1. **Is Java Platform Independent? If so, how?**

Yes, Java is a platform-independent language. The compilation process generates bytecode (.class files), which can run on any system with a Java Virtual Machine (JVM) installed. The JVM interprets the bytecode, making Java code portable across different platforms.

1. JVM (Java Virtual Machine):

* The JVM is responsible for interpreting and executing Java bytecode.
* It provides a runtime environment in which Java programs run.
* Different JVM implementations exist for various hardware and software platforms (e.g., Windows, Linux).
* Therefore, the JVM is platform-dependent.

2. JRE (Java Runtime Environment):

* The JRE includes the JVM along with libraries and tools required for executing Java applications.
* It physically exists and provides the minimum requirements for running Java code.
* Like the JVM, the JRE is platform-dependent due to differences in OS configurations.

3. JDK (Java Development Kit):

* The JDK includes the JRE, a compiler (written in Java), and additional development tools.
* Development tools include javac, jar, Javadoc for documentation generator and Jvisualvm which provides visual profiler and monitoring interface
* It is used for developing and compiling Java applications.
* Since the JDK includes the JRE, which in turn includes the JVM, it is also platform-dependent

In summary, while the JVM, JRE, and JDK are platform-dependent, Java itself remains platform-independent because its bytecode can run on different OSs with the appropriate JVMIt is used

1. Program.java -------(javac)-------🡪 Program.class (bytecode is platform-independent)

2. The compiled bytecode is not directly executable by OS. It needs to be interpreted or compiled into machine code.

3. **The JVM** loads the bytecode and translates it into machine code that the OS can execute.

This process involves several steps:

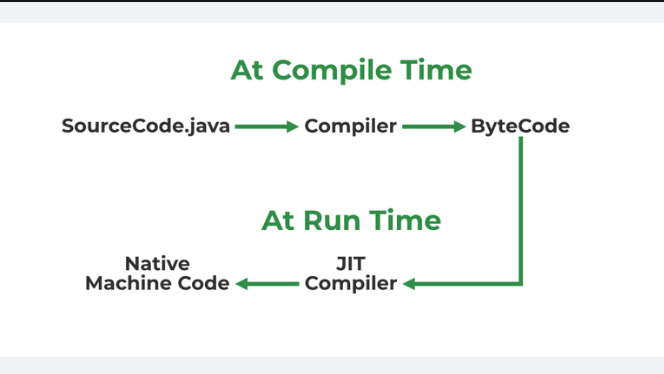
(i) **Class Loader**: Loads the .class files into memory.

(ii) **Bytecode Verifier**: Ensures the bytecode is valid and does not violate Java’s security constraints.

(iii) **Just-In-Time (JIT) Compiler**: Converts bytecode into native machine code for execution.

*i.e:- Java Program*

[**https://www.geeksforgeeks.org/java-interview-questions/**](https://www.geeksforgeeks.org/java-interview-questions/)

****

JIT stands for (Just-in-Time) compiler is a part of JRE(Java Runtime Environment), it is used for better performance of the Java applications during run-time. The use of JIT is mentioned in step-by-step process mentioned below:

* + Source code is compiled with javac(.java) compiler to form bytecode(.class)
  + Bytecode is further passed on to JVM
  + JIT is a part of JVM, JIT is responsible for compiling bytecode into native machine code at run time.
  + The JIT compiler is enabled throughout, while it gets activated when a method is invoked. For a compiled method, the JVM directly calls the compiled code, instead of interpreting it.
  + As JVM calls the compiled code that increases the performance and speed of the execution.

 JIT compilers allow the JVM to translate Java code to machine code as and when needed by the JDK. AOT compilers compile the Java code within a JAR file into native shared libraries before the execution time.

**Primitive datatypes in java**

Boolean,byte,char,short,int,long,float,double

**Non-primitive**

Strings, array, class, object, interface

**Wrapper Class**

Wrapper is referred to a larger entity that encapsulates a smaller entity. Here in java

Wrapper class is an object class that encapsulates the primitive data types.

Java contains 8 wrapper classes Boolean, Byte, Short, Integer, Character, Long, Float and Double.

**Why do we need wrapper classes**

Wrapper classes enclose around primitive data types, giving them an object appearance

1. wrapper classes are final and immutable

2. helps in working with collections and data structures

3. provides methods like valueOf(), parseInt(), etc.

This function converts primitive value into an object of corresponding wrapper class

Integer.valueOf(“100”);

* For example, an ArrayList<Integer> can hold a list of integers.

Without wrapper classes we cannot store primitive types directly in collections and also

Primitive types cannot be used directly for synchronization

In summary, wrapper classes bridge the gap between primitive types and objects

**Java Spring Boot Project Development**

**1. Why is Lombok :** popular java library used to reduce boiler plate code.

Lombok eliminates the need to manually write repetitive code such as getters, setters, all args and no args constructors, equals(), hashCode(), and toString() methods

i) @Data Annotation combines -> @Getter, @Setter, @ToString, @EqualsAndHashCode, and @RequiredArgsConstructor (A constructor that takes each parameter if they are final or marked with @NonNull)

ii) @AllArgsConstructor

iii) @NoArgsConstructor

**2. Object Mapper (**needs Jackson dependency**):** ObjectMapper is widely used for converting Java objects to JSON and vice versa. It provides a high-level API for reading and writing JSON, making it a powerful tool for handling JSON data in Java applications.

1. ObjectMapper objectMapper = new ObjectMapper();

String jsonString = "{\"name\":\"John\",\"age\":30}";

Person person = objectMapper.readValue(jsonString, Person.class);

System.out.println(person.getName()); // John

2. ObjectMapper objectMapper = new ObjectMapper();

Person person = new Person("John", 30);

String jsonString = objectMapper.writeValueAsString(person);

System.out.println(jsonString); // {"name":"John","age":30}

Using JSONObject

String jsonString = "{\"name\":\"John\",\"age\":30,\"city\":{\"name\":\"New YORK\",\"pincode\":507001}";

// Convert string to JSONObject

JSONObject jsonObject = new JSONObject(jsonString);

// Access values from the JSONObject

String name = jsonObject.getJSONObject("city");

String name = jsonObject.getString("name");

How to Optimize Spring boot App performance

1. Use caching to reduce load on database and to store frequently accessed data in memory

2. Lazy initialization can help reduce memory usage by loading beans only when they are needed. [This can be done using the @Lazy annotation](https://dev.to/jackynote/efficiently-optimizing-spring-boot-applications-faster-startup-and-lower-memory-usage-hjo)

@Configuration

public class MyConfig {

@Bean

@Lazy

public MyBean myBean() {

return new MyBean();

}

}

3. minimize auto-configuration

Disable unnecessary auto-configuration to reduce load time

4. profile-specific configuration

Use profiles to load configurations selectively based on the environment

5. Using spring boot actuator provides various endpoints for monitoring and managing your application. It helps to diagnose performance bottlenecks

**Spring IOC**

Spring IoC (Inversion of Control) is a core concept in the Spring Framework that manages creation of objects, configures and assembles their dependencies, manage their entire life cycle. The container uses DI to manage the components that make up the application.

