

# Java, Spring Boot, Microservices, and Angular Interview Preparation Guide

August 2025

## Contents

<b>1 Core Java &amp; OOPs</b>	<b>4</b>
1.1 Difference between String, StringBuilder, and StringBuffer . . . . .	4
1.2 Difference between .equals() method and == operator . . . . .	5
1.3 Difference between method overloading and method overriding . . . . .	5
1.4 What is Singleton class? . . . . .	5
1.5 Create Singleton class . . . . .	5
1.6 What is thread safety and how do you ensure thread-safe classes? . . . . .	6
1.7 How does HashMap work internally? . . . . .	6
1.8 Difference: HashMap vs Hashtable vs ConcurrentHashMap . . . . .	6
1.9 How does HashMap work with Employee object as key? . . . . .	6
1.10 What is immutability and how does it help in concurrency? . . . . .	7
1.11 What is volatile and synchronized? . . . . .	7
1.12 How many ways can an object be created in Java? . . . . .	7
1.13 What is the use of ResponseEntity? . . . . .	8
1.14 What is meant by functional interfaces? . . . . .	8
1.15 How do functional interfaces work? . . . . .	8
1.16 What is dependency injection and its types? . . . . .	8
1.17 Difference between IOC and Dependency Injection . . . . .	9
1.18 How many ways to achieve dependency injection and which is best? . . . . .	9
1.19 @Primary vs @Qualifier — which takes priority? . . . . .	9
1.20 How to create beans manually? . . . . .	9
1.21 What is the difference between @RestController and @Controller? . . . . .	9
1.22 What is @SpringBootApplication annotation? . . . . .	10
1.23 What are stereotype annotations in Spring Boot? . . . . .	10
1.24 What is path variable? . . . . .	10
1.25 What is the default server in Spring Boot? . . . . .	10
1.26 What is the default port in Spring Boot? . . . . .	10
<b>2 Java 8 Features</b>	<b>11</b>
2.1 Java 8 features used in your project . . . . .	11
2.2 What is a lambda expression? . . . . .	11
2.3 Write code using lambda expression . . . . .	11
2.4 What is Stream API? . . . . .	11
2.5 Intermediate vs terminal operations in streams . . . . .	12
2.6 List of intermediate and terminal methods . . . . .	12
2.7 Why Java introduced default and static methods in interfaces . . . . .	12
2.8 How to create an immutable class in Java . . . . .	12

2.9 How to use groupingBy() in streams . . . . .	13
2.10 How to create an Optional of an employee object . . . . .	13
2.11 Use Java 8 streams to remove duplicates from a list . . . . .	13
2.12 Find 3rd highest salary using Java 8 streams . . . . .	13
<b>3 Spring Boot &amp; Microservices</b>	<b>14</b>
3.1 How do you configure different environments in Spring Boot? . . . . .	14
3.2 What is a Spring Boot profile and how did you use it? . . . . .	14
3.3 What is exception handling and what is an advisor? . . . . .	14
3.4 How does @Transactional annotation work? . . . . .	14
3.5 What are the design patterns used in microservices? . . . . .	15
3.6 Explain each design pattern and when to use them . . . . .	15
3.7 What is microservices architecture? . . . . .	15
3.8 Why microservices? . . . . .	15
3.9 How do microservices communicate with each other? . . . . .	15
3.10 Difference between synchronous and asynchronous communication . . . . .	15
3.11 When to use synchronous vs asynchronous communication . . . . .	16
3.12 What is Kafka and how have you implemented it? . . . . .	16
3.13 How have you used Spring Security in your project? . . . . .	16
3.14 What is JWT security and how have you used it? . . . . .	16
3.15 What are the things to consider while developing REST APIs? . . . . .	17
3.16 Difference between PUT, POST, and PATCH . . . . .	17
3.17 API versioning strategies . . . . .	17
3.18 What is idempotency and why is it critical? . . . . .	17
3.19 How do you achieve idempotency in microservices? . . . . .	18
3.20 Securing APIs using JWT and OAuth2 . . . . .	18
3.21 Validating incoming payloads with annotations . . . . .	18
3.22 What are the steps to test Spring Boot microservices applications? . . . . .	18
3.23 How to handle exceptions in Spring Boot . . . . .	19
3.24 How to create global exceptions and what annotations are used . . . . .	19
3.25 How to exclude classes from component scan . . . . .	19
3.26 How does component scan work? . . . . .	19
3.27 What is the most challenging task you've done? . . . . .	19
3.28 What are the top 3 performance bottlenecks in microservices? . . . . .	19
3.29 How do you monitor microservices? . . . . .	20
3.30 How do you ensure system resiliency under high load? . . . . .	20
3.31 What is centralized configuration and secrets management? . . . . .	20
3.32 What is service discovery (Eureka/Consul)? . . . . .	20
3.33 Inter-service communication: Feign vs RestTemplate vs WebClient . . . . .	20
3.34 Circuit Breaker and Retry: Resilience4j . . . . .	20
3.35 Blue-green vs canary deployments . . . . .	21
3.36 Handling version mismatch between services . . . . .	21
3.37 What are Maven commands you use daily? . . . . .	21
3.38 Have you used Docker? Benefits and challenges? . . . . .	21
<b>4 Spring &amp; Bean Lifecycle</b>	<b>21</b>
4.1 Bean lifecycle and Spring container . . . . .	21
4.2 Use of @Component, @Service, @Repository . . . . .	21
4.3 Role of @ComponentScan, @Configuration, @Bean . . . . .	22
4.4 What is Spring Boot auto-configuration? . . . . .	22
<b>5 JPA &amp; Database</b>	<b>22</b>

5.1	What is Fetch Type (Lazy vs Eager Loading)? . . . . .	22
5.2	Lazy vs Eager loading — real-time use cases . . . . .	22
5.3	Complex entity relationship experience . . . . .	22
5.4	What is the N+1 query problem? . . . . .	22
5.5	How to optimize N+1 using Spring Data JPA . . . . .	23
5.6	Difference between get() and load() in Hibernate . . . . .	23
5.7	Writing optimized JPQL and Criteria queries . . . . .	23
5.8	Transaction management: @Transactional deep dive . . . . .	23
<b>6</b>	<b>Testing &amp; Mocking</b>	<b>23</b>
6.1	Differences between stubbing and mocking . . . . .	23
6.2	Why do we need both approaches? . . . . .	24
6.3	What is Spy in Mockito and when to use it? . . . . .	24
6.4	How to write JUnit test cases for static methods . . . . .	24
6.5	What is the use of Mockito framework? . . . . .	24
<b>7</b>	<b>Coding &amp; Problem Solving</b>	<b>24</b>
7.1	Reverse a string by preserving word position . . . . .	24
7.2	Remove duplicates from string/array/list . . . . .	25
7.3	Longest substring without repeating characters . . . . .	25
7.4	Check if two strings/numbers are palindrome . . . . .	25
7.5	Check if two strings/numbers are anagram . . . . .	26
7.6	Sort an array/list/string . . . . .	26
7.7	Count occurrence of characters in a string . . . . .	26
7.8	Print duplicate characters in a string . . . . .	27
7.9	Print only special characters in a string . . . . .	27
7.10	Reverse the given string . . . . .	27
7.11	Print difference between two strings . . . . .	27
7.12	Count words in a string . . . . .	28
7.13	Find second highest element in an array . . . . .	28
7.14	Find common elements of two arrays . . . . .	28
7.15	Decode string like a2b3c1 to aabbcc . . . . .	29
7.16	Check if a number is prime . . . . .	29
7.17	Generate Fibonacci series . . . . .	29
7.18	Find factorial of a number . . . . .	30
7.19	Find min and max element in an array . . . . .	30
7.20	Find max repeated word in a sentence . . . . .	30
7.21	Rotate array from k=2 . . . . .	30
7.22	Merge characters from two strings alternately . . . . .	31
7.23	Find max repeated word in a sentence . . . . .	31
7.24	Modify and improve given code . . . . .	31
7.25	Find a file in a subdirectory . . . . .	32
7.26	Write a method to fetch employee details . . . . .	32
<b>8</b>	<b>Angular &amp; Frontend</b>	<b>32</b>
8.1	What is ng-content and how does content projection work? . . . . .	32
8.2	Difference between @ViewChild and @ContentChild . . . . .	32
8.3	Explain Dependency Injection in Angular . . . . .	33
8.4	How does Angular load dynamic components? . . . . .	33
8.5	What is the PipeTransform interface? . . . . .	33
8.6	How does Angular bootstrapping work via AppModule? . . . . .	34
8.7	What is an HTTP interceptor? . . . . .	34

