

Java Prep

Q1. What are the features you know of in Java 8?

- **Lambda Expressions**
 1. Introduced functional-style programming to Java.
 2. Enables passing behavior as a parameter, making code more concise and readable.
 - 3.

```
List<String> names = Arrays.asList("John", "Jane", "Jack");  
names.forEach(name -> System.out.println(name));
```
- **Functional Interfaces**
 1. Interfaces with a single abstract method, used primarily with lambda expressions.
 2. Common ones include Runnable, Callable, Comparator, and Java 8's new Function, Predicate, Consumer, and Supplier.
- **Streams API**
 1. Allows processing collections in a functional and declarative way.
 2. Supports operations like map, filter, reduce, collect, etc.
 3. Great for working with large datasets and parallel processing.
- **Default and Static Methods in Interfaces**
 1. Interfaces can now have method implementations using default and static keywords.
 2. Helps in evolving APIs without breaking existing implementations.
- **Method References and Constructor References**
 1. A shorthand notation for calling methods or constructors using ::
 2. Example: List::size or String::toUpperCase.
- **Optional Class**
 1. A container object which may or may not contain a non-null value.
 2. Helps in avoiding NullPointerException and encourages better null handling.
- **New Date and Time API (java.time package)**
 1. A much-improved, immutable, and thread-safe alternative to the old Date and Calendar classes.
 2. Includes classes like LocalDate, LocalTime, LocalDateTime, ZonedDateTime, and DateTimeFormatter.
- **Collectors and Collectors Utility**
 1. Used with streams to accumulate elements into collections, strings, maps, etc.
 2. Example: Collectors.toList(), Collectors.groupingBy(), Collectors.joining().

I've used these features in various enterprise applications, especially lambda expressions and streams for data processing, and the new Date/Time API for handling time zones and scheduling. I also ensure to use Optional to write more robust and null-safe code.

Q2. Do you know about the functional interfaces in Java 8? Which ones?

Java 8 introduced several functional interfaces in the `java.util.function` package. These are interfaces with a single abstract method, enabling lambda expressions and method references.

◆ Core Functional Interfaces:

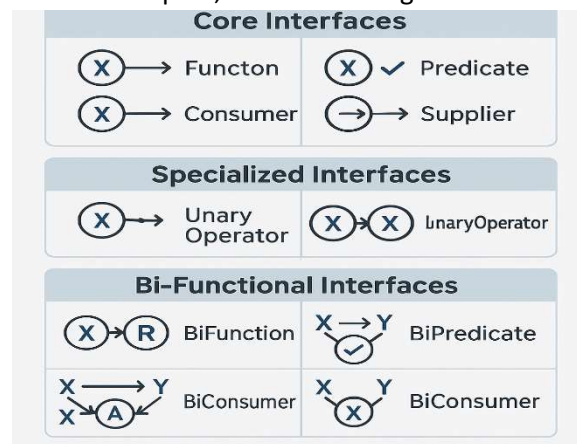
- **Function<T, R>**
 1. Takes one input of type T, returns result of type R.
 2. Example: `Function<String, Integer> length = str -> str.length();`
- **Predicate**
 1. Takes one input, returns a boolean.
 2. Example: `Predicate<String> isEmpty = str -> str.isEmpty();`
- **Consumer**
 1. Takes one input, returns nothing.
 2. Example: `Consumer<String> print = str -> System.out.println(str);`
- **Supplier**
 1. Takes no input, returns a result.
 2. Example: `Supplier<Double> random = () -> Math.random();`

◆ Specialized Functional Interfaces:

- **UnaryOperator**
 1. A `Function<T, T>` — input and output are of same type.
 2. Example: `UnaryOperator<String> toUpper = str -> str.toUpperCase();`
- **BinaryOperator**
 1. A `BiFunction<T, T, T>` — two inputs, one output of same type.
 2. Example: `BinaryOperator<Integer> sum = (a, b) -> a + b;`

◆ Bi-Functional Interfaces:

- **BiFunction<T, U, R>**
 1. Takes two inputs of types T and U, returns result of type R.
- **BiPredicate<T, U>**
 1. Takes two inputs, returns a boolean.
- **BiConsumer<T, U>**
 1. Takes two inputs, returns nothing.



Q3. what the difference between the supplier and the consumer?