**Dependency Injection (DI):**

DI is a pattern used to implement IoC, where the framework injects dependencies into objects rather than the objects creating their own dependencies.

Types of DI:

Constructor Injection: Dependencies are provided through a class constructor.

Setter Injection: Dependencies are provided through setter methods

Field Injection (Not recommended due issues with immutability and testability)

**Spring BEAN**

Spring Beans are objects that are managed by the Spring IoC (Inversion of Control) container. They are defined in the Spring configuration file or annotated with specific Spring annotations.

Example:

**Java**

import org.springframework.stereotype.Component;

@Component

public class MyBean {

public void doSomething() {

System.out.println("Doing something...");

}

}

In this example, MyBean is a Spring Bean managed by the Spring container. The @Component annotation indicates that this class is a Spring-managed bean.

In Spring, @Component and @Bean are both used to define beans, but they serve different purposes and are used in different contexts.

**@Component** is a class-level annotation used to indicate that a class is a Spring-managed component. It allows Spring to automatically detect and register the class as a bean during classpath scanning.

Use @Component when you want Spring to automatically detect and manage your beans. It’s particularly useful for your own classes where you have control over the source code.

Specializations: There are specialized annotations derived from @Component for specific use cases:

@Service: Indicates a service layer component.

@Repository: Indicates a data access layer component.

@Controller: Indicates a web controller.

This annotation is used to mark a class as a Spring MVC controller. This means that the class will handle HTTP requests and return a view to be rendered as a response. Spring automatically detects and registers this as a bean while component scanning

**@Controller**:

* **Purpose**: Used to define a controller in a Spring MVC application.
* **View Resolution**: Typically returns a view name, which is resolved to an HTML page or another view technology by a view resolver.
* **Usage**: Suitable for web applications that render views (like JSP, Thymeleaf).

**@RestController**:

* **Purpose**: A specialized version of @Controller used to create RESTful web services.
* **Combines Annotations**: Combines @Controller and @ResponseBody, meaning that the return value of the methods is directly written to the HTTP response body.
* **Usage**: Suitable for REST APIs that return data (usually JSON or XML) directly.

@ResponseBody on a method, Spring converts the return value and writes it to the HTTP response automatically without resolving it to a view unlike in case of @Controller

**@Bean**

@Bean is a method-level annotation used in a configuration class to define a bean. The method annotated with @Bean will return an object that Spring should register as a bean in the application context.

**Use Case:** Use @Bean when you need to create and configure beans that are not under your control, such as third-party library classes, or when you need more control over the bean creation process.

Configuration Class: Typically used within a class annotated with @Configuration.

**Java**

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

@Bean

public MyBean myBean() {

return new MyBean();

}

}

public class MyBean {

public void doSomething() {

System.out.println("Doing something...");

}

}

Key Differences:

**Level**: @Component: Class-level annotation.

@Bean: Method-level annotation.

@Component: Automatically detected and registered by Spring during class path scanning.

@Bean: Explicitly defined in a configuration class.

**Use Case:**

@Component: For your own classes where you have control over the source code.

[@Bean: For third-party classes or when you need more control over bean creation](https://stackoverflow.com/questions/10604298/spring-component-versus-bean)

When Spring starts, it will:

1. **Scan** for MyComponent because it is annotated with @Component.
2. **Create a bean definition** for MyComponent.
3. **Register** MyComponent as a bean in the application context.
4. **Inject dependencies** if MyComponent has any.
5. **Initialize** MyComponent.
6. **Make**MyComponent**available** for injection and use in other parts of the application.

The @ComponentScan annotation is used to specify the packages to scan for annotated components.

By default, Spring Boot scans the package of the main application class and its sub-packages.

Spring scans for classes annotated with @Component, @Service, @Repository, @Controller, and other stereotype annotations.

These annotations indicate that the class should be managed by the Spring IoC container.

**Types of Annotations**

1. Core Annotations

* **@SpringBootApplication**: Combines @Configuration, @EnableAutoConfiguration, and @ComponentScan to bootstrap a Spring Boot application.
* **@Component**: Indicates that a class is a Spring-managed component.
* **@Configuration**: Indicates that a class declares one or more @Bean methods.
* **@Bean**: Indicates that a method produces a bean to be managed by Spring.
* **@Autowired**: Used for automatic dependency injection.

2. Web Annotations

* **@Controller**: Marks a class as a Spring MVC controller.
* **@RestController**: Combines @Controller and @ResponseBody to create RESTful web services.
* **@RequestMapping**: Maps HTTP requests to handler methods of MVC and REST controllers.
* **@GetMapping, @PostMapping, @PutMapping, @DeleteMapping**: Specialized versions of @RequestMapping for specific HTTP methods.

3. Data Annotations

* **@Entity**: Specifies that the class is an entity and is mapped to a database table.
* **@Table**: Specifies the table in the database with which the entity is mapped.
* **@Id**: Specifies the primary key of an entity.
* **@GeneratedValue**: Specifies the generation strategy for the primary key.
* **@Repository**: Indicates that the class is a repository, which is an abstraction of data access and storage.

4. Validation Annotations

* **@Valid**: Marks a method parameter or field for validation.
* **@NotNull, @Size, @Min, @Max, @Pattern**: Common validation annotations for bean validation.

5. Security Annotations

* **@Secured**: Specifies a list of roles that are allowed to access a method.
* **@PreAuthorize, @PostAuthorize**: Used for method-level security based on expressions.

6. Testing Annotations

* **@SpringBootTest**: Used to create an application context for integration tests.
* **@MockBean**: Used to add mock objects to the Spring application context.
* **@WebMvcTest**: Used to test Spring MVC controllers.

7. Caching Annotations

* **@EnableCaching**: Enables Spring’s annotation-driven cache management capability.
* **@Cacheable**: Indicates that the result of a method can be cached.
* **@CacheEvict**: Indicates that one or more caches should be evicted.

8. Scheduling Annotations

* **@EnableScheduling**: Enables Spring’s scheduled task execution capability.
* **@Scheduled**: Used to schedule tasks to be executed at fixed intervals.

9. Aspect-Oriented Programming (AOP) Annotations

* **@Aspect**: Indicates that a class is an aspect.
* **@Before, @After, @Around**: Used to define advice that runs before, after, or around method executions.

**Java 8 Features**

1. Lambda Expressions

Lambda expressions provide a clear and concise way to represent one method interface using an expression. They enable functional programming in Java and are particularly useful for iterating, filtering, and extracting data from collections.

**Example**:

**Java**

List<String> names = Arrays.asList("John", "Jane", "Jack");

names.forEach(name -> System.out.println(name));

2. Functional Interfaces

A functional interface is an interface that contains only one abstract method. They can have multiple default or static methods. Common examples include Runnable, Callable, and custom interfaces annotated with @FunctionalInterface.