8.8	Common use cases of HTTP interceptors . . . . .	34
8.9	What is GraphQL and how does it compare to REST? . . . . .	34
8.10	Use Apollo in Angular to fetch GraphQL data . . . . .	34
8.11	Differences: BehaviorSubject, Subject, ReplaySubject . . . . .	35
8.12	Explain: switchMap, mergeMap, concatMap, exhaustMap . . . . .	35
8.13	Design a semantic HTML navigation menu . . . . .	35
8.14	List 5 semantic HTML tags and their uses . . . . .	35
8.15	Fetch API data and display in a table . . . . .	35
8.16	Search by name using Reactive Form . . . . .	36
8.17	Error handling on API failure . . . . .	36
8.18	Flatten a nested array . . . . .	36
<b>9</b>	<b>Miscellaneous</b>	<b>36</b>
9.1	What do you know about ISO8583? . . . . .	36
9.2	How do Angular applications interact with backend APIs? . . . . .	36
9.3	How do you clone code from Git and commit changes? . . . . .	37
9.4	How to resolve merge conflicts? . . . . .	37
9.5	Daily Git commands you use . . . . .	37
<b>10</b>	<b>Additional Frequently Asked Questions</b>	<b>37</b>
10.1	What is the difference between checked and unchecked exceptions? . . . . .	37
10.2	What is the purpose of @Autowired annotation in Spring? . . . . .	37
10.3	What is the Circuit Breaker pattern in microservices? . . . . .	37

## Introduction

This document provides answers and examples for Java, Spring Boot, Microservices, and Angular interview questions, organized by topic. Each question includes a beginner-friendly explanation and a practical example, suitable for interview preparation. Additional common questions are included at the end.

## 1 Core Java & OOPs

### 1.1 Difference between String, StringBuilder, and StringBuffer

**Answer:** String is immutable, creating new objects for modifications. StringBuilder is mutable, non-thread-safe, and fast. StringBuffer is mutable, thread-safe, but slower due to synchronization.

```

1 public class StringExample {
2     public static void main(String[] args) {
3         String str = "Hello";
4         str += " World"; // New object
5         System.out.println(str); // Hello World
6         StringBuilder sb = new StringBuilder("Hello");
7         sb.append(" World");
8         System.out.println(sb); // Hello World
9         StringBuffer sbf = new StringBuffer("Hello");
10        sbf.append(" World");
11        System.out.println(sbf); // Hello World
12    }
13 }
```

## 1.2 Difference between .equals() method and == operator

**Answer:** == compares object references or primitive values. .equals() compares object content, customizable for classes.

```
1 public class EqualsExample {  
2     public static void main(String[] args) {  
3         String s1 = new String("Hello");  
4         String s2 = new String("Hello");  
5         System.out.println(s1 == s2); // false  
6         System.out.println(s1.equals(s2)); // true  
7         String s3 = "Hello";  
8         String s4 = "Hello";  
9         System.out.println(s3 == s4); // true  
10    }  
11 }
```

## 1.3 Difference between method overloading and method overriding

**Answer:** Overloading uses different parameters in the same class (compile-time). Overriding redefines a method in a subclass (runtime).

```
1 class Animal {  
2     void sound() { System.out.println("Animal sound"); }  
3 }  
4 class Dog extends Animal {  
5     void sound() { System.out.println("Dog barks"); } // Override  
6     void sound(String type) { System.out.println("Dog " + type); } // Overload  
7 }  
8 public class PolymorphismExample {  
9     public static void main(String[] args) {  
10        Dog dog = new Dog();  
11        dog.sound(); // Dog barks  
12        dog.sound("growls"); // Dog growls  
13    }  
14 }
```

## 1.4 What is Singleton class?

**Answer:** Ensures one instance with global access, used for shared resources.

## 1.5 Create Singleton class

**Answer:** Uses private constructor, static instance, and static access method.

```
1 public class Singleton {  
2     private static Singleton instance;  
3     private Singleton() {}  
4     public static synchronized Singleton getInstance() {  
5         if (instance == null) instance = new Singleton();  
6         return instance;  
7     }  
8     public static void main(String[] args) {  
9         Singleton s1 = Singleton.getInstance();  
10        Singleton s2 = Singleton.getInstance();  
11        System.out.println(s1 == s2); // true  
12    }
```

13 }

## 1.6 What is thread safety and how do you ensure thread-safe classes?

**Answer:** Thread safety prevents data corruption in multi-threaded environments using synchronization, immutability, or thread-safe classes.

```
1 public class ThreadSafeCounter {  
2     private int count = 0;  
3     public synchronized void increment() { count++; }  
4     public synchronized int getCount() { return count; }  
5     public static void main(String[] args) throws InterruptedException {  
6         ThreadSafeCounter counter = new ThreadSafeCounter();  
7         Runnable task = () -> { for (int i = 0; i < 1000; i++) counter.  
8             increment(); };  
9         Thread t1 = new Thread(task); Thread t2 = new Thread(task);  
10        t1.start(); t2.start(); t1.join(); t2.join();  
11        System.out.println(counter.getCount()); // 2000  
12    }  
13 }
```

## 1.7 How does HashMap work internally?

**Answer:** Uses a hash table with buckets. Keys' hashCode() determines bucket index; collisions use linked lists or trees (Java 8+).

```
1 import java.util.HashMap;  
2 public class HashMapExample {  
3     public static void main(String[] args) {  
4         HashMap<String, Integer> map = new HashMap<>();  
5         map.put("A", 1);  
6         map.put("B", 2);  
7         System.out.println(map.get("A")); // 1  
8     }  
9 }
```

## 1.8 Difference: HashMap vs Hashtable vs ConcurrentHashMap

**Answer:** HashMap: non-thread-safe, allows nulls. Hashtable: thread-safe, no nulls. ConcurrentHashMap: thread-safe, concurrent access, no nulls.

```
1 import java.util.*;  
2 public class MapComparison {  
3     public static void main(String[] args) {  
4         HashMap<String, Integer> hm = new HashMap<>();  
5         hm.put(null, 1);  
6         System.out.println(hm); // {null=1}  
7         Hashtable<String, Integer> ht = new Hashtable<>();  
8         ConcurrentHashMap<String, Integer> chm = new ConcurrentHashMap<>();  
9     }  
10 }
```

## 1.9 How does HashMap work with Employee object as key?

**Answer:** Uses Employee's hashCode() and equals() for bucket placement and collision resolution.

```

1 import java.util.HashMap;
2 class Employee {
3     int id; String name;
4     Employee(int id, String name) { this.id = id; this.name = name; }
5     @Override public int hashCode() { return id * 31 + name.hashCode(); }
6     @Override public boolean equals(Object obj) {
7         if (!(obj instanceof Employee)) return false;
8         Employee other = (Employee) obj;
9         return id == other.id && name.equals(other.name);
10    }
11 }
12 public class HashMapEmployee {
13     public static void main(String[] args) {
14         HashMap<Employee, String> map = new HashMap<>();
15         Employee e1 = new Employee(1, "Alice");
16         map.put(e1, "Developer");
17         System.out.println(map.get(new Employee(1, "Alice"))); // Developer
18     }
19 }
```

## 1.10 What is immutability and how does it help in concurrency?

**Answer:** Immutability prevents state changes, ensuring thread safety without synchronization.

```

1 public final class ImmutableClass {
2     private final int value;
3     public ImmutableClass(int value) { this.value = value; }
4     public int getValue() { return value; }
5     public static void main(String[] args) {
6         ImmutableClass obj = new ImmutableClass(42);
7         System.out.println(obj.getValue()); // 42
8     }
9 }
```

## 1.11 What is volatile and synchronized?

**Answer:** volatile ensures variable visibility; synchronized ensures mutual exclusion.

```

1 public class VolatileSynchronized {
2     private volatile boolean running = true;
3     public synchronized void update() { running = false; }
4     public static void main(String[] args) throws InterruptedException {
5         VolatileSynchronized vs = new VolatileSynchronized();
6         new Thread(() -> { while (vs.running) {} } ).start();
7         Thread.sleep(1000);
8         vs.update();
9     }
10 }
```