Feature	Supplier	Consumer
Purpose	Supplies a value (produces data)	Consumes a value (performs an action)
Input	Takes no input	Takes one input of type T
Output	Returns a value of type T	Returns nothing (void)
Use Case	Used when you need to generate or fetch a value	Used when you need to process or act on a value
Example	<code>Supplier<String> s = () -> "Hello";</code>	<code>Consumer<String> c = str -> System.out.println(str);</code>

Q4. Default methods vs static methods? Where and why did you use them?

◆ Default Methods

- Introduced in Java 8 to allow method implementation in interfaces.
- Use the default keyword.
- Can be overridden by implementing classes.
- Helps in interface evolution without breaking existing implementations.

```
1 interface MyInterface {
2     default void show() {
3         System.out.println("Default implementation");
4     }
5 }
6
```

◆ Static Methods

- Also introduced in Java 8 for interfaces.
- Use the static keyword.
- Cannot be overridden.
- Called using the interface name, not the implementing class.
- Useful for utility or helper methods related to the interface.

```
1 interface MyInterface {
2     static void log(String msg) {
3         System.out.println("Log: " + msg);
4     }
5 }
6
```

🧠 Where and Why I Used Them

✅ Default Methods

- Used in **API design** to add new methods to interfaces without breaking existing code.
- Example: In a logging framework, added a default `logDebug()` method to the `Logger` interface.

✅ Static Methods

- Used for **common utility functions** like validation, formatting, or logging.
- Example: In a `Validator` interface, added static `isValidEmail(String email)` to centralize validation logic.

Q5. Can you override any static methods?

No, you cannot override static methods in Java.

- Static methods belong to the class, not to instances of the class.
 - Method overriding is a concept that applies to instance methods, where the method behavior is determined at runtime based on the object's actual type.
 - Static methods are resolved at compile time using the reference type, not the object type.
- You can **hide** a static method in a subclass (this is called **method hiding**), but you **cannot override** it.

Q6. What is `filter()` and its uses

- `filter()` is a Stream intermediate operation.
 - It takes a Predicate as input and returns a stream with elements that match the condition.
 - It's used to select elements based on a condition.
- ◆ **Where I Used It:**
1. Filtering data from collections
Example: Filtering active users from a list of user objects.
 2. Validating input data
Example: Filtering out null or empty strings before processing.
 3. Working with APIs and DTOs
Example: Filtering records based on status or date before mapping to DTOs.
 4. In microservices
Used in service layers to filter business-relevant data before sending to clients.

Q7. Explain Optional classes. Have you used them? Where?

- Optional is a container object that may or may not contain a non-null value.
- It helps in avoiding **NullPointerException** and encourages better null handling.
- Introduced in `java.util` package.
◆ Common Methods in Optional: **Optional.of()** , **Optional.ofNullable()**, **Optional.empty()**, **isPresent()** / **ifPresent()**

Q8. You have a list of strings in that list of string, you must be having the names. What you need to do is you have to filter the list of strings that starts with "A". ?

```
public class FilterNames {  
    public static void main(String[] args) {  
        List<String> names = Arrays.asList("Alice", "Bob", "Ankit", "Steve", "Aman",  
        "Ravi");  
  
        List<String> filteredNames = names.stream()  
            .filter(name -> name.startsWith("A"))  
            .collect(Collectors.toList());  
  
        System.out.println("Names starting with A: " + filteredNames);  
    }  
}
```

Q9. If we have a class of employee. In an employee class we have employee name and its salary, and it can also keep the ID. So, we need to find the average salaries of all the employees.

```
1  public class Employee {  
2      private String name;  
3      private double salary;  
4      private int id;  
5  
6      // Constructor  
7      public Employee(String name, double salary, int id) {  
8          this.name = name;  
9          this.salary = salary;  
10         this.id = id;  
11     }  
12  
13     // Getters  
14     public double getSalary() {  
15         return salary;  
16     }  
17 }
```

```
1  import java.util.Arrays;  
2  import java.util.List;  
3  
4  public class AverageSalary {  
5      public static void main(String[] args) {  
6          List<Employee> employees = Arrays.asList(  
7              new Employee("Alice", 50000, 101),  
8              new Employee("Bob", 60000, 102),  
9              new Employee("Charlie", 55000, 103)  
10         );  
11  
12         double averageSalary = employees.stream()  
13             .mapToDouble(Employee::getSalary)  
14             .average()  
15             .orElse(0.0);  
16  
17         System.out.println("Average Salary: " + averageSalary);  
18     }
```

Q10. what is dependency injection in spring?

