Experienced Java Developer real interview experience

1. **OOPs concept with short example related to current work.**

- **Abstraction** is the process of hiding the implementation details and showing only the essential features of an object. It allows focusing on what an object does rather than how it does it. Abstraction can be achieved through abstract classes and interfaces.

  Example: In Spring Boot development, abstraction can be seen in service interfaces. For example, consider a UserService interface defining methods like createUser(), getUserById(), and updateUser(). The interface hides the implementation details of these methods, allowing different service implementations (e.g., UserServiceImpl) to provide specific logic while adhering to the contract defined by the interface. Other parts of the application interact with the UserService interface without needing to know the specific implementation details, promoting loose coupling and flexibility.

- **Encapsulation** is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit, called a class. It hides the internal state of an object and only exposes necessary functionalities through methods.

  Example: In Spring Boot development, encapsulation can be seen in repository classes, which encapsulate database access logic. For example, a UserRepository class encapsulates methods for CRUD operations on user entities, abstracting away the underlying database implementation details.

- **Inheritance** is a mechanism where a new class (subclass/child class) is derived from an existing class (superclass/parent class), inheriting attributes and methods. It promotes code reuse and establishes a hierarchical relationship between classes.

  Example: In Spring Boot, inheritance is used in controller classes. A BaseController class may contain common functionalities such as exception handling or logging. Subclasses like UserController and ProductController inherit from BaseController to reuse these common functionalities while adding specific features.

- **Polymorphism** means the ability of objects to take on multiple forms. In OOP, it allows objects of different classes to be treated as objects of a common superclass. Polymorphism can be achieved through method overriding.

  Example: Polymorphism is present in Spring Boot through method overriding in service interfaces. For instance, a PaymentService interface can have multiple implementations (e.g., CreditCardPaymentService, PayPalPaymentService), each providing its specific logic for processing payments. Despite different implementations, the application can treat all payment services uniformly, promoting flexibility and maintainability.

2. **Difference between Interface and Abstract class?**

   Interfaces define contracts for classes to implement, promoting loose coupling and enabling polymorphism.

   Abstract classes provide a common base implementation for related classes, allowing code reuse through inheritance and promoting code organization.

3. **Difference between public, protected, private & default**?

   - public: Accessible everywhere.
   - protected: Accessible within the same package and by subclasses (regardless of the package).
   - private: Accessible only within the same class.
   - Default (package-private): Accessible only within the same package.

4. **Basic difference between dependency injection and inversion of control.**

   Dependency Injection (DI):

   - DI is a design pattern used to implement IoC by injecting dependencies into a class rather than allowing the class to create them itself.
   - It helps in achieving loose coupling between classes by decoupling the creation and use of objects.
   - DI frameworks provide mechanisms for automatically injecting dependencies into classes, typically through constructor injection, setter injection, or method injection.

   Inversion of Control (IoC):

   - IoC is a broader design principle where the control of the flow of a program is inverted or handed over to a framework or container.
   - It delegates the responsibility of managing the lifecycle and dependencies of objects to an external container or framework.
   - IoC containers manage the instantiation, configuration, and assembly of objects, thereby promoting decoupling and modularity in the application.

   In summary, DI is a technique used to achieve IoC by injecting dependencies into classes, while IoC is a design principle that dictates the flow of control in a program, shifting the responsibility of managing dependencies and object lifecycles to an external container or framework.