## 1.12 How many ways can an object be created in Java?

**Answer:** Using new, Class.forName(), clone(), deserialization, factory methods.

```

1 public class ObjectCreation {
2     public static void main(String[] args) throws Exception {
3         ObjectCreation obj1 = new ObjectCreation();
4         ObjectCreation obj2 = (ObjectCreation) Class.forName("ObjectCreation").
5             newInstance();
6     }
}

```

### 1.13 What is the use of ResponseEntity?

**Answer:** Represents HTTP response with status, headers, and body.

```

1 import org.springframework.http.*;
2 import org.springframework.web.bind.annotation.*;
3 @RestController
4 public class ResponseEntityExample {
5     @GetMapping("/user")
6     public ResponseEntity<String> getUser() {
7         return new ResponseEntity<>("User found", HttpStatus.OK);
8     }
9 }

```

### 1.14 What is meant by functional interfaces?

**Answer:** Interfaces with one abstract method, used with lambdas.

```

1 @FunctionalInterface
2 interface MyFunction { void apply(String s); }
3 public class FunctionalInterfaceExample {
4     public static void main(String[] args) {
5         MyFunction func = s -> System.out.println(s);
6         func.apply("Hello"); // Hello
7     }
8 }

```

### 1.15 How do functional interfaces work?

**Answer:** Enable functional programming via lambda expressions.

```

1 import java.util.function.Consumer;
2 public class FunctionalInterfaceWork {
3     public static void main(String[] args) {
4         Consumer<String> consumer = s -> System.out.println(s);
5         consumer.accept("Hello Consumer"); // Hello Consumer
6     }
7 }

```

### 1.16 What is dependency injection and its types?

**Answer:** Provides dependencies externally. Types: constructor, setter, field injection.

```

1 import org.springframework.stereotype.Component;
2 import org.springframework.beans.factory.annotation.Autowired;
3 @Component
4 class Service { public void serve() { System.out.println("Serving"); } }
5 @Component

```

```

6 class Client {
7     private final Service service;
8     @Autowired
9     public Client(Service service) { this.service = service; }
10    public void doWork() { service.serve(); }
11 }

```

## 1.17 Difference between IOC and Dependency Injection

**Answer:** IoC inverts control to a framework; DI is a way to achieve IoC.

```

1 // See above (Question 16)

```

## 1.18 How many ways to achieve dependency injection and which is best?

**Answer:** Constructor, setter, field injection. Constructor is best for explicit dependencies.

```

1 // See Question 16

```

## 1.19 @Primary vs @Qualifier — which takes priority?

**Answer:** @Primary sets default bean; @Qualifier overrides it.

```

1 import org.springframework.stereotype.Component;
2 import org.springframework.beans.factory.annotation.*;
3 interface Service {}
4 @Component @Primary
5 class DefaultService implements Service {}
6 @Component @Qualifier("special")
7 class SpecialService implements Service {}
8 @Component
9 class Client {
10     @Autowired @Qualifier("special") Service service;
11 }

```

## 1.20 How to create beans manually?

**Answer:** Use @Bean in a @Configuration class.

```

1 import org.springframework.context.annotation.*;
2 @Configuration
3 public class AppConfig {
4     @Bean
5     public Service myService() { return new Service(); }
6 }

```

## 1.21 What is the difference between @RestController and @Controller?

**Answer:** @Controller returns view names; @RestController returns data (e.g., JSON).

```

1 import org.springframework.stereotype.*;
2 import org.springframework.web.bind.annotation.*;
3 @Controller
4 public class MyController {
5     @GetMapping("/view")

```

```

6   public String getView() { return "view"; }
7 }
8 @RestController
9 public class MyRestController {
10   @GetMapping("/api")
11   public String getData() { return "Data"; }
12 }
```

## 1.22 What is @SpringBootApplication annotation?

**Answer:** Combines @EnableAutoConfiguration, @ComponentScan, @Configuration.

```

1 import org.springframework.boot.*;
2 import org.springframework.boot.autoconfigure.*;
3 @SpringBootApplication
4 public class Application {
5   public static void main(String[] args) {
6     SpringApplication.run(Application.class, args);
7   }
8 }
```

## 1.23 What are stereotype annotations in Spring Boot?

**Answer:** @Component, @Service, @Repository, @Controller mark classes for scanning.

```

1 import org.springframework.stereotype.*;
2 @Service
3 public class MyService {
4   public void serve() { System.out.println("Service"); }
5 }
```

## 1.24 What is path variable?

**Answer:** @PathVariable extracts URL path values.

```

1 import org.springframework.web.bind.annotation.*;
2 @RestController
3 public class PathVariableExample {
4   @GetMapping("/user/{id}")
5   public String getUser(@PathVariable int id) {
6     return "User ID: " + id;
7   }
8 }
```

## 1.25 What is the default server in Spring Boot?

**Answer:** Tomcat.

```

1 // Run Application.java to start Tomcat
```

## 1.26 What is the default port in Spring Boot?

**Answer:** 8080.

```

1 // Access http://localhost:8080
```

## 2 Java 8 Features

### 2.1 Java 8 features used in your project

**Answer:** Lambda expressions, Stream API, Optional, default/static methods, forEach.

```
1 import java.util.*;
2 public class Java8Features {
3     public static void main(String[] args) {
4         List<String> list = Arrays.asList("A", "B");
5         list.forEach(s -> System.out.println(s)); // A, B
6     }
7 }
```

### 2.2 What is a lambda expression?

**Answer:** Concise function representation: (params) -> expression.

```
1 import java.util.*;
2 public class LambdaExample {
3     public static void main(String[] args) {
4         List<String> list = Arrays.asList("A", "B");
5         list.forEach(s -> System.out.println(s)); // A, B
6     }
7 }
```

### 2.3 Write code using lambda expression

```
1 import java.util.*;
2 public class LambdaCode {
3     public static void main(String[] args) {
4         List<Integer> numbers = Arrays.asList(1, 2, 3);
5         numbers.forEach(n -> System.out.println(n * 2)); // 2, 4, 6
6     }
7 }
```

### 2.4 What is Stream API?

**Answer:** Processes collections functionally with operations like filter, map, collect.

```
1 import java.util.*;
2 import java.util.stream.*;
3 public class StreamExample {
4     public static void main(String[] args) {
5         List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
6         List<Integer> evens = numbers.stream()
7             .filter(n -> n % 2 == 0)
8             .collect(Collectors.toList());
9         System.out.println(evens); // [2, 4]
10    }
11 }
```

## 2.5 Intermediate vs terminal operations in streams

**Answer:** Intermediate (lazy, e.g., filter, map); terminal (triggers, e.g., collect, forEach).

```
1 import java.util.*;
2 public class StreamOperations {
3     public static void main(String[] args) {
4         List<Integer> numbers = Arrays.asList(1, 2, 3);
5         numbers.stream().filter(n -> n > 1).forEach(System.out::println); // 2,
6                                         // 3
7     }
8 }
```

## 2.6 List of intermediate and terminal methods

**Answer:** Intermediate: filter, map, sorted, distinct. Terminal: collect, forEach, reduce, count.

```
1 import java.util.*;
2 import java.util.stream.*;
3 public class StreamMethods {
4     public static void main(String[] args) {
5         List<Integer> numbers = Arrays.asList(1, 2, 2, 3);
6         long count = numbers.stream().distinct().count(); // 3
7         System.out.println(count);
8     }
9 }
```

## 2.7 Why Java introduced default and static methods in interfaces

**Answer:** Default: Add methods without breaking implementations. Static: Utility methods.

```
1 interface MyInterface {
2     default void defaultMethod() { System.out.println("Default"); }
3     static void staticMethod() { System.out.println("Static"); }
4 }
5 class MyClass implements MyInterface {}
6 public class InterfaceExample {
7     public static void main(String[] args) {
8         new MyClass().defaultMethod(); // Default
9         MyInterface.staticMethod(); // Static
10    }
11 }
```

## 2.8 How to create an immutable class in Java

**Answer:** Use final class, final fields, no setters, deep copy for mutable objects.

```
1 public final class ImmutableClass {
2     private final int value;
3     public ImmutableClass(int value) { this.value = value; }
4     public int getValue() { return value; }
5     public static void main(String[] args) {
6         ImmutableClass obj = new ImmutableClass(42);
7         System.out.println(obj.getValue()); // 42
8     }
9 }
```