Definition: Design pattern where Spring injects required dependencies into a class automatically.

Purpose: Promotes loose coupling, testability, and maintainability.

Spring Container: Manages object creation and wiring.

◆ Types of Dependency Injection:

- Constructor Injection
Dependencies passed via constructor.
Recommended for mandatory dependencies.
- Setter Injection
Dependencies set via setter methods.
Suitable for optional dependencies.
- Field Injection
Dependencies injected directly into fields.
Quick but less testable.

Q11. How can we achieve the field injection in a class?

Definition: Injecting dependencies directly into class fields using the `@Autowired` annotation.

Annotation Used: `@Autowired`

Spring Requirement: The class must be a Spring-managed bean (`@Component`, `@Service`, etc.).

◆ How It Works:

Spring scans the class and sees `@Autowired`.

It injects the appropriate bean (`EmployeeRepository`) into the field.

No need for constructor or setter methods.

```
1  @Service
2  public class EmployeeService {
3
4      @Autowired
5      private EmployeeRepository repository;
6
7      public void showData() {
8          System.out.println(repository.getData());
9      }
10 }
```

Q12. What is Bean scopes? Explain. What is the most frequently used scope of a bean that you are using in your project?

Bean scope defines the lifecycle and visibility of a bean — i.e., how many instances of a bean Spring creates and how long they live. It controls how Spring manages object instances in the application context.

◆ 1. Singleton (Default)

- One instance per Spring container.
- Shared across the application.
- Most commonly used scope.
- Example: `@Scope("singleton")`

◆ 2. Prototype

- New instance every time the bean is requested.
- Useful for stateful beans.
- Example: `@Scope("prototype")`

◆ 3. Request (Web only)

- One bean per HTTP request.
- Used in web applications.
- Example: `@Scope("request")`

◆ 4. Session (Web only)

- One bean per HTTP session.
- Example: `@Scope("session")`

◆ 5. Application (Web only)

- One bean per ServletContext.
- Shared across the entire application lifecycle.
- Example: `@Scope("application")`

✦ Most Frequently Used Scope in My Project: **Singleton**

Used for services, repositories, and controllers.

Ensures consistent behavior and efficient resource usage.

```
1 import org.springframework.stereotype.Component;
2
3 @Component // Automatically makes this a singleton bean
4 public class MySingletonBean {
5
6     public void showMessage() {
7         System.out.println("This is a singleton bean.");
8     }
}
```

Used in another class

```
@Service
public class MyService {

    @Autowired
    private MySingletonBean singletonBean;

    public void useBean() {
        singletonBean.showMessage();
    }
}
```

◆ Key Points:

- @Component by default creates a singleton bean.
- Spring maintains one instance of MySingletonBean throughout the application context.
- No need to explicitly specify @Scope("singleton") unless you want to be explicit.

Q13. what is SOLID Principal?

The **SOLID principles** are a set of design guidelines in object-oriented programming that help create maintainable, scalable, and robust software.

1. Single Responsibility Principle (SRP)

- **Definition:** A class should have only one reason to change, meaning it should have only one responsibility.

```
class InvoicePrinter {
    public void print(Invoice invoice) {
        // printing Logic
    }
}

class InvoiceSaver {
    public void save(Invoice invoice) {
        // saving Logic
    }
}
```

2. Open/Closed Principle (OCP)

- **Definition:** Classes should be open for extension but closed for modification.

```
1 interface Discount {
2     double apply(double price);
3 }
4
5 class ChristmasDiscount implements Discount {
6     public double apply(double price) {
7         return price * 0.9;
8     }
9 }
10
11 class NewYearDiscount implements Discount {
12     public double apply(double price) {
13         return price * 0.85;
14     }
15 }
```

- Add new discount types without changing existing code.

3. Liskov Substitution Principle (LSP)

- **Definition:** Subtypes must be substitutable for their base types without altering the correctness of the program.

```
1 class Bird {
2     public void eat() {}
3 }
4
5 class Sparrow extends Bird {
6     public void eat() {} // valid substitution
7 }
8
9 class Penguin extends Bird {
10    public void eat() {} // valid substitution
11    // no fly() method to avoid breaking LSP
12 }
```

-
- Avoid overriding behavior that breaks expectations.

4. Interface Segregation Principle (ISP)

- **Definition:** A class should not be forced to implement interfaces it doesn't use.

```
interface Printer {
    void print();
}

interface Scanner {
    void scan();
}

class MultiFunctionPrinter implements Printer, Scanner {
    public void print() {}
    public void scan() {}
}
```

-

Separate interfaces ensure classes implement only what they need.