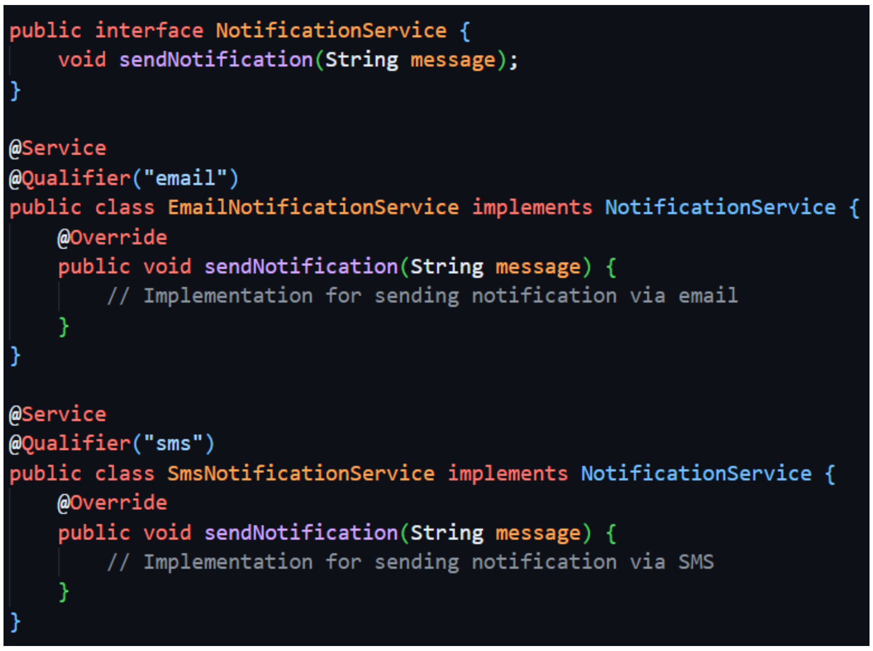
   In Spring MVC, beans are typically configured using XML or Java-based configuration files. However, with Spring Boot, the need for explicit bean configuration is reduced. Spring Boot uses auto-configuration and component scanning to automatically configure beans based on dependencies and annotations. This eliminates the need for manual bean initialization, making development more efficient.

5. **Can we create multiple service implementation class of service interface? If yes then how we will differentiate that in controller which service implementation we want to use.**

   Yes, you can create multiple implementations of a service interface in Spring. To differentiate between these implementations in your controller, you can use the @Qualifier annotation along with constructor injection.

   Here's how you can do it:

   a. Define multiple implementations of the service interface:

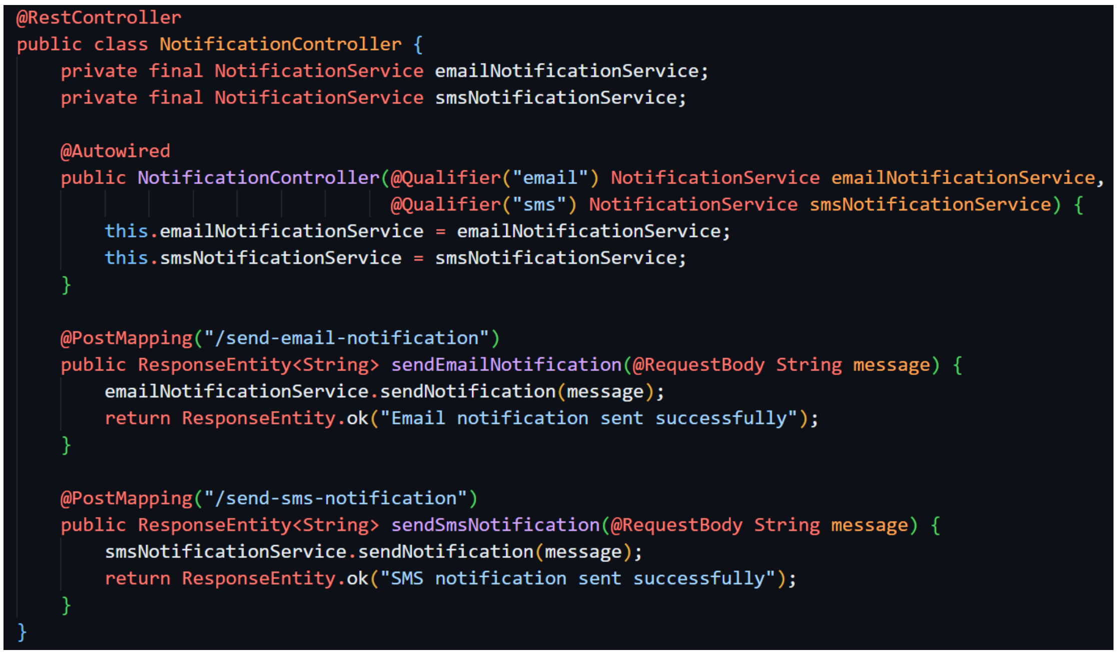


```java
public interface NotificationService {
    void sendNotification(String message);
}

@Service
@Qualifier("email")
public class EmailNotificationService implements NotificationService {
    @Override
    public void sendNotification(String message) {
        // Implementation for sending notification via email
    }
}

@Service
@Qualifier("sms")
public class SmsNotificationService implements NotificationService {
    @Override
    public void sendNotification(String message) {
        // Implementation for sending notification via SMS
    }
}
```

