceptions:** Extend Exception or RuntimeException for business-specific errors.

8. Multithreading & Concurrency

- Thread Creation: Extend Thread or implement Runnable/Callable.
- Synchronization: synchronized methods/blocks ensure only one thread accesses resource. Locks/ReentrantLocks give more control.
- Volatile: Guarantees visibility of changes to variables across threads (not atomicity).
- Executor Framework: High-level API to manage pools of threads (ExecutorService, ScheduledExecutorService).
- Future & CompletableFuture: Handle async tasks and callbacks (CompletableFuture = non-blocking, chainable).
- Atomic Classes: Thread-safe operations (AtomicInteger, AtomicBoolean).

- Parallel Streams: Divide stream operations into multiple threads automatically.

9. Java Memory & Performance

- Stack vs Heap: Stack = method calls, local vars; Heap = object instances, GC-managed.
- Garbage Collection: JVM automatically reclaims memory. Different algorithms: Serial, Parallel, G1, ZGC.
- Memory Leaks: Commonly due to unclosed streams, static collections holding references.
- JVM Tuning: Configure heap size (-Xmx), GC behavior, thread stack size for optimization.

10. Java 8+ Features

- Lambdas: Concise anonymous functions ($x \rightarrow x*x$).
- Functional Interfaces: Interfaces with one abstract method (Supplier, Consumer, Predicate).
- Streams: Declarative style for processing collections with intermediate ops (map, filter) and terminal ops (collect, reduce).
- Optional: Container to avoid NullPointerException (orElse, ifPresent).
- Default & Static Methods in Interfaces: Provide implementations in interfaces without breaking old code.
- New Date/Time API: Immutable, thread-safe API (LocalDate, Period, ZonedDateTime).

11. Advanced

- Generics: Add type safety (List<String> vs List<Object>). Avoids casting errors.
- Annotations: Metadata used at compile or runtime (@Override, custom annotations, @Retention).
- Reflection API: Inspect and modify classes, fields, methods dynamically (used in frameworks).
- JDBC: API for DB connectivity using SQL queries (Connection, PreparedStatement, ResultSet).
- JPA/Hibernate: Object Relational Mapping (ORM) to simplify DB interaction via entities.
- Spring Integration: Use with Java for dependency injection, AOP, REST APIs, microservices.

12. Testing in Java

- JUnit (annotations: @Test, @BeforeEach, @AfterEach, @BeforeAll, @AfterAll)
- Mockito: mocks, spies, stubbing
- TestContainers (integration testing with DBs.)

13. Performance & Best Practices

- StringBuilder vs String concatenation
- Use primitives when possible
- Object pooling (where needed)
- Avoid unnecessary synchronization

- Caching results (Memoization)
- JVM tuning (Xms, Xmx, GC tuning)

476 Spring Cheat Sheet

1. Core Spring (IoC & DI)

- IoC (Inversion of Control): Spring manages object lifecycle.
- DI (Dependency Injection): Inject dependencies via:
 - Constructor injection (preferred)
 - Setter injection
 - Field injection (discouraged)
- Bean Scopes:
 - singleton (default)
 - prototype
 - Web scopes: request, session, application, websocket
- Bean Lifecycle:
Instantiate: Dependency Injection > @PostConstruct > @PreDestroy.
- Annotations:
 - Stereotypes: @Component, @Service, @Repository, @Controller, @RestController
 - Config & Bean definitions: @Configuration, @Bean, @Scope, @Lazy
 - Injection & qualifiers: @Autowired, @Qualifier, @Primary, @Value
 - Scanning & imports: @ComponentScan, @Import
 - Conditional configs: @Profile, @Conditional
 - Ordering: @Order, Ordered interface
- Advanced Concepts:
 - BeanFactory vs ApplicationContext (lazy vs eager loading, features)
 - BeanPostProcessor, BeanFactoryPostProcessor (customize bean lifecycle)
 - FactoryBean (produce other beans)
 - Environment & PropertySources (externalized config)
 - Circular dependencies: resolve with @Lazy or restructuring

2. Spring Boot Basics

- Main Annotation
 - @SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan

- Key Features
 - Auto-configuration (based on classpath & beans)
 - Starter dependencies (spring-boot-starter-*)
 - Embedded servers (Tomcat, Jetty, Undertow)
 - JAR packaging (spring-boot-maven-plugin)
- Configuration
 - application.properties / application.yml
 - Profiles: spring.profiles.active=dev
 - Externalized config: Environment variable, command line arguments, system properties
- Dev Tools
 - spring-boot-devtools: auto-restart, live reload
- Run Options
 - SpringApplication.run() customization
 - CommandLineRunner / ApplicationRunner for startup logic
- Actuator (Basics)
 - /actuator/health, /actuator/info
 - For monitoring & health checks

3. Web & REST (Spring MVC)

- DispatcherServlet > Front controller.
- Annotations:
 - @RequestMapping, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping.
 - @PathVariable, @RequestParam, @RequestBody, @ResponseBody.
 - @ControllerAdvice + @ExceptionHandler for global error handling.
- Response:
 - ResponseEntity<T> for status + headers + body.
- CORS:
 - @CrossOrigin, or configure via WebMvcConfigurer.