5. Dependency Inversion Principle (DIP)

- **Definition:** High-level modules should not depend on low-level modules. Both should depend on abstractions.
-

Copy the code

```
interface NotificationService {
    void sendNotification(String message);
```

```

}

class EmailService implements NotificationService {
    public void sendNotification(String message) { /* logic */ }
}

class NotificationManager {
    private NotificationService service;

    NotificationManager(NotificationService service) {
        this.service = service;
    }

    void notify(String message) {
        service.sendNotification(message);
    }
}

```

Here, `NotificationManager` depends on the abstraction `NotificationService`, not the concrete `EmailService`.

14. How do you handle exceptions in your applications (Java and Spring Boot)?

Java:

- Use try-catch-finally blocks to handle exceptions.
- Create custom exceptions by extending `Exception` OR `RuntimeException`.
- Use `throws` keyword to propagate exceptions.

Spring Boot:

- Use `@ControllerAdvice` with `@ExceptionHandler` to handle exceptions globally.
- Example:

```

1 @ControllerAdvice
2 public class GlobalExceptionHandler {
3     @ExceptionHandler(ResourceNotFoundException.class)
4     public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {
5         return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
6     }
7 }
8

```

15. Have you used inheritance in your project? Use case? Does it resemble any SOLID principle?

Yes, inheritance is used for:

- Reusing common logic across classes.
- Creating base controller/service classes.

Use Case:

- A BaseEntity class with common fields like id, createdAt, etc., extended by all entity classes.

SOLID Principle:

- Resembles **Open/Closed Principle**: Classes are open for extension but closed for modification.

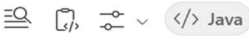
16. What is immutability? How can we make a class immutable?

Immutability means the object's state cannot change after it's created.

To make a class immutable:

- Declare class as `final`.
- Make fields `private` and `final`.
- No setters.
- Initialize fields via constructor.
- Return copies of mutable fields in getters.

Example:



```

1 public final class Employee {
2     private final String name;
3     private final int id;
4
5     public Employee(String name, int id) {
6         this.name = name;
7         this.id = id;
8     }
9
10    public String getName() { return name; }
11    public int getId() { return id; }
12 }
13

```

17. Explain Lombok, Jackson, and Guava. Which annotations have you used?

Lombok:

- Reduces boilerplate code.
- Common annotations:
 - @Getter, @Setter
 - @ToString, @EqualsAndHashCode
 - @Data, @Builder
 - @RequiredArgsConstructor

Jackson:

- Used for JSON serialization/deserialization.
- Common annotations:
 - @JsonProperty, @JsonIgnore
 - @JsonInclude, @JsonFormat
 - @JsonCreator

Guava:

- Google's core libraries for Java.
- Provides utilities for collections, caching, string manipulation, etc.
- Common usages:
 - ImmutableList, Optional, Preconditions

18. RequiredArgsConstructor – When and how to use it?

When to use:

- When you want a constructor with required (final or @NonNull) fields only.

How to use:

```
1 @RequiredArgsConstructor
2 public class UserService {
3     private final UserRepository userRepository;
4 }
5
```

Lombok generates a constructor with UserRepository as a parameter.

19. How can we achieve serialization? How to make a class serializable?

Serialization is converting an object into a byte stream.

To make a class serializable:

- Implement Serializable interface.
- Add serialVersionUID for version control.

Example:

```
1 public class Employee implements Serializable {
2     private static final long serialVersionUID = 1L;
3     private String name;
4     private int id;
5 }
6
```

20. What are all sorting techniques? Explain with example.

Common Sorting Techniques:

1. **Bubble Sort** – Compare adjacent elements and swap.
2. **Selection Sort** – Select the minimum and place it at the beginning.
3. **Insertion Sort** – Insert elements in the correct position.
4. **Merge Sort** – Divide and conquer, merge sorted halves.
5. **Quick Sort** – Partition and recursively sort.
6. **Heap Sort** – Use heap data structure.

Example – Quick Sort:

```
1 public void quickSort(int[] arr, int low, int high) {
2     if (low < high) {
3         int pi = partition(arr, low, high);
4         quickSort(arr, low, pi - 1);
5         quickSort(arr, pi + 1, high);
6     }
7 }
8
```

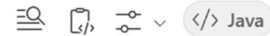
21. How do you sort a list of strings?