## 2.9 How to use groupingBy() in streams

**Answer:** Groups elements by a classifier.

```
1 import java.util.*;
2 import java.util.stream.*;
3 public class GroupingByExample {
4     public static void main(String[] args) {
5         List<String> names = Arrays.asList("Alice", "Bob", "Adam");
6         Map<Character, List<String>> grouped = names.stream()
7             .collect(Collectors.groupingBy(s -> s.charAt(0)));
8         System.out.println(grouped); // {A=[Alice, Adam], B=[Bob]}
9     }
10 }
```

## 2.10 How to create an Optional of an employee object

**Answer:** Wraps object to handle null cases.

```
1 import java.util.Optional;
2 class Employee {
3     String name;
4     Employee(String name) { this.name = name; }
5 }
6 public class OptionalExample {
7     public static void main(String[] args) {
8         Optional<Employee> emp = Optional.of(new Employee("Alice"));
9         System.out.println(emp.get().name); // Alice
10    }
11 }
```

## 2.11 Use Java 8 streams to remove duplicates from a list

```
1 import java.util.*;
2 import java.util.stream.*;
3 public class RemoveDuplicates {
4     public static void main(String[] args) {
5         List<Integer> numbers = Arrays.asList(1, 2, 2, 3);
6         List<Integer> unique = numbers.stream().distinct().collect(Collectors.
7             toList());
8         System.out.println(unique); // [1, 2, 3]
9     }
10 }
```

## 2.12 Find 3rd highest salary using Java 8 streams

```
1 import java.util.*;
2 import java.util.stream.*;
3 class Employee {
4     double salary;
5     Employee(double salary) { this.salary = salary; }
6     double getSalary() { return salary; }
7 }
8 public class ThirdHighestSalary {
9     public static void main(String[] args) {
10         List<Employee> employees = Arrays.asList(
11             new Employee(100),
12             new Employee(200),
13             new Employee(300),
14             new Employee(400),
15             new Employee(500),
16             new Employee(600),
17             new Employee(700),
18             new Employee(800),
19             new Employee(900),
20             new Employee(1000)
21         );
22         employees.stream()
23             .sorted(Comparator.comparingDouble(Employee::getSalary).reversed())
24             .skip(2)
25             .findFirst()
26             .ifPresent(System.out::println);
27     }
28 }
```

```

11     new Employee(50000), new Employee(70000), new Employee(60000)
12 );
13 double thirdHighest = employees.stream()
14     .map(Employee::getSalary)
15     .distinct()
16     .sorted(Comparator.reverseOrder())
17     .skip(2)
18     .findFirst()
19     .orElse(0.0);
20 System.out.println(thirdHighest); // 0.0 (if <3 salaries)
21 }
22 }
```

### 3 Spring Boot & Microservices

#### 3.1 How do you configure different environments in Spring Boot?

**Answer:** Use application-{profile}.properties and spring.profiles.active.

```

1 # application-dev.properties
2 server.port=8081
```

```

1 import org.springframework.context.annotation.*;
2 @Profile("dev")
3 @Component
4 public class DevConfig {}
```

#### 3.2 What is a Spring Boot profile and how did you use it?

**Answer:** Profiles activate environment-specific configurations.

```

1 # application.properties
2 spring.profiles.active=dev
```

#### 3.3 What is exception handling and what is an advisor?

**Answer:** Exception handling manages errors; advisor combines AOP advice and point-cut.

```

1 import org.springframework.http.*;
2 import org.springframework.web.bind.annotation.*;
3 @RestController
4 public class ExceptionController {
5     @ExceptionHandler(NullPointerException.class)
6     public ResponseEntity<String> handleNPE() {
7         return new ResponseEntity<>("Error", HttpStatus.BAD_REQUEST);
8     }
9 }
```

#### 3.4 How does @Transactional annotation work?

**Answer:** Manages transactions, ensuring atomicity.

```

1 import org.springframework.stereotype.*;
2 import org.springframework.transaction.annotation.*;
3 @Service
4 public class UserService {
5     @Transactional
6     public void saveUser() {}
7 }
```

### 3.5 What are the design patterns used in microservices?

**Answer:** Circuit Breaker, API Gateway, Service Discovery, Event-Driven, CQRS.

### 3.6 Explain each design pattern and when to use them

**Answer:** Circuit Breaker: Prevents failures. API Gateway: Routing/security. Service Discovery: Dynamic location. Event-Driven: Loose coupling. CQRS: Complex domains.

```

1 import io.github.resilience4j.circuitbreaker.annotation.*;
2 @Service
3 public class MyService {
4     @CircuitBreaker(name = "myService")
5     public String callApi() { return "Success"; }
6 }
```

### 3.7 What is microservices architecture?

**Answer:** Small, independent services communicating via APIs.

```

1 import org.springframework.web.bind.annotation.*;
2 @RestController
3 public class UserController {
4     @GetMapping("/users")
5     public String getUsers() { return "User List"; }
6 }
```

### 3.8 Why microservices?

**Answer:** Scalability, independent deployment, fault isolation.

### 3.9 How do microservices communicate with each other?

**Answer:** Via REST, message queues, or gRPC.

```

1 import org.springframework.web.client.RestTemplate;
2 public class Communication {
3     public static void main(String[] args) {
4         RestTemplate restTemplate = new RestTemplate();
5         String response = restTemplate.getForObject("http://other-service/users",
6             String.class);
7     }
}
```

### 3.10 Difference between synchronous and asynchronous communication

**Answer:** Synchronous: Blocking (REST). Asynchronous: Non-blocking (Kafka).

### 3.11 When to use synchronous vs asynchronous communication

**Answer:** Synchronous for immediate responses; asynchronous for decoupled systems.

```
1 import org.springframework.kafka.annotation.*;
2 @Service
3 public class KafkaConsumer {
4     @KafkaListener(topics = "myTopic")
5     public void consume(String message) {
6         System.out.println(message);
7     }
8 }
```

### 3.12 What is Kafka and how have you implemented it?

**Answer:** Distributed messaging system for event-driven architectures.

```
1 import org.springframework.kafka.core.*;
2 import org.springframework.stereotype.*;
3 @Service
4 public class KafkaProducer {
5     @Autowired
6     private KafkaTemplate<String, String> kafkaTemplate;
7     public void sendMessage(String msg) {
8         kafkaTemplate.send("myTopic", msg);
9     }
10 }
```

### 3.13 How have you used Spring Security in your project?

**Answer:** Secures applications via authentication/authorization.

```
1 import org.springframework.context.annotation.*;
2 import org.springframework.security.config.annotation.web.builders.*;
3 @Configuration
4 @EnableWebSecurity
5 public class SecurityConfig {
6     @Bean
7     public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
8             Exception {
9         http.authorizeRequests().anyRequest().authenticated().and().httpBasic()
10            ;
11         return http.build();
12     }
13 }
```

### 3.14 What is JWT security and how have you used it?

**Answer:** Token-based authentication using JSON Web Tokens.

```
1 import io.jsonwebtoken.*;
2 public class JwtExample {
3     public String generateToken(String username) {
4         return Jwts.builder()
5             .setSubject(username)
6             .signWith(SignatureAlgorithm.HS512, "secret")
7             .compact();
8     }
9 }
```

9 }

### 3.15 What are the things to consider while developing REST APIs?

**Answer:** RESTful principles, versioning, error handling, security, documentation.

```
1 import org.springframework.http.*;
2 import org.springframework.web.bind.annotation.*;
3 @RestController
4 @RequestMapping("/api/v1")
5 public class ApiController {
6     @GetMapping("/users")
7     public ResponseEntity<List<String>> getUsers() {
8         return ResponseEntity.ok(Arrays.asList("Alice", "Bob"));
9     }
10 }
```

### 3.16 Difference between PUT, POST, and PATCH

**Answer:** POST creates, PUT updates entire resource, PATCH updates partially.

```
1 import org.springframework.web.bind.annotation.*;
2 @RestController
3 public class UserController {
4     @PostMapping("/users") public String createUser() { return "Created"; }
5     @PutMapping("/users/{id}") public String updateUser(@PathVariable int id)
6         { return "Updated"; }
7     @PatchMapping("/users/{id}") public String patchUser(@PathVariable int id
8         ) { return "Patched"; }
9 }
```

### 3.17 API versioning strategies

**Answer:** URI versioning, query parameters, headers, media types.

```
1 import org.springframework.web.bind.annotation.*;
2 @RestController
3 @RequestMapping("/api/v1")
4 public class VersionedController {
5     @GetMapping("/users")
6     public String getUsers() { return "Version 1"; }
7 }
```