**Example**:

**Java**

@FunctionalInterface

public interface MyFunctionalInterface {

void execute();

}

3. Stream API

The Stream API allows for functional-style operations on streams of elements, such as map-reduce transformations on collections. It provides a powerful way to process sequences of elements.

**Example**:

**Java**

List<String> names = Arrays.asList("John", "Jane", "Jack");

names.stream()

.filter(name -> name.startsWith("J"))

.forEach(System.out::println);

4. Default Methods

Interfaces can now have default methods, which are methods with a default implementation. This allows the addition of new methods to interfaces without breaking existing implementations.

**Example**:

**Java**

public interface MyInterface {

default void defaultMethod() {

System.out.println("Default method");

}

}

5. Optional Class

The Optional class is a container object which may or may not contain a non-null value. It helps to avoid NullPointerException and provides methods to deal with the presence or absence of a value.

**Example**:

**Java**

Optional<String> optional = Optional.ofNullable("Hello");

optional.ifPresent(System.out::println);

6. Date and Time API

Java 8 introduced a new date and time API in the java.time package, which is more comprehensive and user-friendly compared to the old java.util.Date and java.util.Calendar classes.

**Example**:

**Java**

LocalDate today = LocalDate.now();

System.out.println(today);

7. Method References

Method references provide a way to refer to methods without invoking them. They are a shorthand notation of a lambda expression to call a method.

**Example**:

**Java**

List<String> names = Arrays.asList("John", "Jane", "Jack");

names.forEach(System.out::println);

8. Nashorn JavaScript Engine

Nashorn is a JavaScript engine that allows you to run JavaScript code on the Java Virtual Machine (JVM). It provides better performance and compliance with the ECMAScript specification.

9. Base64 Encoding and Decoding

Java 8 includes a utility class for Base64 encoding and decoding.

**Example**:

**Java**

String encoded = Base64.getEncoder().encodeToString("Hello".getBytes());

System.out.println(encoded);

10. Parallel Array Sorting

Java 8 introduced the Arrays.parallelSort method, which uses the Fork/Join framework to sort large arrays more efficiently.

**Example**:

**Java**

int[] array = {5, 3, 1, 4, 2};

Arrays.parallelSort(array);

System.out.println(Arrays.toString(array));

**Collections.unmodifiableList()** returns a list which **can’t be altered** i.e. it can neither add or delete an element. Any attempt to modify the list will result in an UnsupportedOperationExample.

Syntax:

List<Integer> list = Collections

.unmodifiableList(Arrays.asList(1, 2, 3));

The Class which is used to bootstrap the Spring boot application is annotated with @SpringBootApplication

**INTERVIEW QUESTIONS**

Below are important Spring Boot dependencies that need to be added to a Gradle-based or Maven-based application, to ensure application compatibility with Spring Boot features.

* spring-boot-starter-parent
* spring-boot-maven-plugin
* spring-boot-starter-test
* spring-boot-starter-security
* spring-boot-starter-actuator
* Spring-boot-starter-web

**What are microservices?**

Microservices is an architectural style that structures an application as a collection of small, loosely coupled, and independently deployable services. [Each service represents a specific business capability](https://www.javaguides.net/2023/07/spring-boot-microservices-interview.html).

**What is Spring Boot?**

Spring Boot is a framework built on top of the Spring framework that simplifies the development of Java applications, including microservices. [It provides a convention-over-configuration approach, auto-configuration, and embedded servers](https://www.javaguides.net/2023/07/spring-boot-microservices-interview.html)

**What are the advantages of using Spring Boot for microservices?**

[Simplified development, auto-configuration, embedded servers, and cloud-native support](https://www.javaguides.net/2023/07/spring-boot-microservices-interview.html).

**How does Spring Boot simplify dependency management?**

[By using starter dependencies, which are pre-configured sets of dependencies for common use cases](https://www.geeksforgeeks.org/spring-boot-interview-questions-and-answers/).

**What is the role of @SpringBootApplication?**

[It is a convenience annotation that combines @Configuration, @EnableAutoConfiguration, and @ComponentScan](https://www.geeksforgeeks.org/spring-boot-interview-questions-and-answers/).

**How do you communicate between microservices in Spring Boot?**

[Using synchronous HTTP/REST, messaging (e.g., RabbitMQ, Kafka), and service discovery mechanisms like Netflix Eureka or Spring Cloud Consul](https://www.javaguides.net/2023/07/spring-boot-microservices-interview.html).

**What are service registration and discovery?**

[A mechanism that allows microservices to register themselves with a service registry and discover other services dynamically](https://www.javaguides.net/2023/07/spring-boot-microservices-interview.html).

**How does Spring Cloud help with building microservices?**

[Provides tools for service discovery, client-side load balancing, distributed tracing, circuit breakers, and more](https://www.javaguides.net/2023/07/spring-boot-microservices-interview.html).

**What is a circuit breaker pattern?**

[A design pattern used to detect failures and encapsulate the logic of preventing a failure from constantly recurring during maintenance, temporary external system failure, or unexpected system difficulties](https://www.javaguides.net/2023/07/spring-boot-microservices-interview.html).

**How do you handle security in microservices?**

[Using Spring Security for authentication and authorization, OAuth2 for token-based security, and API gateways for centralized security management](https://www.javaguides.net/2023/07/spring-boot-microservices-interview.html).

**What is the role of an API Gateway in microservices architecture?**

[An API Gateway acts as a single-entry point for all client requests, handling routing, composition, and protocol translation](https://www.javaguides.net/2023/07/spring-boot-microservices-interview.html).

**How do you implement inter-service communication in Spring Boot?**

[Using RestTemplate or WebClient for synchronous calls and message brokers like RabbitMQ or Kafka for asynchronous communication](https://www.javaguides.net/2023/07/spring-boot-microservices-interview.html).

**How do you monitor microservices?**

[Using Spring Boot Actuator for health checks and metrics, and integrating with monitoring tools like Prometheus, Grafana, or ELK stack](https://www.javaguides.net/2023/07/spring-boot-microservices-interview.html).

**What is the use of @EnableEurekaClient?**

[It enables a Spring Boot application to register with a Eureka server for service discovery](https://www.javaguides.net/2023/07/spring-boot-microservices-interview.html).

**Key Reasons for Using @Configuration**

**Bean Definitions:**

The @Configuration annotation marks a class as a source of bean definitions. [This means that the Spring container will process the class to generate Spring Beans that can be used throughout the application](https://docs.spring.io/spring-framework/reference/core/beans/java/configuration-annotation.html).

**Java-Based Configuration:**

It allows you to configure your Spring application using Java code instead of XML. [This can make your configuration more type-safe and easier to refactor](https://www.geeksforgeeks.org/spring-configuration-annotation-with-example/).

**Inter-Bean Dependencies:**

Methods annotated with @Bean within a @Configuration class can call each other to express inter-bean dependencies. [This is not possible with plain @Component classes](https://docs.spring.io/spring-framework/reference/core/beans/java/configuration-annotation.html).