   b. Inject the desired implementation into your controller using constructor injection along with **@Qualifier** annotation:



```java
@RestController
public class NotificationController {
    private final NotificationService emailNotificationService;
    private final NotificationService smsNotificationService;

    @Autowired
    public NotificationController(@Qualifier("email") NotificationService emailNotificationService,
                                  @Qualifier("sms") NotificationService smsNotificationService) {
        this.emailNotificationService = emailNotificationService;
        this.smsNotificationService = smsNotificationService;
    }

    @PostMapping("/send-email-notification")
    public ResponseEntity<String> sendEmailNotification(@RequestBody String message) {
        emailNotificationService.sendNotification(message);
        return ResponseEntity.ok("Email notification sent successfully");
    }

    @PostMapping("/send-sms-notification")
    public ResponseEntity<String> sendSmsNotification(@RequestBody String message) {
        smsNotificationService.sendNotification(message);
        return ResponseEntity.ok("SMS notification sent successfully");
    }
}
```

6. **How to handle different API call scenarios like successful, user not authorized and any failure like DB failure etc?**

   To handle the scenarios of successful API calls, unauthorized access, and failures like database errors, you can implement appropriate error handling mechanisms in your application. Here's how you can handle each scenario:

   Successful API Call:

   - In the case of a successful API call, your application should process the response data and perform any required actions.
   - Typically, you would return the successful response data to the client along with an appropriate HTTP status code (e.g., 200 OK).

   Unauthorized Access:

   - When a user is not authorized to access a resource or perform an action, your application should return an error response indicating the unauthorized access.
   - You can use HTTP status code 401 (Unauthorized) to indicate that authentication is required and has failed, or 403 (Forbidden) to indicate that the server understood the request but refuses to authorize it.

   Failure (e.g., Database Error):

   - In case of failures such as database errors or other unexpected exceptions, your application should handle the error gracefully and provide a meaningful error response to the client.
   - You can return an appropriate HTTP status code (e.g., 500 Internal Server Error) along with an error message describing the nature of the failure.

Another approach to handle API responses in a more structured and standardized manner is by using exception handling and global error handling mechanisms provided by Spring Boot.

Exception Handling:

- Define custom exception classes to represent different types of errors in your application, such as UnauthorizedAccessException or DatabaseException.
- Throw these custom exceptions from your service layer when specific error conditions occur.

Global Error Handling:

- Use Spring's **@ControllerAdvice** annotation to define a global exception handler class.
- Implement methods in this class to handle specific types of exceptions and return appropriate error responses.

7. Basic difference between String, StringBuffer & StringBuilder and when to use what?

The basic difference between **String**, **StringBuffer**, and **StringBuilder** lies in their mutability, synchronization, and performance characteristics:

I. **String**:
- Immutable: Once created, a string object cannot be modified.
- Thread-safe: Strings are inherently thread-safe because they cannot be changed.
- Use cases: Use **String** when you need an immutable sequence of characters, such as storing constants, representing fixed values, or passing data between methods without risk of modification.