### 3.18 What is idempotency and why is it critical?

**Answer:** Ensures multiple identical requests have the same effect, critical for reliability.

```
1 import org.springframework.web.bind.annotation.*;
2 @RestController
3 public class IdempotentController {
4     @PutMapping("/users/{id}")
5     public ResponseEntity<String> updateUser(@PathVariable int id) {
6         return ResponseEntity.ok("Updated");
7     }
8 }
```

### 3.19 How do you achieve idempotency in microservices?

**Answer:** Use unique request IDs or design idempotent operations.

```
1 import java.util.*;
2 @Service
3 public class IdempotentService {
4     private Set<String> processedIds = new HashSet<>();
5     public String process(String requestId) {
6         if (processedIds.contains(requestId)) return "Already processed";
7         processedIds.add(requestId);
8         return "Processed";
9     }
10 }
```

### 3.20 Securing APIs using JWT and OAuth2

**Answer:** JWT for authentication, OAuth2 for authorization.

```
1 // See JWT example (Question 14)
```

### 3.21 Validating incoming payloads with annotations

**Answer:** Use @Valid and Bean Validation annotations.

```
1 import javax.validation.constraints.*;
2 public class User {
3     @NotNull private String name;
4 }
5 import org.springframework.web.bind.annotation.*;
6 import javax.validation.*;
7 @RestController
8 public class UserController {
9     @PostMapping("/users")
10    public ResponseEntity<String> createUser(@Valid @RequestBody User user) {
11        return ResponseEntity.ok("Valid");
12    }
13 }
```

### 3.22 What are the steps to test Spring Boot microservices applications?

**Answer:** Unit tests, integration tests, mock external services, test REST APIs.

```
1 import org.junit.jupiter.api.*;
2 import org.springframework.boot.test.context.*;
3 import org.springframework.boot.test.web.client.*;
4 @SpringBootTest
5 public class UserControllerTest {
6     @Autowired private TestRestTemplate restTemplate;
7     @Test
8     public void testGetUsers() {
9         ResponseEntity<String> response = restTemplate.getForEntity("/users",
10             String.class);
11         Assertions.assertEquals(HttpStatus.OK, response.getStatusCode());
12     }
13 }
```

### 3.23 How to handle exceptions in Spring Boot

**Answer:** Use @ExceptionHandler or @ControllerAdvice.

```
1 import org.springframework.http.*;
2 import org.springframework.web.bind.annotation.*;
3 @ControllerAdvice
4 public class GlobalExceptionHandler {
5     @ExceptionHandler(Exception.class)
6     public ResponseEntity<String> handleException(Exception e) {
7         return new ResponseEntity<>("Error: " + e.getMessage(), HttpStatus.
8             INTERNAL_SERVER_ERROR);
9     }
}
```

### 3.24 How to create global exceptions and what annotations are used

**Answer:** Use @ControllerAdvice and @ExceptionHandler.

```
1 // See above
```

### 3.25 How to exclude classes from component scan

**Answer:** Use exclude or excludeFilters in @ComponentScan.

```
1 import org.springframework.context.annotation.*;
2 @ComponentScan(basePackages = "com.example", excludeFilters = @Filter(type
3     = FilterType.ASSIGNABLE_TYPE, classes = MyClass.class))
4 @Configuration
5 public class AppConfig {}
```

### 3.26 How does component scan work?

**Answer:** Scans packages for stereotype-annotated classes, registering them as beans.

```
1 import org.springframework.stereotype.*;
2 @Component
3 public class MyComponent {}
```

### 3.27 What is the most challenging task you've done?

**Answer:** Optimizing a microservice for high load with caching.

```
1 import org.springframework.cache.annotation.*;
2 @Service
3 public class OptimizedService {
4     @Cacheable("data")
5     public String getData() { return "Data"; }
6 }
```

### 3.28 What are the top 3 performance bottlenecks in microservices?

**Answer:** Network latency, database queries, resource management.

```

1 import org.springframework.data.jpa.repository.*;
2 public interface UserRepository extends JpaRepository<User, Long> {
3     @Query("SELECT u FROM User u WHERE u.active = true")
4     List<User> findActiveUsers();
5 }

```

### 3.29 How do you monitor microservices?

**Answer:** Use Prometheus, Grafana, or Spring Actuator.

```

1 # application.properties
2 management.endpoints.web.exposure.include=*

```

### 3.30 How do you ensure system resiliency under high load?

**Answer:** Use circuit breakers, retries, load balancing, caching.

```

1 // See Circuit Breaker (Question 6)

```

### 3.31 What is centralized configuration and secrets management?

**Answer:** Centralized configuration (Spring Cloud Config); secrets (Vault).

```

1 # application.properties
2 spring.config.import=configserver:http://config-server

```

### 3.32 What is service discovery (Eureka/Consul)?

**Answer:** Dynamically locates services.

```

1 import org.springframework.cloud.netflix.eureka.*;
2 @SpringBootApplication
3 @EnableEurekaClient
4 public class Application {}

```

### 3.33 Inter-service communication: Feign vs RestTemplate vs WebClient

**Answer:** RestTemplate (synchronous), WebClient (reactive), Feign (declarative).

```

1 import feign.*;
2 @FeignClient(name = "user-service")
3 interface UserClient {
4     @GetMapping("/users")
5     List<String> getUsers();
6 }

```

### 3.34 Circuit Breaker and Retry: Resilience4j

**Answer:** Circuit Breaker prevents failures; Retry attempts failed operations.

```

1 import io.github.resilience4j.circuitbreaker.annotation.*;
2 import io.github.resilience4j.retry.annotation.*;
3 @Service
4 public class ResilientService {
5     @CircuitBreaker(name = "myService")

```

```
6  @Retry(name = "myService")
7  public String call() { return "Success"; }
8 }
```

### 3.35 Blue-green vs canary deployments

**Answer:** Blue-Green: Two environments, switch traffic. Canary: Gradual rollout.

### 3.36 Handling version mismatch between services

**Answer:** Use API versioning or backward compatibility.

```
1 // See API versioning (Question 17)
```

### 3.37 What are Maven commands you use daily?

**Answer:** mvn clean install, mvn test, mvn package.

```
1 mvn clean install
```

### 3.38 Have you used Docker? Benefits and challenges?

**Answer:** Benefits: consistency, portability. Challenges: resource overhead.

```
1 FROM openjdk:11
2 COPY target/myapp.jar /app.jar
3 ENTRYPOINT ["java", "-jar", "/app.jar"]
```

## 4 Spring & Bean Lifecycle

### 4.1 Bean lifecycle and Spring container

**Answer:** Creation, dependency injection, initialization, use, destruction.

```
1 import javax.annotation.*;
2 @Component
3 public class MyBean {
4     @PostConstruct
5     public void init() { System.out.println("Initialized"); }
6 }
```

### 4.2 Use of @Component, @Service, @Repository

**Answer:** @Component: generic, @Service: business logic, @Repository: data access.

```
1 import org.springframework.stereotype.*;
2 @Repository
3 public class UserRepository {}
```

### 4.3 Role of @ComponentScan, @Configuration, @Bean

**Answer:** @ComponentScan scans beans, @Configuration defines config, @Bean creates beans.

```
1 import org.springframework.context.annotation.*;
2 @Configuration
3 @ComponentScan("com.example")
4 public class AppConfig {
5     @Bean
6     public MyBean myBean() { return new MyBean(); }
7 }
```

### 4.4 What is Spring Boot auto-configuration?

**Answer:** Automatically configures beans based on dependencies.

```
1 import org.springframework.boot.autoconfigure.*;
2 @SpringBootApplication
3 public class Application {}
```

## 5 JPA & Database

### 5.1 What is Fetch Type (Lazy vs Eager Loading)?

**Answer:** Lazy loads data on demand; Eager loads immediately.

```
1 import javax.persistence.*;
2 @Entity
3 public class User {
4     @OneToMany(fetch = FetchType.LAZY)
5     private List<Order> orders;
6 }
```

### 5.2 Lazy vs Eager loading — real-time use cases

**Answer:** Lazy for large datasets; Eager for small, frequent data.

### 5.3 Complex entity relationship experience

**Answer:** Managed @OneToOne, @ManyToOne with cascading.

```
1 import javax.persistence.*;
2 @Entity
3 public class Department {
4     @OneToMany(mappedBy = "department")
5     private List<Employee> employees;
6 }
```