**Singleton Scope:**

Beans defined in a @Configuration class are singleton by default. [This means that the same instance of the bean will be used throughout the application context](https://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html)

**HOW TO ACCESS APPLICATION PROPERTIES IN JAVA CLASS**

You can use the @Value annotation to inject property values directly into your fields.

Example for application.yml:

@Value("${my.property}") //@Value also used to inject value to variables

private String myProperty;

### @ConfigurationProperties

This approach is useful for binding hierarchical properties to a POJO.

Example for application.yml:

my:

config:

name: "John"

age: 30

@Component

@ConfigurationProperties(prefix = "my.config")

public class MyConfig {

private String name;

private int age;

// getters and setters

}

### Environment

@Autowired

private Environment env;

public void printProperty() {

String myProperty = env.getProperty("my.property");

**DIFFERENCE BETWEEN .yml AND .properties FILE**

* .properties**File:**
  + All values are treated as strings.
  + Requires manual conversion for other data types.
  + Allows the use of @PropertySource annotation to load properties files with a different name.
  + @PropertySource(“classpath:my-application.properties”)
* .yml**File:**
  + Supports native data types (e.g., integers, booleans).
  + Provides better type safety.
  + Does not support @PropertySource directly, but can be used with @ConfigurationProperties

**AUTOWIRE**

adding dependencies of one bean class into dependent bean class without configuring it in our configuration class is called autowiring

1.constructor 2. Setter method 3. Property

**Scope of BEAN**

i) Singleton - default - @Scope(“singleton”)

ii)Prototype - @Scope(“prototype”)

**LOGGING**

Process opf writing log messages during execution of the program.

**Types** - JDK Logging API, Apache Log4j and Commons Logging API

**@Qualifier**

When you have multiple beans of the same type, Spring cannot decide which one to inject. This situation can lead to a **NoUniqueBeanDefinitionException**. The @Qualifier annotation helps to specify exactly which bean should be injected.

@Component("fooFormatter")

public class FooFormatter implements Formatter {

public String format() {

return "foo";

}

}

@Component("barFormatter")

public class BarFormatter implements Formatter {

public String format() {

return "bar";

}

}

**Solution:**

@Component

public class MyService {

@Autowired

@Qualifier("fooFormatter")

private Formatter formatter; // No ambiguity now

}

**Alternatives:**

1. using @primary

2. Autowiring by name - @private Formatter fooFormatter;

### ****Arrays****

1. **Find the minimum and maximum element in an array**.
2. **Reverse an array**.
3. **Find the Kth largest and Kth smallest number in an array**.
4. **Sort an array of 0s, 1s, and 2s**.
5. **Subarray with a given sum**.
6. **Move all negative elements to one side of the array**.
7. **Find the union and intersection of two sorted arrays**.
8. **Find the missing number in an array**.
9. **Find duplicates in an array**.
10. [**Find the first repeating element in an array**1](https://www.geeksforgeeks.org/top-50-array-coding-problems-for-interviews/).

### ****Strings****

1. **Reverse a string**.
2. **Check if a string is a palindrome**.
3. **Find the first non-repeating character in a string**.
4. **Check if two strings are anagrams**.
5. **Longest substring without repeating characters**.
6. **Count and say problem**.
7. **Longest common prefix**.
8. **String to integer (atoi)**.
9. **Implement strStr()**.
10. [**Group anagrams**1](https://www.geeksforgeeks.org/top-50-array-coding-problems-for-interviews/).

### ****Hashmaps****

1. **Two Sum**.
2. **Find the frequency of characters in a string**.
3. **Find the intersection of two arrays**.
4. **Find the first unique character in a string**.
5. **Group anagrams**.
6. **Find the longest substring without repeating characters**.
7. **Check if a string can be rearranged to form a palindrome**.
8. **Find the number of subarrays with a given XOR**.
9. **Find the longest consecutive sequence in an array**.
10. [**Check if two strings are isomorphic**](https://interviewprep.org/hashmap-interview-questions/)

**Array Functions**

### ****1.****Arrays.toString()

Converts the array to a string representation.

### ****2.****Arrays.sort()

Sorts the array in ascending order.

### ****3.****Arrays.binarySearch()

Searches for a specific element in a sorted array and returns its index.

### ****4.****Arrays.copyOf()

Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.

### ****5.****Arrays.equals()

Checks if two arrays are equal.

### ****6.****Arrays.fill()

Fills the specified array with the specified value.

### ****7.****Arrays.stream()

Creates a stream from the array.

### ****8.****Arrays.asList()

Converts an array to a list.

**String Functions**

### ****1.****length()

Returns the length of the string.

### ****2.****charAt(int index)

Returns the character at the specified index.

### ****3.****substring(int beginIndex, int endIndex)

Returns a new string that is a substring of the original string.

### ****4.****contains(CharSequence s)

Checks if the string contains the specified sequence of characters.

### ****5.****equals(Object Object)

Compares this string to the specified object.

### ****6.****equalsIgnoreCase(String anotherString)

Compares this string to another string, ignoring case considerations.

### ****7.****toUpperCase()****and****toLowerCase()

Converts all characters in the string to upper or lower case.

### ****8.****trim()

Removes leading and trailing whitespace.

### ****9.****replace(char oldChar, char newChar)

Replaces all occurrences of a specified character with a new character.

### ****10.****split(String regex)

Splits the string around matches of the given regular expression.

### ****11.****indexOf(int ch)

Returns the index within this string of the first occurrence of the specified character.

### ****12.****concat(String str)

Concatenates the specified string to the end of this string.

Class implements Comparable<Student> interface

* Here original class is modifies

@Override

public int compareTo(Student other) {

return Integer.compare(this.age, other.age);

}

List<Student> students = Arrays.asList(new Student(20), new Student(18), new Student(22));

Collections.sort(students);

Class implements Comparator<Student> interface

* Here original class is not modified

@Override

public int compare(Student s1, Student s2) {

return Integer.compare(s1.age, s2.age);

}

List<Student> students = Arrays.asList(new Student(20), new Student(18), new Student(22));

Collections.sort(students, new AgeComparator())

* **Comparable**: Use when you want a natural ordering for objects and the class itself should define this order.
* **Comparator**: Use when you need multiple ways to sort objects or when you cannot modify the class to implement Comparable

**Java Interview Questions**

1. **What are the main features of Java?**

i) it is object-oriented language, it organizes

ii) platform independent

iii) provides secure environment for developing applications

iv) supports dynamic loading of classes, functions and libraries at runtime

v) multi-threaded language. Concurrent execution of 2 or more threads

1. **Explain the concept of OOP (Object-Oriented Programming) in Java.**
2. **What is the difference between**==**and**equals()**in Java?**
3. **How does garbage collection work in Java?**

Garbage collection in java is an automatic process that manages memory by removing the objects from memory that are no longer in use by any thread

How It works:

When Java programs run, objects are created on the heap, a portion of memory dedicated to the program. [Over time, some objects become unreachable because they are no longer referenced by any part of the program](https://www.geeksforgeeks.org/garbage-collection-java/)

Garbage collector which is part of JVM that automatically identifies and deletes these unreachable objects to free up memory

**System.gc() Method**: This method suggests that the JVM perform garbage collection, but it is not guaranteed to do so.

1. **What are Java Streams and how do you use them?**
2. **Explain the difference between**ArrayList**and**LinkedList**.**
3. **What is the purpose of the**final**keyword in Java?**
4. **How do you handle exceptions in Java?**
5. **What is the difference between**HashMap**and**Hashtable**?**
6. **Explain the concept of multithreading in Java.**

**Spring Boot Interview Questions**

1. **What is Spring Boot and how does it differ from the Spring Framework?**
2. **What are the main features of Spring Boot?**
3. **Explain the concept of auto-configuration in Spring Boot.**
4. **How do you create a Spring Boot application?**
5. **What is Spring Boot Starter?**
6. **How do you configure a Spring Boot application?**
7. **What is Spring Boot Actuator and how is it used?**
8. **Explain the concept of dependency injection in Spring Boot.**
9. **How do you handle exceptions in Spring Boot?**
10. **What are Spring Boot profiles and how do you use them?**

**Advanced Spring Boot Questions**

1. **What is Spring Data JPA and how do you use it in Spring Boot?**
2. **Explain the concept of microservices and how Spring Boot supports it.**
3. **How do you secure a Spring Boot application?**
4. **What is Spring Cloud and how does it integrate with Spring Boot?**
5. **How do you implement logging in a Spring Boot application?**
6. **What are the different ways to run a Spring Boot application?**
7. **How do you test a Spring Boot application?**
8. **Explain the concept of caching in Spring Boot.**
9. **How do you configure a database connection in Spring Boot?**
10. **What is the role of**@SpringBootApplication**annotation?**

**TCS INTERVIEW EXPERIENCE**

1. Difference between Interface and Abstract class

* Use **interfaces** when you need to define a contract that can be implemented by any class, achieve multiple inheritance, or design APIs.
* Use **abstract classes** when you need to share code among closely related classes, encapsulate state, or provide a common base implementation.

Abstract classes can have fields and methods with any access modifier (public, protected, private). This allows for encapsulation and controlled access to the class members, which is not possible with interfaces (where all fields are implicitly public, static, and final).

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |

**OOP Concepts**

Object-Oriented Programming (OOP) in Java is a programming paradigm that uses objects and classes to structure software programs. Here are the four primary OOP concepts in Java:

**1. Encapsulation**

Encapsulation is the mechanism of wrapping the data (variables) and code (methods) together as a single unit. It restricts direct access to some of an object’s components, which can prevent the accidental modification of data. This is typically achieved using access modifiers like private, protected and public.

**2. Inheritance**

Inheritance is a mechanism where one class acquires the properties (fields) and behaviors (methods) of another class. It promotes code reusability and establishes a natural hierarchy between classes.

**3. Polymorphism**

Polymorphism allows methods to do different things based on the object it is acting upon, even though they share the same name. It can be achieved through method overloading (compile-time polymorphism) and method overriding (runtime polymorphism).

public abstract class Animal {

private String name;

public Animal(String name) {

this.name = name;

}

public String getName() {

return name;

}

public abstract void makeSound();

public void sleep() {

System.out.println(name + " is sleeping.");

}

}

}

**Explanation**

1. **Abstract Class**Animal: This class has a field name, a constructor to initialize it, an abstract method makeSound(), and a concrete method sleep().
2. **Concrete Subclasses**Dog**and**Cat: These classes extend Animal and provide implementations for the abstract method makeSound().
3. **Main Class**: This class creates instances of Dog and Cat, demonstrating polymorphism by calling methods on the Animal type.

**Benefits**

* **Code Reusability**: Common code (like the sleep() method) is placed in the abstract class, reducing duplication.
* **Polymorphism**: The makeSound() method can be called on an Animal reference, but the actual method that gets executed depends on the object’s runtime type.
* **Encapsulation**: The name field is encapsulated within the Animal class, with controlled access through the getName() method.

**Functional Interfaces**

1. These functional interfaces helps us in reducing the boilerplate code with lambda expressions and method references in concise view of a single abstract method. Integrates with standard java libraries especially Stream API

Below are inbuild functional interfaces

import java.util.function.(Function)\*;

1. Function<T,R> -------> accepts one argument and produces a result

Function<String, Integer> lengthFunction = String::length;

Integer length = lengthFunction.apply("Hello");

System.out.println("Length: " + length); // Outputs: Length: 5

2. Consumer<T> ------->  accepts a single input argument and returns no result

Consumer<String> printConsumer = System.out::println;

printConsumer.accept("Hello, World!"); // Outputs: Hello, World!

3. Supplier<T> --------->  It has a single abstract method get

Supplier<String> stringSupplier = () -> "Hello from Supplier";

System.out.println(stringSupplier.get()); // Outputs: Hello from Supplier

4. Predicate<T> ---------> (boolean-valued function) of one argument

Predicate<String> isEmpty = String::isEmpty;

boolean result = isEmpty.test("");

System.out.println("Is empty: " + result); // Outputs: Is empty: true

**HashMap and its internal implementation?**

**Can we insert duplicate values in hashmap, if yes how?**

HashMap<String, List<Integer>> map = new HashMap<>();

map.computeIfAbsent("Alice", k -> new ArrayList<>()).add(30);

System.out.println(map); // Outputs: {Bob=[25], Alice=[30, 40]}

**Why SpringBoot starters and explain?**

Spring boot uses starter dependencies to simplify dependency management and configuration.

Suppose if we have to implement a certain feature, spring starters group commonly used dependencies for that particular feature to be implemented. so that we need not manually configure each one

Example:- spring-boot-starter-web

This helps in building web application, including RESTful services, using spring MVC

Includes Jackson, embedded server, spring MVC provides core web framework for building web application

By using this starter dependencies, we can focus more on developing your application’s business logic rather than managing dependencies and configurations

* Starters ensure that you use compatible versions of libraries, reducing the risk of conflicts.
* Reduced configuration time

Other starts are

1. spring-boot-starter-parent

This is a special starter that provides default configuration for our application to quickly build our springboot project, It also provides default configurations for Maven plugins, such as  *maven-jar-plugin*, *maven-surefire-plugin*, and *maven-war-plugin*.