Using Java Collections:

```
1 List<String> names = Arrays.asList("John", "Alice", "Bob");
2 Collections.sort(names);
3
```

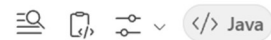
Using Streams:

```
1 List<String> sortedNames = names.stream()
2   .sorted()
3   .collect(Collectors.toList());
4
```



Custom Comparator:

```
1 Collections.sort(names, (a, b) -> b.compareTo(a)); // Descending
2
```



22. How does HashMap work internally? How are hash collisions handled?

Internal Working:

- Uses an array of buckets.
- Each bucket is a linked list or tree (after Java 8).
- Key's hashCode() is used to determine bucket index.

Collision Handling:

- If multiple keys map to the same bucket, they are stored in a linked list or tree.
- Tree is used when bucket size exceeds threshold (default: 8).

23. What is the load factor in HashMap? What is the resizing strategy?

Load Factor:

- Default is **0.75**.
- It determines when to resize the map.
- Resize occurs when `size >= capacity * load factor`.

Resizing Strategy:

- Capacity is doubled.
- All entries are rehashed and redistributed.

24. What is the default capacity of HashMap?

- Default initial capacity is **16**.
- It grows as needed based on the load factor.

25. HashMap vs HashSet?

Feature	HashMap	HashSet
Stores	Key-Value pairs	Unique elements
Backed by	Array of buckets	Internally uses HashMap
Nulls	One null key, multiple null values	One null element
Performance	O(1) for get/put	O(1) for add/contains

26. Internally, HashSet uses which data structure?

- **HashSet** uses a **HashMap** internally.
- Each element is stored as a key in the map with a dummy value.

27. What is the complexity of adding an element to HashMap?

- **Average case:** $O(1)$
- **Worst case (collision with tree):** $O(\log n)$
- **During resize:** $O(n)$

28. Fail-fast vs Fail-safe iterators?

Feature	Fail-fast	Fail-safe
Behavior	Throws <code>ConcurrentModificationException</code>	Doesn't throw exception
Examples	<code>ArrayList</code> , <code>HashMap</code> iterator	<code>CopyOnWriteArrayList</code> , <code>ConcurrentHashMap</code>
Mechanism	Checks <code>modCount</code>	Works on a clone or snapshot

29. `ConcurrentModificationException` – When and how to fix it?

Occurs When:

- Collection is modified while iterating using fail-fast iterator.

Fixes:

- Use `Iterator.remove()` instead of `Collection.remove()`.
- Use `CopyOnWriteArrayList` Or `ConcurrentHashMap`.
- Use `forEach` with lambda (Java 8+).

30. How does `ConcurrentHashMap` work internally?

- Uses **segments** (Java 7) or **bucket-level locking** (Java 8+).

- Allows concurrent read and write.
- Uses **CAS (Compare-And-Swap)** for atomic operations.
- No locking for read operations.

31. When do we use generics? Give a use case.

Generics allow type safety and code reusability.

Use Case:

Creating a generic repository:

```
1 public class Repository<T> {  
2     private List<T> items = new ArrayList<>();  
3     public void add(T item) { items.add(item); }  
4     public T get(int index) { return items.get(index); }  
5 }  
6
```

Benefits:

- Compile-time type checking.
- Avoids casting.
- Reusable for multiple types.

32. Gang of Four (GoF) Design Patterns – Categories and Examples

Creational Patterns:

- Deal with object creation.
- Examples:
 - Singleton
 - Factory Method
 - Abstract Factory
 - Builder
 - Prototype

Structural Patterns:

- Deal with object composition.
- Examples:
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Proxy

Behavioral Patterns:

- Deal with object interaction.
- Examples:
- Observer
- Strategy
- Command
- Template Method
- State
- Chain of Responsibility

33. Write a Singleton class with lazy initialization, thread safety, and protection against reflection, serialization, and cloning

```
1 public final class Singleton implements Serializable {
2     private static final long serialVersionUID = 1L;
3
4     private static class Holder {
5         private static final Singleton INSTANCE = new Singleton();
6     }
7
8     private Singleton() {
9         if (Holder.INSTANCE != null) {
10             throw new RuntimeException("Use getInstance()");
11         }
12     }
13 }
```

```

13
14     public static Singleton getInstance() {
15         return Holder.INSTANCE;
16     }
17
18     // Prevent cloning
19     @Override
20     protected Object clone() throws CloneNotSupportedException {
21         throw new CloneNotSupportedException();
22     }
23
24     // Prevent serialization
25     protected Object readResolve() {
26         return getInstance();
27     }
28 }

```

34. Explain Multithreading and Executor Framework with Example

Multithreading allows concurrent execution of tasks.