II. **StringBuffer**:
- Mutable: StringBuffer objects can be modified after creation.
- Thread-safe: StringBuffer is synchronized, making it safe for use in multi-threaded environments.
- Slightly slower performance compared to StringBuilder.
- Use cases: StringBuffer is suitable for scenarios where thread safety is required, such as when performing string manipulation in a multi-threaded environment.

III. **StringBuilder**:
- Mutable: StringBuilder objects can be modified after creation.
- Not thread-safe: StringBuilder is not synchronized, so it's not suitable for use in multi-threaded environments without external synchronization.
- Better performance compared to StringBuffer due to lack of synchronization.
- Use cases: StringBuilder is ideal for single-threaded scenarios where high-performance string manipulation is required. It's commonly used in situations where thread safety is not a concern.

When to use what:

- Use **String** when you need an immutable sequence of characters, and you don't require modifications.
- Use **StringBuffer** when you need a mutable sequence of characters and thread safety is a concern (e.g., in multi-threaded applications).
- Use **StringBuilder** when you need a mutable sequence of characters in a single-threaded environment, and performance is critical.

In summary, choose **String** for immutability, **StringBuffer** for thread safety, and **StringBuilder** for performance in single-threaded scenarios. Selecting the appropriate class ensures efficient and safe string manipulation based on the requirements of your application.

8. How Garbage Collection works?

Garbage collection is an automatic memory management feature in Java that automatically reclaims memory occupied by objects that are no longer in use, allowing it to be reused by the program. Here's how it works:

I. **Identification**: The garbage collector identifies objects in memory that are no longer reachable or accessible by the program. An object is considered unreachable if there are no references to it from any active part of the program.
II. **Marking**: The garbage collector traverses the object graph starting from a set of known root objects (such as local variables, static variables, and active threads) and marks all reachable objects as live.
III. **Sweeping**: After marking all live objects, the garbage collector sweeps through the heap and deallocates memory for objects that are not marked as live. This memory is then made available for future allocations.

Java's garbage collector runs automatically in the background, periodically checking for objects that are no longer needed and reclaiming their memory. The exact algorithm and timing of garbage collection can vary depending on the JVM implementation and configuration settings.
Garbage collection helps developers focus on writing code without worrying too much about memory management and resource cleanup. However, it's essential to understand its behaviour and implications to design efficient and scalable Java applications.

In Java, you can request the garbage collector to run manually, but it's generally not recommended to do so under normal circumstances.
The Java Virtual Machine (JVM) manages the garbage collection process automatically, and it's usually efficient at reclaiming memory when needed. However, there might be some scenarios where you want to suggest the JVM to perform garbage collection, such as before taking memory-intensive operations or when you suspect a memory leak.
To manually suggest garbage collection, you can call the **System.gc()** method. However, it's important to note that calling **System.gc()** is just a suggestion to the JVM, and there is no guarantee that garbage collection will occur immediately. The JVM may choose to ignore the request or postpone garbage collection based on its internal algorithms and heuristics.

Again, it's crucial to understand that manually invoking garbage collection should be done sparingly and only in specific cases where you have a valid reason to do so, such as performance optimization or debugging memory-related issues. In most cases, trusting the JVM's automatic garbage collection mechanism is the preferred approach.

9. What are the different generation during Garbage collection how the cleaning is done?

Garbage collection in Java typically occurs in multiple generations, primarily to optimize performance and minimize pause times. The main generations involved in garbage collection are:

I. **Young Generation**: Newly created objects are allocated in the young generation. Garbage collection in this generation is typically fast and aims to reclaim short-lived objects. It usually involves the following steps:
   - **Minor GC (Minor Garbage Collection)**: This is triggered when the young generation becomes full. During a minor GC, live objects are moved to the old generation, and unreachable objects are reclaimed. The surviving objects are promoted to the old generation.