4. Data Access

- Spring JDBC:
 - JdbcTemplate, NamedParameterJdbcTemplate.
- Spring Data JPA:
 - Repos: CrudRepository, JpaRepository, PagingAndSortingRepository.

- Derived queries: `findByName`, `findByAgeGreaterThan`.
- Custom queries: `@Query`, JPQL, native SQL.
- Entities:
 - `@Entity`, `@Table`, `@Id`, `@GeneratedValue`.
 - Relationships: `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`.
- Transactions:
 - `@Transactional`.
 - Propagation: REQUIRED, REQUIRES_NEW, MANDATORY, etc.
 - Isolation: READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE.
- Lazy vs Eager Loading
 - Default to `FetchType.LAZY`.
 - Avoid `FetchType.EAGER` (leads to N+1 problems and large queries).
- N+1 Query Problem
 - Use JOIN FETCH in JPQL or `@EntityGraph` to pre-fetch relationships when needed.
- 2nd Level Cache (L2 Cache)
 - Enable Hibernate L2 cache with Ehcache/Redis.
 - Cache static reference data.
- Query Cache
 - Enable query caching for repeated queries.
 - Use carefully.
- Batch Inserts/Updates
 - Group writes using Hibernate batching.
- Avoid Select
 - Explicitly fetch only needed columns.

5. Spring Boot Advanced Features

- Profiles: Environment-specific beans with `@Profile("dev")`.
- Actuator:
 - Endpoints: `/actuator/health`, `/actuator/metrics`, `/actuator/env`.
 - Custom endpoints with `@Endpoint` / `@ReadOperation`
 - Integrates with Prometheus, Grafana.
- Scheduling & Async:
 - `@EnableScheduling`, `@Scheduled(fixedRate=1000)`.

- @EnableAsync, @Async for background tasks
- Async exception handling with AsyncUncaughtExceptionHandler
- Caching:
 - @EnableCaching, @Cacheable, @CachePut, @CacheEvict.
 - Cache providers: Caffeine, Redis, Ehcache, Hazelcast

6. AOP (Aspect Oriented Programming)

- Concepts:
 - Aspect: cross-cutting concern.
 - Join point: method execution.
 - Advice: code run at join points.
 - Pointcut: expression matching join points.
- Annotations:
 - @Aspect, @Before, @After, @Around, @AfterReturning, @AfterThrowing.

7. Security

- **Spring Security Basics**
 - Filters: SecurityFilterChain (replaces WebSecurityConfigurerAdapter)
 - @EnableWebSecurity
 - DelegatingFilterProxy: Ties security filters into servlet filter chain
- **Authentication**
 - In-memory users (InMemoryUserDetailsManager)
 - JDBC-based authentication (with UserDetailsService)
 - LDAP-based authentication (common in enterprises)
 - JWT-based (stateless APIs)
 - OAuth2 / OpenID Connect (social login, SSO)
- **Authorization**
 - Role-based access: hasRole, hasAuthority
 - Method-level: @PreAuthorize, @Secured, @RolesAllowed
 - URL-based: .authorizeHttpRequests() in security config
- **Password Management**
 - PasswordEncoder (e.g., BCryptPasswordEncoder)
- **Filters & Interceptors in Security**
 - Security filters (auth, CSRF, CORS) run before controllers
 - Custom filters: extend OncePerRequestFilter

- Custom interceptors for logging/auditing
- **CORS (Cross-Origin Resource Sharing)**
 - Configure via @CrossOrigin or global config in WebMvcConfigurer

8. Web Security

- CSRF Protection
 - Enabled by default in Spring Security (CsrfFilter)
 - Use CSRF tokens in forms (<input type="hidden" name="_csrf">)
 - Disable only for stateless APIs with JWT
- XSS Protection
 - Escape output in views (Thymeleaf auto-escapes by default)
 - Use Spring's HtmlUtils.htmlEscape() for manual escaping
 - Set response headers (X-XSS-Protection, Content-Security-Policy)
- SQL Injection Prevention
 - Always use prepared statements / parameterized queries
 - With Spring Data JPA : repository methods prevent injection
 - Avoid string concatenation in queries ("WHERE name=' + input + '')")

9. Testing

- Test Annotations:
 - @SpringBootTest: full context.
 - @WebMvcTest: controllers only.
 - @DataJpaTest: JPA only.
- Mocking:
 - @MockBean, Mockito.
 - MockMvc for REST testing.
- Database Tests:
 - @AutoConfigureTestDatabase, H2 in-memory DB.