Executor Framework:

- Manages thread pools.
- Example:

```

1  ExecutorService executor = Executors.newFixedThreadPool(5);
2  executor.submit(() -> System.out.println("Task executed"));
3  executor.shutdown();
4

```

Benefits:

- Better resource management.
- Avoids manual thread creation.

35. What is a thread?

A **thread** is the smallest unit of execution in a process.

In Java:

- Can be created by extending Thread or implementing Runnable.

```
1 class MyThread extends Thread {  
2     public void run() {  
3         System.out.println("Running thread");  
4     }  
5 }  
6
```

36. How can we create a thread pool?

Using Executors class:

```
1 ExecutorService pool = Executors.newFixedThreadPool(10);  
2
```

Other types:

- newCachedThreadPool()
- newSingleThreadExecutor()
- newScheduledThreadPool(int corePoolSize)

37. Have you heard about submit and execute methods?

Yes.

- execute(Runnable command) – returns void.
- submit(Callable or Runnable) – returns Future.

Use submit when you need a result or exception handling.

38. What is Callable and Runnable?

- **Runnable:** No return value, no checked exception.

```
1 Runnable task = () -> System.out.println("Runnable");  
2
```

Callable: Returns a value and can throw exceptions.

```
1 Callable<String> task = () -> "Callable Result";  
2
```

39. What is ReentrantLock?

- A lock that can be acquired multiple times by the same thread.
- Provides more control than synchronized.

Example:

```
1 ReentrantLock lock = new ReentrantLock();
2 lock.lock();
3 try {
4     // critical section
5 } finally {
6     lock.unlock();
7 }
8
```

40. Explain Kafka – Why are you using Kafka in your project? Why not other MQs?

Kafka is a distributed event streaming platform.

Why Kafka:

- High throughput and scalability.
- Durable and fault-tolerant.
- Supports real-time processing.

Use Case in Project:

- For asynchronous communication between microservices.
- Event-driven architecture.

Why not others (e.g., RabbitMQ):

- Kafka is better for high-volume, persistent logs.
- RabbitMQ is good for low-latency messaging but not ideal for stream processing.

41. What is the benefit of Kafka over others?

Kafka Advantages:

- **High Throughput:** Handles millions of messages per second.
- **Scalability:** Easily scales horizontally.
- **Durability:** Messages are persisted on disk.
- **Fault Tolerance:** Replication across brokers.
- **Stream Processing:** Native support via Kafka Streams and integration with Apache Flink/Spark.
- **Decoupling:** Producers and consumers are loosely coupled.

Compared to others (e.g., RabbitMQ, ActiveMQ):

- Kafka is better for **event sourcing**, **log aggregation**, and **real-time analytics**.
- RabbitMQ is better for **low-latency messaging** and **complex routing**.

42. Explain Docker and Kubernetes. How do you set up Kubernetes clusters?

Docker:

- Containerization platform.
- Packages applications with dependencies into containers.

Kubernetes (K8s):

- Container orchestration platform.
- Manages deployment, scaling, and operations of containers.

Setting up Kubernetes Cluster:

1. **Minikube** for local setup.
2. **Kubeadm** for manual setup.
3. **Managed Services** like GKE (Google), EKS (AWS), AKS (Azure).
4. Define:
 - Deployment for app lifecycle.
 - Service for networking.
 - Ingress for routing.
 - ConfigMap and Secrets for configuration.

43. Explain JProfiler

JProfiler is a Java profiling tool used to analyze:

- **Memory usage**
- **CPU performance**
- **Thread behavior**
- **Garbage collection**

Use Cases:

- Detect memory leaks.
- Optimize performance bottlenecks.
- Analyze thread contention.

Integration:

- Can be attached to running JVM.
- Supports remote profiling.