II. **Old Generation (Tenured Generation)**: Objects that survive multiple minor GC cycles are promoted to the old generation. Garbage collection in this generation is less frequent but involves larger pauses. It usually includes:
   - **Major GC (Full Garbage Collection)**: Also known as a full GC, this is triggered when the old generation becomes full. During a major GC, the entire heap, including both the young and old generations, is collected. This process can result in longer pauses compared to minor GC.

III. **Permanent Generation (Java 8 and earlier)**: This generation is used to store metadata related to classes, methods, and other internal JVM structures. In Java 8 and earlier versions, it's collected as part of the full GC.
Starting from Java 9, the Permanent Generation was replaced by the **Metaspace**, which is an area of native memory outside the Java heap. Metaspace stores metadata related to classes and is managed dynamically by the JVM.

The cleaning process during garbage collection involves identifying live objects (objects still in use) and reclaiming memory occupied by unreachable objects (objects no longer referenced). The JVM uses various garbage collection algorithms and strategies to optimize memory usage, minimize pause times, and ensure efficient application performance.
Overall, garbage collection is a complex process managed by the JVM, and the specific algorithms and behaviours may vary depending on the JVM implementation, configuration settings, and garbage collection strategy (e.g., serial, parallel, G1, ZGC, Shenandoah).

10. When and when not to use Microservices?

Microservices architecture has become increasingly popular for building scalable, resilient, and maintainable software systems. However, it's important to understand when it's appropriate to use microservices and when it might not be the best choice. Here are some factors to consider:

When to Use Microservices:

a. **Complexity**: Use microservices when you have a large, complex application that can be divided into multiple smaller, loosely coupled services. Microservices can help manage the complexity of such applications by breaking them down into smaller, more manageable parts.
b. **Scalability**: Microservices are well-suited for applications that need to scale horizontally. Each microservice can be independently scaled based on its specific resource requirements, allowing for better resource utilization and scalability.
c. **Technology Diversity**: Microservices allow you to use different technologies, frameworks, and programming languages for different parts of your application. This flexibility can be beneficial when certain technologies are better suited for specific tasks or when you want to leverage the strengths of different technologies.
d. **Team Autonomy**: Microservices enable teams to work independently on different services, allowing for faster development cycles and greater agility. Each team can choose its own technology stack, development process, and deployment strategy, leading to increased autonomy and innovation.
e. **Fault Isolation**: Microservices promote fault isolation, meaning that failures in one service do not necessarily affect the entire application. This can improve overall system resilience and reliability.

When Not to Use Microservices:

a. **Simple Applications**: For small, simple applications with limited functionality, the overhead of managing multiple microservices may outweigh the benefits. In such cases, a monolithic architecture might be more suitable and easier to manage.
b. **Resource Constraints**: Microservices introduce additional complexity and overhead in terms of deployment, monitoring, and management. If you have limited resources or expertise to manage a distributed system, it may be better to start with a monolithic architecture and refactor into microservices as needed.
c. **Performance Overhead**: Communication between microservices typically involves network calls, which can introduce latency and performance overhead compared to in-process communication in a monolithic application. If performance is a critical concern, carefully consider the impact of microservices on latency and throughput.
d. **Data Sharing and Transactions**: Microservices work best when each service has its own independent data store and can operate autonomously. If your application requires complex data sharing or transactions across multiple services, managing data consistency and ensuring transactional integrity can become challenging in a microservices architecture.
e. **Organizational Readiness**: Adopting microservices requires changes not only in technology but also in organizational structure, processes, and culture. If your organization is not ready to embrace DevOps practices, continuous delivery, and cross-functional teams, the transition to microservices may be difficult.

In summary, while microservices offer many benefits in terms of scalability, agility, and fault tolerance, they are not a one-size-fits-all solution. It's essential to carefully evaluate your application requirements, team capabilities, and organizational readiness before deciding whether to adopt a microservices architecture.