2. spring-boot-starter-data-jpa -----> simplifies integration with JPA

3. spring-boot-starter-test

It includes a variety of testing libraries and tools to help us to write and run tests effectively

4. spring-boot-starter-security

5. spring-boot-starter-actuator

Helps in identifying the operational information of our running springboot application

This exposes few endpoints like /health and /metrics and /info

**How to create Spring REST API**

1. create a new project using spring initializer (start.spring.io) and add

i) spring web

ii) spring data JPA

iii) mysql driver

2. create an entity that represents data model for app

3. create a repository interface extending JPA repository to handle CRUD operations

4. create a service to handle business logic -> autowire repository to perform DB operations

5. create a controller and annotate with @RestController & @RequestMapping

Autowire service class in controller and call the business logic implemented in service layer

i) **@RequestBody** -> deserializes JSON in request to java object

ii) **@PathVariable** -> used to extract values from the URI path “ /{id} “ -> @PathVariable Long id

iii) **@Requestparam** -> “/users” -> ( @RequestParam String name ) -> /users?name=John

iv) **@RequestHeader** annotation is used to extract values from HTTP headers

getUsers(@RequestHeader("Authorization") String token)

v) **@CookieValue** annotation is used to extract values from cookies.

getPreferences(@CookieValue("sessionId") String sessionId)

vi) **@RequestPart** annotation is used to handle multipart requests, such as file uploads.

handleFileUpload(@RequestPart("file") MultipartFile file)

**YAML and properties file**

Both .yml (YAML) and .properties files are used for configuration in Spring Boot applications

**.properties** :

1. Format : Key-value pairs, each line represents a single property

spring.datasource.username=user

2. Hierarchy : uses dot notation to represent data

3. Lists: represented using array indices

servers[0].ip=127.0.0.1

servers[0].path=/path1

servers[1].ip=127.0.0.2

servers[1].path=/path2

4. Profiles : application-dev.properties, application-prod.properties

**.yml:**

1. Format : Uses identation to represent hierarchical data

2. Uses colon notation

3. Lists : represented using dashes

servers:

- ip: 127.0.0.1

path: /path1

- ip: 127.0.0.2

path: /path2

4. Profiles : Multiple profiles can be defined in single file using --- separator

StringBuilder vs StringBuffer

StringBuilder sb = new StringBuilder("Hello");

* **Use**StringBuilder: When you are working in a single-threaded environment and need better performance.
* **Use**StringBuffer: When you are working in a multi-threaded environment and need thread safety.

**@EnableAutoConfiguration Annotation**:

Spring Boot achieves auto-configuration by automatically based on the dependencies we added in pom.xml

* This annotation is crucial for enabling auto-configuration in a Spring Boot application. It tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

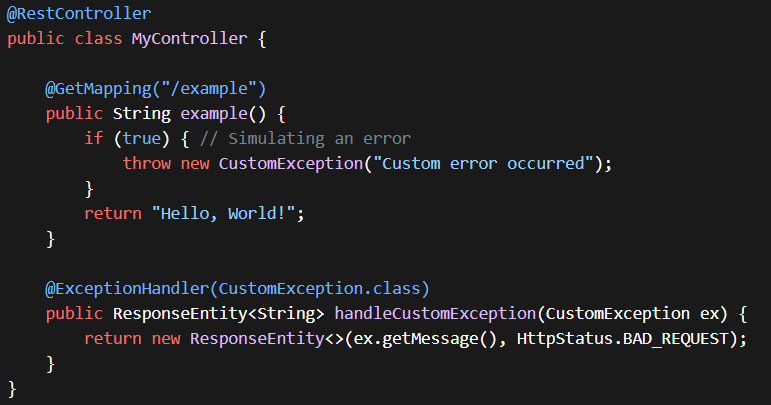
**new ResponseEntity<>(“msg”,headers,HttpStatus.OK)**in Spring Boot is a powerful class that allows you to customize the entire HTTP response, including the status code, headers, and body.

We will have full control over the HTTP Response

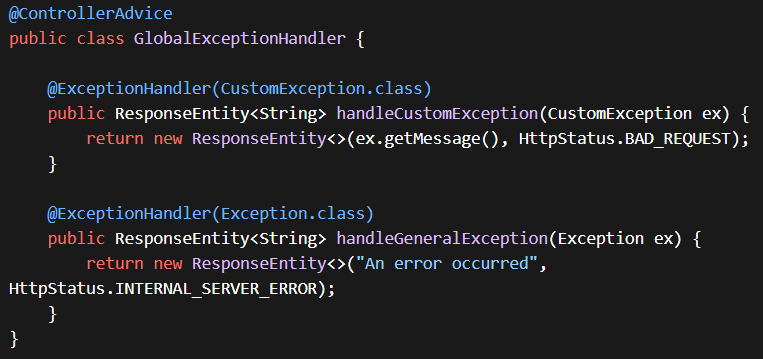
**Exception Handling**

Spring Boot provides default exception handling, which returns a generic error response when an exception occurs, this default doesn’t provide meaningful information to client

We can use **@ExceptionHandler** annotation to handle specific exceptions in our controller.



For global exception handling, you can use **the @ControllerAdvice**annotation. This allows you to handle exceptions across the entire application in a centralized manner.



Service layer exceptions can be handled using **try, catch** blocks or use standard exception handling like throw new Exception and then propagate exception to controller layer, where they can be handled by @ExceptionHandler methods

**How to scale a java application to handle high traffic**

1) breaking down the application to microservices, specific parts which experience more load can be scaled independently

2) load balancing

3) we can cache frequently accessed data in memory which reduces the load on database

4) make use of some monitoring tools to identify CPU utilization and performance bottlenecks

**How to Secure Rest Endpoints**

**What is a circular dependency and how to handle them**

Circular depend is when one or more entities dependent on each other, this can lead to issues such as infinite loops. Suppose if two beans are in circular dependency, spring does not know which one to create first.

1.We can use setter injection to handle circular dependency

2. @Lazy

3. manually fetch bean from application context

**What is binding in java springboot**

In springboot binding refers to mapping incoming data to java objects. This is crucial for building web applications

1. Data Binding

Maps HTTP request parameters to methods arguments

2. Configuration property binding

Spring Boot allows you to bind .yml properties to java objects using @ConfigurationProperties

my:

config:

name: "John"

age: 30

@Component

@ConfigurationProperties(prefix = "my.config")

public class MyConfig {

private String name;

private int age;

// getters and setters

}

@SpringBootApplication

@EnableConfigurationProperties(MyConfig.class)

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

**KEY CLOAK**

Open source identity and access management solution to secure java application with minimum efforts

Key features of keycloak include

1. SSO -> login once to gain access to multiple applications without the need to login again

2. Social Login

3. it also provides web-based admin console for role, user and permission management

4. supports standard protocols like OAuth 2.0

**Authentication** is the process of verifying the identity of a user -> password, biometric, OTP’s

**Authorization** determines what an authenticated user is allowed to do. -> Role based access

**Circuit Breaker Pattern**

Resilience4j is a lightweight fault tolerance library

**Screening**

Does java allow overriding static methods ?

A. No.

* If a derived class defines a static method with the same signature as a static method in the base class, it’s not considered overriding. Instead, it’s method hiding.