### 5.4 What is the N+1 query problem?

**Answer:** Multiple queries for related data due to lazy loading.

```
1 List<User> users = repository.findAll();
2 for (User u : users) { u.getOrders().size(); }
```

## 5.5 How to optimize N+1 using Spring Data JPA

**Answer:** Use @EntityGraph or JOIN FETCH.

```
1 import org.springframework.data.jpa.repository.*;
2 public interface UserRepository extends JpaRepository<User, Long> {
3     @EntityGraph(attributePaths = {"orders"})
4     List<User> findAll();
5 }
```

## 5.6 Difference between get() and load() in Hibernate

**Answer:** get() loads immediately, returns null if not found; load() uses proxy, throws exception if not found.

```
1 import org.hibernate.*;
2 public class HibernateExample {
3     public static void main(String[] args) {
4         Session session = sessionFactory.getCurrentSession();
5         User user = session.get(User.class, 1L);
6         User proxy = session.load(User.class, 1L);
7     }
8 }
```

## 5.7 Writing optimized JPQL and Criteria queries

```
1 import org.springframework.data.jpa.repository.*;
2 public interface UserRepository extends JpaRepository<User, Long> {
3     @Query("SELECT u FROM User u WHERE u.age > ?1")
4     List<User> findByAgeGreaterThan(int age);
5 }
```

## 5.8 Transaction management: @Transactional deep dive

**Answer:** Controls transaction boundaries, rolls back on exceptions.

```
1 import org.springframework.transaction.annotation.*;
2 @Transactional(rollbackOn = Exception.class)
3 public void saveUser(User user) {}
```

# 6 Testing & Mocking

## 6.1 Differences between stubbing and mocking

**Answer:** Stubbing provides canned responses; mocking verifies interactions.

```
1 import org.junit.jupiter.api.*;
2 import static org.mockito.Mockito.*;
3 public class MockTest {
4     @Test
5     public void testMock() {
6         List<String> mockedList = mock(List.class);
7         when(mockedList.get(0)).thenReturn("Stubbed");
8         verify(mockedList).get(0);
9     }
10 }
```

## 6.2 Why do we need both approaches?

**Answer:** Stubbing for outputs, mocking for behavior verification.

## 6.3 What is Spy in Mockito and when to use it?

**Answer:** Wraps real object, allows real calls unless overridden.

```
1 import org.junit.jupiter.api.*;
2 import static org.mockito.Mockito.*;
3 public class SpyTest {
4     @Test
5     public void testSpy() {
6         List<String> list = new ArrayList<>();
7         List<String> spy = spy(list);
8         spy.add("test");
9         verify(spy).add("test");
10    }
11 }
```

## 6.4 How to write JUnit test cases for static methods

**Answer:** Use PowerMock or refactor to instance methods.

```
1 public class StaticUtil {
2     public static String getName() { return "Test"; }
3 }
4 import org.junit.jupiter.api.*;
5 public class StaticTest {
6     @Test
7     public void testStatic() {
8         Assertions.assertEquals("Test", StaticUtil.getName());
9     }
10 }
```

## 6.5 What is the use of Mockito framework?

**Answer:** Creates mocks/spies for dependency-free testing.

```
1 import org.junit.jupiter.api.*;
2 import static org.mockito.Mockito.*;
3 public class MockitoTest {
4     @Test
5     public void testMockito() {
6         Service service = mock(Service.class);
7         when(service.getData()).thenReturn("Mocked");
8         Assertions.assertEquals("Mocked", service.getData());
9     }
10 }
```

# 7 Coding & Problem Solving

## 7.1 Reverse a string by preserving word position

```

1 public class StringReverse {
2     public static String reverseWords(String s) {
3         String[] words = s.split(" ");
4         StringBuilder result = new StringBuilder();
5         for (String word : words) {
6             result.append(new StringBuilder(word).reverse()).append(" ");
7         }
8         return result.toString().trim();
9     }
10    public static void main(String[] args) {
11        System.out.println(reverseWords("Hello World")); // olleH dlrow
12    }
13 }
```

## 7.2 Remove duplicates from string/array/list

```

1 import java.util.*;
2 import java.util.stream.*;
3 public class RemoveDuplicates {
4     public static void main(String[] args) {
5         List<Integer> numbers = Arrays.asList(1, 2, 2, 3);
6         List<Integer> unique = numbers.stream().distinct().collect(Collectors.
7             toList());
8         System.out.println(unique); // [1, 2, 3]
9     }
}
```

## 7.3 Longest substring without repeating characters

```

1 import java.util.*;
2 public class LongestSubstring {
3     public static int lengthOfLongestSubstring(String s) {
4         Set<Character> set = new HashSet<>();
5         int max = 0, i = 0, j = 0;
6         while (j < s.length()) {
7             if (!set.contains(s.charAt(j))) {
8                 set.add(s.charAt(j++));
9                 max = Math.max(max, set.size());
10            } else {
11                set.remove(s.charAt(i++));
12            }
13        }
14        return max;
15    }
16    public static void main(String[] args) {
17        System.out.println(lengthOfLongestSubstring("abcabcbb")); // 3
18    }
19 }
```

## 7.4 Check if two strings/numbers are palindrome

```

1 public class Palindrome {
2     public static boolean isPalindrome(String s) {
```

```

3     s = s.toLowerCase().replaceAll("[^a-zA-Z]", "");
4     int i = 0, j = s.length() - 1;
5     while (i < j) {
6         if (s.charAt(i++) != s.charAt(j--)) return false;
7     }
8     return true;
9 }
10 public static void main(String[] args) {
11     System.out.println(isPalindrome("A man, a plan, a canal: Panama")); // true
12 }
13 }
```

## 7.5 Check if two strings/numbers are anagram

```

1 public class Anagram {
2     public static boolean isAnagram(String s1, String s2) {
3         if (s1.length() != s2.length()) return false;
4         int[] count = new int[26];
5         for (char c : s1.toCharArray()) count[c - 'a']++;
6         for (char c : s2.toCharArray()) count[c - 'a']--;
7         for (int c : count) if (c != 0) return false;
8         return true;
9     }
10    public static void main(String[] args) {
11        System.out.println(isAnagram("listen", "silent")); // true
12    }
13 }
```

## 7.6 Sort an array/list/string

```

1 import java.util.*;
2 public class SortExample {
3     public static void main(String[] args) {
4         List<Integer> list = Arrays.asList(3, 1, 2);
5         Collections.sort(list);
6         System.out.println(list); // [1, 2, 3]
7     }
8 }
```

## 7.7 Count occurrence of characters in a string

```

1 import java.util.*;
2 import java.util.stream.*;
3 public class CharCount {
4     public static Map<Character, Integer> countChars(String s) {
5         return s.chars()
6             .mapToObj(c -> (char) c)
7             .collect(Collectors.groupingBy(c -> c, Collectors.counting()))
8             .entrySet().stream()
9             .collect(Collectors.toMap(Map.Entry::getKey, e -> e.getValue().
10                 intValue()));
11     }
12     public static void main(String[] args) {
```

```

12     System.out.println(countChars("hello")); // {h=1, e=1, l=2, o=1}
13 }
14 }
```

## 7.8 Print duplicate characters in a string

```

1 import java.util.*;
2 import java.util.stream.*;
3 public class DuplicateChars {
4     public static void printDuplicates(String s) {
5         Map<Character, Long> map = s.chars()
6             .mapToObj(c -> (char) c)
7             .collect(Collectors.groupingBy(c -> c, Collectors.counting()));
8         map.entrySet().stream()
9             .filter(e -> e.getValue() > 1)
10            .forEach(e -> System.out.println(e.getKey()));
11    }
12    public static void main(String[] args) {
13        printDuplicates("hello"); // l
14    }
15 }
```

## 7.9 Print only special characters in a string

```

1 public class SpecialChars {
2     public static void printSpecial(String s) {
3         s.chars()
4             .mapToObj(c -> (char) c)
5             .filter(c -> !Character.isLetterOrDigit(c))
6             .forEach(System.out::println);
7     }
8     public static void main(String[] args) {
9         printSpecial("Hello!@#"); // !, @, #
10    }
11 }
```

## 7.10 Reverse the given string

```

1 public class ReverseString {
2     public static String reverse(String s) {
3         return new StringBuilder(s).reverse().toString();
4     }
5     public static void main(String[] args) {
6         System.out.println(reverse("Hello")); // olleH
7     }
8 }
```