44. Difference between LEFT JOIN and RIGHT OUTER JOIN

Feature	LEFT JOIN	RIGHT OUTER JOIN
Base Table	Left table	Right table
Result	All rows from left + matched right	All rows from right + matched left
Nulls	Nulls for unmatched right rows	Nulls for unmatched left rows

Example:

```

1 SELECT * FROM A LEFT JOIN B ON A.id = B.a_id;
2 SELECT * FROM A RIGHT JOIN B ON A.id = B.a_id;
3

```

🔍 📋 ⚙️ </> SQL

45. How do you connect to other microservices to fetch data?

Common Approaches:

- **REST APIs** using RestTemplate or FeignClient.
- **Message Queues** like Kafka or RabbitMQ for async communication.
- **Service Discovery** via Eureka or Consul.
- **API Gateway** for routing and security.

46. RestTemplate vs FeignClient

Feature	RestTemplate	FeignClient
Type	Low-level HTTP client	Declarative HTTP client
Configuration	Manual	Auto-configured with Spring Cloud
Readability	Verbose	Clean and concise
Error Handling	Manual	Built-in

```
1 @FeignClient(name = "user-service")
2 public interface UserClient {
3     @GetMapping("/users/{id}")
4     User getUser(@PathVariable Long id);
5 }
6
```

47. Explain JUnit, Mockito, Spy vs Mock

JUnit:

- Unit testing framework.
- Annotations: @Test, @BeforeEach, @AfterEach, @Disabled

Mockito:

- Mocking framework.
- Used to simulate dependencies.

Mock vs Spy:

- @Mock: Creates a dummy object.
- @Spy: Wraps a real object but allows stubbing.

Example:

```
1 @Mock
2 UserService userService;
3
4 @Spy
5 List<String> spyList = new ArrayList<>();
6
```

48. @Mock vs @InjectMocks

Annotation	Purpose
@Mock	Creates mock object
@InjectMocks	Injects mocks into the tested class

Example:

```
1 @Mock
2 UserRepository userRepository;
3
4 @InjectMocks
5 UserService userService;
6
```

49. Steps to fix a bug in production

1. **Identify:** Use logs, monitoring tools (e.g., ELK, Prometheus).
2. **Reproduce:** Try to replicate in staging.
3. **Analyze:** Debug code, check recent changes.

4. **Fix:** Apply patch or hotfix.
5. **Test:** Unit, integration, regression testing.
6. **Deploy:** Use CI/CD pipeline.
7. **Monitor:** Post-deployment monitoring.
8. **Document:** Root cause analysis and resolution steps.

50. Garbage Collection – Explain G1GC Algorithm

G1GC (Garbage First GC):

- Designed for large heap sizes.
- Divides heap into regions.
- Prioritizes regions with most garbage.
- Performs concurrent marking and compaction.

Phases:

1. **Initial Mark**
2. **Concurrent Mark**
3. **Remark**
4. **Cleanup**

Benefits:

- Predictable pause times.
- Efficient memory management.

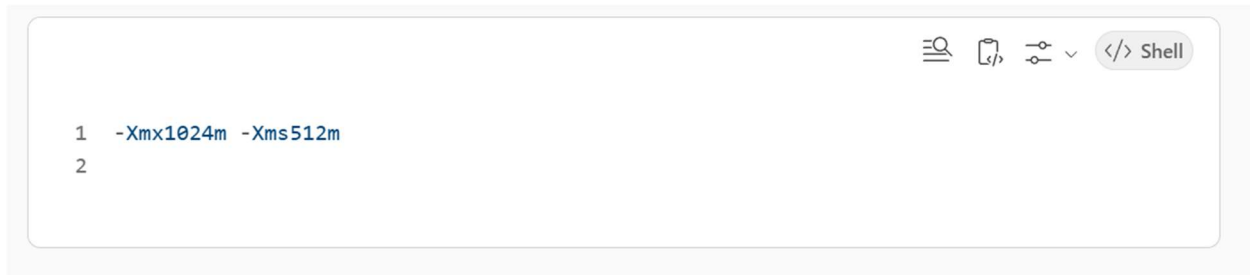
51. OutOfMemoryError – How to resolve it?

Causes:

- Memory leak.
- Large data processing.
- Inefficient object creation.

Resolution Steps:

1. **Analyze heap dump** using tools like JVisualVM or JProfiler.
2. **Increase heap size:**

A screenshot of a code editor window. The editor has a light gray background and a dark border. In the top right corner, there are icons for search, file explorer, and settings, followed by a button labeled 'Shell'. The code area contains two lines of text: line 1 is '-Xmx1024m -Xms512m' and line 2 is empty.

```
1 -Xmx1024m -Xms512m
2
```

1. **Optimize code:**
 - Avoid unnecessary object retention.
 - Use streaming for large files.
2. **Use caching wisely.**
3. **Monitor GC logs** for frequent collections.