Q. Can we overload constructors ?

A. Yes.

Q. Can we declare an abstract method private ?

A. No.

Q. Does Spring Boot support batch processing?

A. Yes.

Q. Does Java support multiple inheritance?

A. No

* To avoid diamond problem, java language creators avoided multiple inheritance

Q. Can the value of a final variable be changed once it is initialized?

A. No

Q. Does Java support automatic garbage collection?

A. Yes

* Garbage collector automatically identifies and deletes these unreachable objects to free up memory
* **System.gc()**: This method suggests that the JVM perform garbage collection, but it is not guaranteed to do so.

Q. Is it possible to disable Spring Boot’s auto-configuration feature?

A. Yes, we can selectively exclude few classes from auto-configuration

* @SpringBootApplication(exclude = { MongoAutoConfiguration.class, MongoDataAutoConfiguration.class })
* Alternatively, you can disable auto-configuration using property files

spring.autoconfigure.exclude=\

org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration, \ ..

@EnableAutoConfiguration, Spring Boot achieves auto-configuration by automatically based on the dependencies we added in pom.xml

The @ComponentScan annotation is used to specify the packages to scan for annotated components.

By default, Spring Boot scans the package of the main application class and its sub-packages

**DESIGN PATTERNS**

They are standard solutions to common problems that takes place in software development, they are kind of best practices defined by software developers.

They are divided into 3 types

1. creational design patterns

* These patterns deal with object creation mechanisms.

2. structural design patterns

* These patterns deal with the composition of classes and objects.

3. behavioral design patterns

* These patterns deal with how objects interact and communicate

**Java Project from Scratch**

**1.** add dependencies

Lombok

Spring web

Spring data jpa

Spring security

Mysql driver

2. add application properties

spring.application.name=jwt  
spring.datasource.url=jdbc:mysql://localhost:3306/jwtcapstone  
spring.datasource.username=root  
spring.datasource.password=Prudhvi  
spring.jpa.hibernate.ddl-auto=create  
spring.jpa.show-sql=true  
spring.jpa.properties.hibernate.format\_sql=true  
server.port=8089

@Entity -> is a POJO class representing a table in a relational database

Each instance of an entity corresponds to a row in that table

Bridge between java and relational database tables

**Understanding JWT Authentication and the JwtAuthenticationFilter**

**What Is JWT Authentication?**

JWT (JSON Web Token) is a compact self-contained token used for transmitting information between two parties

Token Structure:

* + A JWT consists of three parts: header, payload (claims), and signature.
  + The header contains information about the algorithm used for signing.

HMAC (Hash-based Message Authentication Code) with SHA-256 (Secure Hash Algorithm 256-bit) is a widely used combination.

It involves creating a hash-based signature using a secret key and the SHA-256 hash function.

HMAC-SHA256 is efficient and provides strong security.

* + The payload contains claims (e.g., user ID, roles, expiration time).
  + The signature ensures the token’s integrity and authenticity.

If any part of the header or payload is modified (even a single character), the hash will change.

Only someone with the secret key (known only to the server) can generate a valid signature.

1. How It’s Used:
   * During authentication, the server generates a JWT and sends it to the client (usually as an HTTP header).
   * The client includes the JWT in subsequent requests (e.g., in the Authorization header).
   * The server validates the token, extracts claims, and grants access based on the information within it.

**The Role of JwtAuthenticationFilter**

The JwtAuthenticationFilter is a custom filter within Spring Security that handles JWT-based authentication. Its primary responsibilities include:

1. Extracting and Validating JWT:
   * When a request hits your Spring Boot application, the JwtAuthenticationFilter intercepts it.
   * It extracts the JWT (usually from the Authorization header) and validates its signature.
   * If the token is valid, the filter proceeds; otherwise, it denies access.
2. Setting Authentication in Security Context:
   * Upon successful validation, the filter creates an Authentication object.
   * This object represents the authenticated user (with roles, authorities, etc.).
   * It sets this Authentication object in the SecurityContextHolder.
   * Now subsequent filters and controllers can access the authenticated user’s details.
3. Integration with Spring Security Filter Chain:
   * The JwtAuthenticationFilter is part of the Spring Security filter chain.
   * It executes before other security-related filters (e.g., authorization filters).
   * By extending OncePerRequestFilter, it ensures that it runs only once per request.

**When Does It Execute?**

1. Request Flow:
   * When a client sends an HTTP request to your Spring Boot application, the request goes through the filter chain.
   * The JwtAuthenticationFilter is one of the filters in this chain.
   * It executes early in the chain, before other security filters.
2. Execution Triggers:
   * The JwtAuthenticationFilter executes for every incoming request.
   * It checks whether the request contains a valid JWT.
   * If so, it sets the authenticated user in the security context.
   * If not, it allows the request to proceed (unauthenticated).

**Example Code Snippet**

**Here’s a simplified example of how you might configure and use a JwtAuthenticationFilter in your Spring Boot application:**

**Java**

**public class JwtAuthenticationFilter extends OncePerRequestFilter {**

**@Override**

**protected void doFilterInternal(**

**HttpServletRequest request,**

**HttpServletResponse response,**

**FilterChain filterChain) throws ServletException, IOException {**

**// Extract JWT from request (e.g., from Authorization header)**

**String token = extractToken(request);**

**// Validate the token (verify signature, expiration, etc.)**

**if (isValidToken(token)) {**

**Authentication authentication = createAuthenticationFromToken(token);**

**SecurityContextHolder.getContext().setAuthentication(authentication);**

**}**

**// Continue with the filter chain**

**filterChain.doFilter(request, response);**

**}**

**// Other methods for token extraction, validation, and user creation**

**}**

**The Main Difference between OAuth2 and OpenID Connect is that OAuth2 is only concerned with authorization, while OpenID connect is also concerned with authentication. Authorization means granting access to resources, while authentication means verifying the identity of a user.**

1. **What is Authentication and Authorization**

Ans : 1. Authentication is proving someone’s identity

i.e:- it’s like verifying an employee using their ID card before entering the campus

2. Authorization is giving access to a particular resource that the user has access to

i.e:- once user enters the campus, next step would be to what all ODC’s he can go

2. **create an entity class**

@Entity //this tells the application that it represents a table in database and this allows JPA to manage it

@Table(Name=”user”) // maps this class to table in database

@Data //@ToString(), @EqualsAndHashCode, @RequiredArgsConstructor

Class User{

@Id

@Generatedvalue(strategy=generationType.IDENTITY)

Private Long id;

Private String name;

Private String age;

}

3. **How to Implement JWT token**

We make use of few dependencies to implement JWT token, JJWT helps in taking a string

and transforming it into a secured and structured token

1. jjwt-api contains classes and interfaces for creating, parsing

and verifying tokens

2. jjwt-impl for customizing the implementation (contains actual impl)

3. jjwt-jackson – helps in packing custom payload with jwt and extracting

them as java objects

4. **how to generate JWT token**

**io.jsonwebtoken.Jwts;**

public String generateToken(String subject, UserRole userRole){  
 return Jwts  
 .*builder*()

.claims(new HashMap<>( Map.*of*("role", userRole.getRole().name())))  
 .subject(subject)  
 .issuedAt(new Date(System.*currentTimeMillis*()))  
 .expiration(new Date(System.*currentTimeMillis*() + 24\*60\*60\*1000))  
 .signWith(getSigninKey())  
 .compact();  
}

private SecretKey getSigninKey(){  
 return Keys.*hmacShaKeyFor*(keyBytes);  
}

String jwt = jwtService.generateJwt("Prudhvi", UserRole.ADMIN);

**import java.util.function.Function;**

1.Function<Person, String> subjectExtractor = Person::getSubject;

2. Person::getSubject is equivalent to the lambda expression

(person) -> person.getSubject()

3. subjectExtractor.apply(person) is used to call the function

5.**How to get data from database, what classes to use to get the data**

Extending JPArepository to repository interface, entity, controller and service classes are used to get the data

**6. What java versions you worked on? What differences have you observed**

I worked with java 17 and now working on java 21.