## 7.11 Print difference between two strings

```

1 public class StringDiff {
2     public static void printDiff(String s1, String s2) {
3         for (int i = 0; i < Math.min(s1.length(), s2.length()); i++) {
4             if (s1.charAt(i) != s2.charAt(i)) {
```

```

5     System.out.println("Diff at " + i + ": " + s1.charAt(i) + " vs " +
6         s2.charAt(i));
7 }
8 }
9 public static void main(String[] args) {
10    printDiff("hello", "heelo"); // Diff at 2: l vs e
11 }
12 }
```

## 7.12 Count words in a string

```

1 public class WordCount {
2     public static int countWords(String s) {
3         return s.trim().split("\\s+").length;
4     }
5     public static void main(String[] args) {
6         System.out.println(countWords("Hello World")); // 2
7     }
8 }
```

## 7.13 Find second highest element in an array

```

1 public class SecondHighest {
2     public static int findSecondHighest(int[] arr) {
3         int max = Integer.MIN_VALUE, second = Integer.MIN_VALUE;
4         for (int num : arr) {
5             if (num > max) {
6                 second = max; max = num;
7             } else if (num > second && num != max) {
8                 second = num;
9             }
10        }
11        return second;
12    }
13    public static void main(String[] args) {
14        System.out.println(findSecondHighest(new int[]{5, 3, 8, 1})); // 5
15    }
16 }
```

## 7.14 Find common elements of two arrays

```

1 import java.util.*;
2 import java.util.stream.*;
3 public class CommonElements {
4     public static List<Integer> findCommon(int[] arr1, int[] arr2) {
5         Set<Integer> set = Arrays.stream(arr1).boxed().collect(Collectors.toSet());
6         return Arrays.stream(arr2).filter(set::contains).boxed().collect(
7             Collectors.toList());
8     }
9     public static void main(String[] args) {
10        System.out.println(findCommon(new int[]{1, 2, 3}, new int[]{2, 3, 4}));
11        // [2, 3]
12 }
```

```
10    }
11 }
```

## 7.15 Decode string like a2b3c1 to aabbabc

```
1 public class DecodeString {
2     public static String decode(String s) {
3         StringBuilder result = new StringBuilder();
4         for (int i = 0; i < s.length(); i += 2) {
5             char c = s.charAt(i);
6             int count = Character.getNumericValue(s.charAt(i + 1));
7             result.append(String.valueOf(c).repeat(count));
8         }
9         return result.toString();
10    }
11    public static void main(String[] args) {
12        System.out.println(decode("a2b3c1")); // aabbabc
13    }
14 }
```

## 7.16 Check if a number is prime

```
1 public class PrimeCheck {
2     public static boolean isPrime(int n) {
3         if (n <= 1) return false;
4         for (int i = 2; i <= Math.sqrt(n); i++) {
5             if (n % i == 0) return false;
6         }
7         return true;
8     }
9     public static void main(String[] args) {
10        System.out.println(isPrime(7)); // true
11    }
12 }
```

## 7.17 Generate Fibonacci series

```
1 public class Fibonacci {
2     public static void printFibonacci(int n) {
3         int a = 0, b = 1;
4         for (int i = 0; i < n; i++) {
5             System.out.print(a + " ");
6             int next = a + b;
7             a = b; b = next;
8         }
9     }
10    public static void main(String[] args) {
11        printFibonacci(5); // 0 1 1 2 3
12    }
13 }
```

## 7.18 Find factorial of a number

```
1 public class Factorial {  
2     public static long factorial(int n) {  
3         if (n == 0) return 1;  
4         return n * factorial(n - 1);  
5     }  
6     public static void main(String[] args) {  
7         System.out.println(factorial(5)); // 120  
8     }  
9 }
```

## 7.19 Find min and max element in an array

```
1 public class MinMax {  
2     public static int[] findMinMax(int[] arr) {  
3         int min = arr[0], max = arr[0];  
4         for (int num : arr) {  
5             min = Math.min(min, num);  
6             max = Math.max(max, num);  
7         }  
8         return new int[]{min, max};  
9     }  
10    public static void main(String[] args) {  
11        int[] result = findMinMax(new int[]{3, 1, 4, 2});  
12        System.out.println("Min: " + result[0] + ", Max: " + result[1]); // Min  
13        : 1, Max: 4  
14    }  
15 }
```

## 7.20 Find max repeated word in a sentence

```
1 import java.util.*;  
2 import java.util.stream.*;  
3 public class MaxRepeatedWord {  
4     public static String findMaxRepeated(String s) {  
5         Map<String, Long> map = Arrays.stream(s.split("\\s+"))  
6             .collect(Collectors.groupingBy(w -> w, Collectors.counting()));  
7         return map.entrySet().stream()  
8             .max(Map.Entry.comparingByValue())  
9             .map(Map.Entry::getKey)  
10            .orElse("");  
11    }  
12    public static void main(String[] args) {  
13        System.out.println(findMaxRepeated("hello world hello")); // hello  
14    }  
15 }
```

## 7.21 Rotate array from k=2

```
1 public class RotateArray {  
2     public static void rotate(int[] arr, int k) {  
3         k = k % arr.length;  
4         reverse(arr, 0, arr.length - 1);
```

```

5     reverse(arr, 0, k - 1);
6     reverse(arr, k, arr.length - 1);
7 }
8 private static void reverse(int[] arr, int start, int end) {
9     while (start < end) {
10         int temp = arr[start];
11         arr[start++] = arr[end];
12         arr[end--] = temp;
13     }
14 }
15 public static void main(String[] args) {
16     int[] arr = {1, 2, 3, 4, 5};
17     rotate(arr, 2);
18     System.out.println(Arrays.toString(arr)); // [4, 5, 1, 2, 3]
19 }
20 }
```

## 7.22 Merge characters from two strings alternately

```

1 public class MergeStrings {
2     public static String mergeAlternately(String s1, String s2) {
3         StringBuilder result = new StringBuilder();
4         int i = 0;
5         while (i < s1.length() || i < s2.length()) {
6             if (i < s1.length()) result.append(s1.charAt(i));
7             if (i < s2.length()) result.append(s2.charAt(i));
8             i++;
9         }
10        return result.toString();
11    }
12    public static void main(String[] args) {
13        System.out.println(mergeAlternately("abc", "pqr")); // apbqcr
14    }
15 }
```

## 7.23 Find max repeated word in a sentence

```
1 // See Question 20
```

## 7.24 Modify and improve given code

**Answer:** Refactor for readability, efficiency, error handling.

```

1 public class BadCode {
2     public void print(String s) { System.out.println(s); }
3 }
4 public class ImprovedCode {
5     public void print(String s) {
6         if (s == null) throw new IllegalArgumentException("Input cannot be null");
7         System.out.println(s.trim());
8     }
9 }
```

## 7.25 Find a file in a subdirectory

```
1 import java.nio.file.*;
2 import java.io.*;
3 public class FileSearch {
4     public static void findFile(Path dir, String fileName) throws IOException
5         {
6             Files.walk(dir)
7                 .filter(p -> p.getFileName().toString().equals(fileName))
8                 .forEach(System.out::println);
9         }
10    public static void main(String[] args) throws IOException {
11        findFile(Paths.get("."), "test.txt");
12    }
13 }
```

## 7.26 Write a method to fetch employee details

```
1 import org.springframework.http.*;
2 import org.springframework.web.bind.annotation.*;
3 @RestController
4 public class EmployeeController {
5     @GetMapping("/employees/{id}")
6     public ResponseEntity<Employee> getEmployee(@PathVariable Long id) {
7         Employee emp = new Employee(id, "Alice");
8         return ResponseEntity.ok(emp);
9     }
10 }
11 class Employee {
12     Long id; String name;
13     Employee(Long id, String name) { this.id = id; this.name = name; }
14 }
```

## 8 Angular & Frontend

### 8.1 What is ng-content and how does content projection work?

**Answer:** ng-content projects content from parent to child components.

```
1 <!-- child.component.html -->
2 <ng-content></ng-content>
3 <!-- parent.component.html -->
4 <app-child><h1>Projected Content</h1></app-child>
```