52. Different memory areas in JVM

1. **Heap** – Stores objects and class instances.
2. **Stack** – Stores method calls and local variables.
3. **Method Area** – Stores class metadata, static variables.
4. **Program Counter (PC) Register** – Tracks current instruction.
5. **Native Method Stack** – For native method calls.

53. Caching mechanisms in Java applications

Types of Caching:

1. **In-Memory Caching:**
 - Using ConcurrentHashMap, Guava Cache, Or Caffeine.
2. **Distributed Caching:**
 - Redis, Hazelcast, Ehcache.
3. **Spring Cache Abstraction:**
 - Annotations: @Cacheable, @CachePut, @CacheEvict.

Example:



```

1 @Cacheable("users")
2 public User getUserById(Long id) {
3     return userRepository.findById(id);
4 }
5

```

54. Securing Java applications – OAuth, JWT, Spring Security

Spring Security:

- Provides authentication and authorization.
- Supports form login, basic auth, OAuth2, JWT.

JWT (JSON Web Token):

- Stateless authentication.
- Token contains user info and expiry.

OAuth2:

- Delegated authorization.
- Used for third-party login (Google, Facebook).

Best Practices:

- Use HTTPS.
- Store secrets securely.
- Validate tokens.
- Use roles and permissions.

55. Batch processing in Java applications

Approaches:

1. Spring Batch:

- Handles large volume data processing.
- Components: Job, Step, ItemReader, ItemProcessor, ItemWriter.

2. Quartz Scheduler:

- For scheduled jobs.

3. Manual Threads/Executors:

- For custom batch logic.

Spring Batch Example:

```
1 @Bean
2 public Job job(JobBuilderFactory jobBuilderFactory, Step step) {
3     return jobBuilderFactory.get("batchJob")
4         .start(step)
5         .build();
6 }
7
```

56. REST Principles and Best Practices

REST Principles:

- **Statelessness:** No client context stored on server.
- **Uniform Interface:** Standard HTTP methods (GET, POST, PUT, DELETE).
- **Resource-Based:** Use nouns in URIs.
- **Representation:** JSON/XML responses.
- **Client-Server:** Separation of concerns.

Best Practices:

- Use **HTTPS**.
- Use **versioning** (/api/v1/).
- Return proper **status codes**.
- Use **pagination** for large data.
- Validate input and handle errors gracefully.

57. API Gateway – APIM, Rate Limiting, Config Experience

API Gateway:

- Entry point for microservices.
- Handles routing, security, rate limiting, logging.

APIM (API Management):

- Tools: Azure APIM, AWS API Gateway, Kong.
- Features:
- Rate limiting
- Caching
- Authentication
- Analytics

Rate Limiting:

- Controls request rate per user/IP.
- Example: 100 requests/minute.

Configuration Experience:

- Define policies in APIM.
- Use YAML/JSON for Kong or Spring Cloud Gateway.
- Example in Spring Cloud Gateway:



```
1 routes:
2   - id: user-service
3     uri: lb://USER-SERVICE
4     predicates:
5       - Path=/users/**
6     filters:
7       - name: RequestRateLimiter
8         args:
9           redis-rate-limiter.replenishRate: 10
10          redis-rate-limiter.burstCapacity: 20
11
```

Coding Question:

You have an array of numbers. You need to find the second largest element. Write the code. What will be the complexity?


```

5 public class SecondLargestWithStream {
6     public static void main(String[] args) {
7         int[] arr = {10, 20, 4, 45, 99, 99};
8
9         Optional<Integer> secondLargest = Arrays.stream(arr)
10            .boxed()
11            .distinct()
12            .sorted(Comparator.reverseOrder())
13            .skip(1)
14            .findFirst();
15
16         secondLargest.ifPresentOrElse(
17             val -> System.out.println("Second Largest: " + val),
18             () -> System.out.println("No second largest element found.")
19         );
20     }
21 }
22

```

Print Occurance of char from String:

```

1 import java.util.function.Function;
2 import java.util.stream.Collectors;
3
4 public class CharacterFrequencyStream {
5     public static void main(String[] args) {
6         String input = "hello world";
7
8         input.chars()
9             .mapToObj(c -> (char) c)
10            .filter(c -> c != ' ') // ignore spaces
11            .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
12            .forEach((character, count) -> System.out.println(character + " : " +
13                count));
14    }
15 }

```