Having worked on java 17, most of the features that we use are from java 8

Apart from java 8 features,

Features introduced in java 17 are Sealed classes

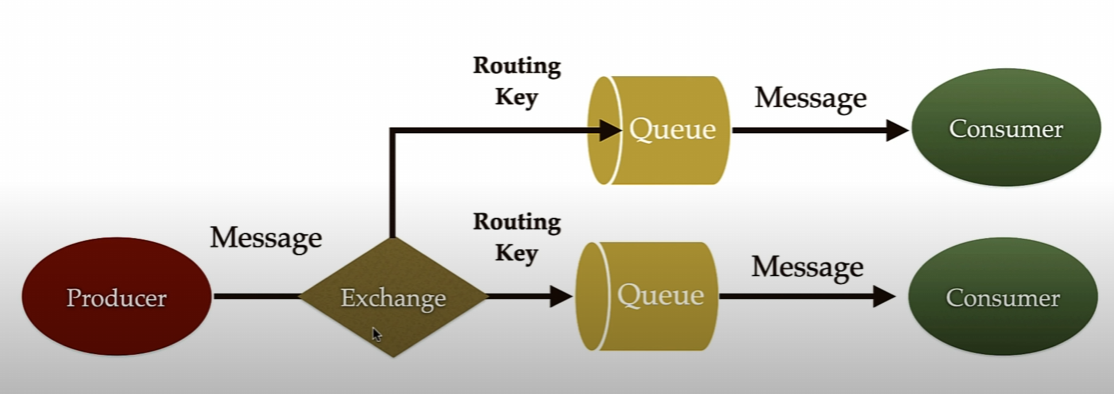
Sealed interface Shape permits Circle, Rectangle, Triangle { }

* Java has introduced few methods in the ‘java.lang.Character’ class to work with different types of emoji’s properties
* it also added a new method to StringBuilder and StringBuffer to repeat a certain character sequence
* In the preview features of java 21, it released StringTemplates but then later it is removed in version 23
* It also improved garbage collection in java 21

**Queue Services** (**RabbitMQ**)

This RabbitMQ acts as a broker between producer and consumer and it also provides us with a temporary storage named as queue

This also helps us in asynchronous communication between applications.



Routing key is used by exchange to route messages to queues, this is called binding

@Bean  
public MessageConverter messageConverter() {  
 return new Jackson2JsonMessageConverter();  
}  
  
@Bean  
public AmqpTemplate amqpTemplate(ConnectionFactory connectionFactory) {  
 RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);  
 rabbitTemplate.setMessageConverter(messageConverter());  
 return rabbitTemplate;  
}

1.Coding

2.Redis cache

3.revision

4.microservices

5.MYSQL

6.Design Patterns

Spring integrates various design patterns among them below are few

**SINGLETON DESIGN PATTERN**

This is a creational design pattern and it aims to have only one instance of object throughout the application

Using a Singleton logger ensures that only one instance is responsible for writing to the log file.

The Loggers created by LoggerFactory follow singleton pattern

* When you request a logger using LoggerFactory.getLogger(MyClass.class), SLF4J checks if a logger for that class already exists. If it does, it returns the existing instance; otherwise, it creates a new one.

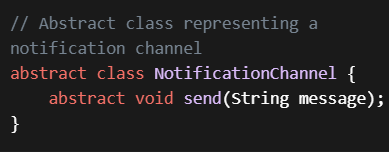
Springboot by default provides singleton scope for all beans. It does have shortfalls, for that we have to explicitly implement singleton design pattern if the variables in the instance are mutable.

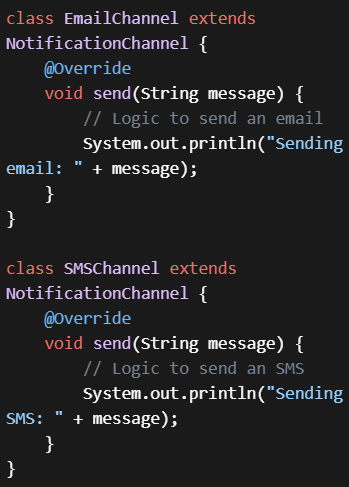
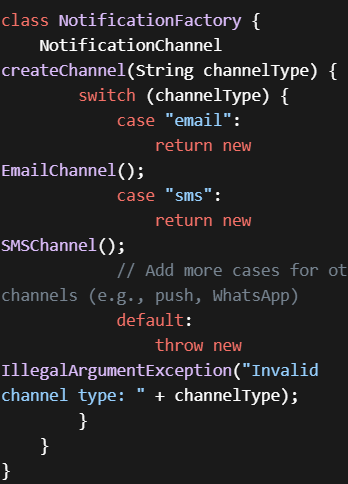
**FACTORY DESIGN PATTERN**

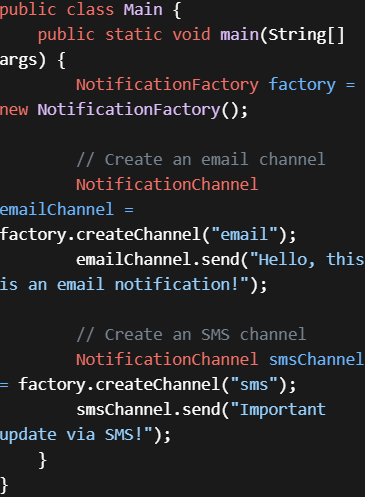
The Factory Method pattern allows you to create objects without specifying their exact class. Instead, you define a factory method responsible for creating instances.

This is used when we have a multiple sub class implementation of a super class and based on input we want to return a new instance

Factory class has a static method which returns the instance of subclass based on input





**TEMPLATE METHOD DESIGN PATTERN**

The Template Method pattern defines a common structure for algorithms but allows specific steps to be implemented by subclasses. This is a behavioral design pattern

Spring provides template classes (e.g., RabbitTemplate ) that encapsulate common tasks (e.g: convertAndSend a string to queue to be consumed by a consumer ) while allowing customization.(like adding a JSON converter method for transferring JSON data)