10. Filters & Interceptors

- **Servlet Filters** (javax.servlet.Filter / OncePerRequestFilter)
 - Run before request reaches DispatcherServlet
 - Used for logging, authentication, CORS, request wrapping
- **Spring Interceptors** (HandlerInterceptor)

- Run before and after controller execution
- Good for auditing, request validation, modifying model attributes
- **Difference:**
 - Filters: generic to all requests (Servlet level)
 - Interceptors: Tied to Spring MVC request lifecycle, Tied to controllers

11. Reactive (Spring WebFlux)

- When to use: Non-blocking, async, high-throughput apps.
- Types:
 - Mono<T>: 0/1 element.
 - Flux<T>: 0...N elements.
- Programming Model:
 - Annotation style (@RestController).
 - Functional style (RouterFunction).

12. Spring Cloud & Microservices

- Components:
 - Eureka (service registry).
 - Config Server (centralized config).
 - Feign Client (REST calls).
 - Gateway (API routing).
 - Resilience4j (Circuit Breaker).
 - Sleuth + Zipkin (tracing).
- Distributed Config:
 - Git-backed configs, refresh with /actuator/refresh.

13. Validation

- Bean Validation (JSR 380):
 - @NotNull, @NotBlank, @Size, @Email, @Pattern, @Min, @Max.
 - Custom validator with ConstraintValidator.
- Usage:
 - @Valid on method params in REST controllers.

14. Events

- Publishing:

```
applicationEventPublisher.publishEvent(new CustomEvent(...));
```

- Listening:
@EventListener(CustomEvent.class).
- Built-in Events:
ContextRefreshedEvent, ContextClosedEvent, ApplicationReadyEvent.

15. Legacy (XML & Old Features)

- XML-based config (applicationContext.xml).
- context:component-scan, bean definitions.

16. Deployment & Observability

- Zero Downtime:
 - Graceful shutdown (server.shutdown=graceful).
 - Readiness + liveness probes.
- Monitoring:
 - Spring Boot Actuator + Micrometer.
 - Logs: SLF4J, Logback.
- Packaging:
 - Executable JAR/WAR.
 - Deploy on Docker, Kubernetes, or Cloud.

17. Performance

Startup Optimization

- Lazy Init: spring.main.lazy-initialization=true
- Limit @ComponentScan scope
- Remove unused starters/dependencies
- Analyze startup with Spring Boot Startup Actuator

Runtime Performance Optimization

- Switch to WebClient (reactive, non-blocking) instead of RestTemplate
- Connection pooling: HikariCP (tune max pool size)
- Cache: Spring Cache (Redis, Caffeine)
- Asynchronous processing: @Async, Kafka, queues
- Batch inserts/updates: JPA batching, bulk APIs

Database Performance

- Use proper indexes (single, composite, covering indexes)
- Optimize queries with JPQL/Criteria instead of fetching everything

- Fetch strategies: prefer lazy loading but balance with JOIN FETCH where needed
- Pagination with Pageable to avoid loading huge datasets
- Avoid N+1 problem: use @EntityGraph, JOIN FETCH, or DTO projections
- Statement batching in JPA (hibernate.jdbc.batch_size)
- Connection pool tuning (HikariCP settings)
- Use database-specific features (partitions, materialized views, query hints)

API Performance Optimization

- **Compression:** Enable GZIP (server.compression.enabled=true in Spring Boot)
- **Response Size :** Use pagination (Pageable, Slice), projections, and DTOs to avoid over-fetching
- **Caching Strategies**
 - Client-side: Cache-Control, ETag, Last-Modified headers
 - Server-side: Spring Cache (Redis, Caffeine), HTTP response caching
- **Connection Keep-Alive:** Reduce handshake overhead
- **Rate Limiting / Throttling:** Bucket4j, Resilience4j
- **Asynchronous APIs:** WebFlux, @Async, message queues (Kafka/RabbitMQ)

477 November

478 December

- Q. When would you prefer CopyOnWriteArrayList?
- Q. How would you get a return value from a thread?
- Q. How does ThreadPoolExecutor manage threads internally?
- Q. What happens if all threads are busy?
- Q. You have 100 tasks, but only 10 threads available - how will you execute all efficiently?
- Q. Explain the CompletableFuture class and give a real-world use cases.
- Q. What happens behind the scenes when a Spring Boot app starts?
- Q. If you declare @Bean and @Component for the same class, which one takes precedence?
- Q. What is Projection in Spring Data JPA
- Q. What is marshaling and unmarshaling in java

Coding Lyf