### 8.2 Difference between @ViewChild and @ContentChild

**Answer:** @ViewChild accesses template elements; @ContentChild accesses projected content.

```
1 import { Component, ContentChild, ViewChild, ElementRef } from '@angular/
2   core';
3 @Component({
4   selector: 'app-child',
5   template: '<ng-content></ng-content>' })
```

```

6 export class ChildComponent {
7   @ContentChild('content') content: ElementRef;
8 }
9 @Component({
10   selector: 'app-parent',
11   template: '<app-child><div #content>Content</div></app-child>'
12 })
13 export class ParentComponent {
14   @ViewChild('view') view: ElementRef;
15 }

```

### 8.3 Explain Dependency Injection in Angular

**Answer:** Provides dependencies via constructor injection.

```

1 import { Injectable, Component } from '@angular/core';
2 @Injectable()
3 class MyService {}
4 @Component({})
5 export class MyComponent {
6   constructor(private service: MyService) {}
7 }

```

### 8.4 How does Angular load dynamic components?

**Answer:** Uses ComponentFactoryResolver and ViewContainerRef.

```

1 import { Component, ViewChild, ViewContainerRef, ComponentFactoryResolver }
2   from '@angular/core';
3 @Component({
4   selector: 'app-dynamic',
5   template: '<ng-container #container></ng-container>'
6 })
7 export class DynamicComponent {
8   @ViewChild('container', { read: ViewContainerRef }) container:
9     ViewContainerRef;
10  constructor(private resolver: ComponentFactoryResolver) {}
11  ngOnInit() {
12    const factory = this.resolver.resolveComponentFactory(
13      MyDynamicComponent);
14    this.container.createComponent(factory);
15  }
16 }

```

### 8.5 What is the PipeTransform interface?

**Answer:** Defines custom pipes for data transformation.

```

1 import { Pipe, PipeTransform } from '@angular/core';
2 @Pipe({ name: 'myPipe' })
3 export class MyPipe implements PipeTransform {
4   transform(value: string): string {
5     return value.toUpperCase();
6   }
7 }

```

## 8.6 How does Angular bootstrapping work via AppModule?

**Answer:** AppModule defines root module, bootstrapped via platformBrowserDynamic.

```
1 import { NgModule } from '@angular/core';
2 @NgModule({
3   declarations: [AppComponent],
4   bootstrap: [AppComponent]
5 })
6 export class AppModule {}
```

## 8.7 What is an HTTP interceptor?

**Answer:** Intercepts HTTP requests/responses for headers, error handling.

```
1 import { Injectable, HttpInterceptor, HttpRequest, HttpHandler } from '@angular/common/http';
2 @Injectable()
3 export class AuthInterceptor implements HttpInterceptor {
4   intercept(req: HttpRequest<any>, next: HttpHandler) {
5     const authReq = req.clone({ setHeaders: { Authorization: 'Bearer token' } });
6     return next.handle(authReq);
7   }
8 }
```

## 8.8 Common use cases of HTTP interceptors

**Answer:** Adding tokens, logging, error handling.

## 8.9 What is GraphQL and how does it compare to REST?

**Answer:** GraphQL: Query-based, flexible. REST: Fixed endpoints.

```
1 query {
2   user(id: 1) {
3     name
4   }
5 }
```

## 8.10 Use Apollo in Angular to fetch GraphQL data

```
1 import { Component } from '@angular/core';
2 import { Apollo } from 'apollo-angular';
3 import gql from 'graphql-tag';
4 @Component({})
5 export class UserComponent {
6   constructor(private apollo: Apollo) {
7     this.apollo.query({ query: gql`{ user(id: 1) { name } }` })
8       .subscribe(result => console.log(result.data));
9   }
10 }
```

## 8.11 Differences: BehaviorSubject, Subject, ReplaySubject

**Answer:** Subject: No initial value. BehaviorSubject: Initial value, emits last. ReplaySubject: Replays multiple values.

```
1 import { BehaviorSubject } from 'rxjs';
2 const subject = new BehaviorSubject('initial');
3 subject.subscribe(v => console.log(v)); // initial
4 subject.next('new'); // new
```

## 8.12 Explain: switchMap, mergeMap, concatMap, exhaustMap

**Answer:** switchMap cancels previous, mergeMap runs concurrently, concatMap sequential, exhaustMap ignores new until complete.

```
1 import { of } from 'rxjs';
2 import { switchMap } from 'rxjs/operators';
3 of(1, 2, 3).pipe(
4   switchMap(id => of(`User ${id}`))
5 ).subscribe(console.log); // User 3
```

## 8.13 Design a semantic HTML navigation menu

```
1 <nav>
2   <ul>
3     <li><a href="/">Home</a></li>
4     <li><a href="/about">About</a></li>
5   </ul>
6 </nav>
```

## 8.14 List 5 semantic HTML tags and their uses

**Answer:** header (page header), nav (navigation), main (main content), article (independent content), footer (page footer).

## 8.15 Fetch API data and display in a table

```
1 import { Component } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3 @Component({
4   template: `
5     <table>
6       <tr *ngFor="let user of users">
7         <td>{{ user.name }}</td>
8       </tr>
9     </table>
10   `
11 })
12 export class UserTableComponent {
13   users: any[] = [];
14   constructor(private http: HttpClient) {
15     this.http.get('https://api.example.com/users').subscribe(data => this.
16       users = data);
17 }
```

## 8.16 Search by name using Reactive Form

```
1 import { Component } from '@angular/core';
2 import { FormGroup, FormControl } from '@angular/forms';
3 @Component({
4   template: `
5     <form [formGroup]="form">
6       <input formControlName="name">
7     </form>
8     <div *ngFor="let user of filteredUsers">{{ user.name }}</div>
9   `
10 })
11 export class SearchComponent {
12   form = new FormGroup({ name: new FormControl('') });
13   users = [{ name: 'Alice' }, { name: 'Bob' }];
14   filteredUsers = this.users;
15   ngOnInit() {
16     this.form.get('name').valueChanges.subscribe(value => {
17       this.filteredUsers = this.users.filter(u => u.name.includes(value));
18     });
19   }
20 }
```

## 8.17 Error handling on API failure

```
1 import { Component } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3 import { of } from 'rxjs';
4 import { catchError } from 'rxjs/operators';
5 @Component({})
6 export class ErrorComponent {
7   constructor(private http: HttpClient) {
8     this.http.get('https://api.example.com/users')
9       .pipe(catchError(err => of([])))
10      .subscribe(data => console.log(data));
11   }
12 }
```

## 8.18 Flatten a nested array

```
1 const nested = [1, [2, 3], [4, [5]]];
2 const flat = nested.flat(Infinity);
3 console.log(flat); // [1, 2, 3, 4, 5]
```

# 9 Miscellaneous

## 9.1 What do you know about ISO8583?

**Answer:** Standard for financial transaction messaging.

## 9.2 How do Angular applications interact with backend APIs?

**Answer:** Via HttpClient for REST or Apollo for GraphQL.

```
1 // See Question 15, Section 8
```

### 9.3 How do you clone code from Git and commit changes?

**Answer:** Clone: git clone. Commit: git add, git commit, git push.

```
1 git clone https://github.com/repo.git
2 git add .
3 git commit -m "Add feature"
4 git push
```

### 9.4 How to resolve merge conflicts?

**Answer:** Edit conflicting files, mark resolved, commit.

```
1 git pull
2 # Resolve conflicts
3 git add .
4 git commit
```

### 9.5 Daily Git commands you use

**Answer:** git pull, git add, git commit, git push, git status.

## 10 Additional Frequently Asked Questions

### 10.1 What is the difference between checked and unchecked exceptions?

**Answer:** Checked: Compile-time, must handle (e.g., IOException). Unchecked: Runtime, optional (e.g., NullPointerException).

```
1 public class ExceptionExample {
2     public static void main(String[] args) {
3         try {
4             new FileReader("file.txt");
5         } catch (IOException e) {
6             System.out.println("Checked: " + e);
7         }
8     }
9 }
```

### 10.2 What is the purpose of @Autowired annotation in Spring?

**Answer:** Injects dependencies automatically.

```
1 import org.springframework.stereotype.*;
2 import org.springframework.beans.factory.annotation.*;
3 @Component
4 class MyComponent {
5     @Autowired
6     private MyService service;
7 }
```

### 10.3 What is the Circuit Breaker pattern in microservices?

**Answer:** Prevents cascading failures by stopping requests to failing services.

```
1 import io.github.resilience4j.circuitbreaker.annotation.*;
2 @Service
3 public class CircuitBreakerService {
4     @CircuitBreaker(name = "myService")
5     public String call() { return "Success"; }
6 }
```