Index

Spring Boot Questions:

1. What is the Global Exception Handler?

A Global Exception Handler is a mechanism in a software application that provides a unified way of handling exceptions that may occur throughout the application. The purpose of a Global Exception Handler is to catch exceptions that are thrown by different parts of the application and handle them in a consistent and centralized way.

In Java, a Global Exception Handler is usually implemented as a catch-all exception handler that captures all exceptions thrown in the application and processes them in a consistent manner. For example, in a web application, the Global Exception Handler can be used to catch exceptions thrown by controllers and provide a unified error response to the client.

In Spring Boot, the Global Exception Handler can be implemented by creating a custom class annotated with `@ControllerAdvice` and implementing the `@ExceptionHandler` annotation on a method that will handle the exception. This method can then return a response to the client or redirect the client to another page.

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleException(Exception ex) {
        ErrorResponse errorResponse = new ErrorResponse();
        errorResponse.setMessage(ex.getMessage());
        errorResponse.setStatus(HttpStatus.INTERNAL_SERVER_ERROR.value());
        return new ResponseEntity<>(errorResponse, HttpStatus.INTERNAL_SERVER_ERROR);
    }y
}
```

In this example, the GlobalExceptionHandler class is annotated with `@ControllerAdvice`, indicating that it is a global exception handler. The handleException method is annotated with `@ExceptionHandler` and takes an Exception object as its argument. This method is called whenever an Exception is thrown in the application.

The `handleException` method creates an instance of the ErrorResponse class and sets its message and status code. It then returns a ResponseEntity with the error response and an HTTP status code of `HttpStatus.INTERNAL_SERVER_ERROR`.

This Global Exception Handler can be used to catch any exception thrown in the application and provide a consistent response to the client. The error response and HTTP status code can be customized as needed, providing a flexible and centralized way of handling exceptions in the application.

2. What is a Bean in Spring Boot?

A bean in Spring Boot is an object that is managed by the Spring framework. Beans are created and managed by the Spring container, which is responsible for their lifecycle and the resolution of their dependencies.

In Spring Boot, a class can be defined as a bean by annotating it with `@Component` or one of its variants, such as `@Service`, `@Repository`, or `@Controller`. These annotations tell the Spring container to create an instance of the class and manage its lifecycle.

Beans can have dependencies, which are other beans that they require to perform their tasks. In Spring Boot, dependencies can be injected into a bean using constructor arguments or setter methods. The Spring container is responsible for creating and managing the dependencies and injecting them into the bean as needed.

In summary, a bean in Spring Boot is a class that is managed by the Spring framework and has its lifecycle and dependencies managed by the Spring container. The use of beans enables the use of other features provided by the Spring framework, such as aspect-oriented programming, event handling, and configuration management.

3. What is `@Bean` annotation?

The `@Bean` annotation in Spring Boot is used to define a bean that is managed by the Spring framework. The `@Bean` annotation can be used in a configuration class to define a method that returns an instance of a bean.

For example, consider the following code:

```
@Configuration
public class AppConfig {

    @Bean
    public MyBean myBean() {
        return new MyBean();
    }
}
```

In this example, the AppConfig class is annotated with `@Configuration`, indicating that it is a configuration class. The myBean method is annotated with `@Bean`, indicating that it returns an instance of the MyBean class, which will be managed by the Spring container.

The `@Bean` annotation is useful when you need to define a bean in a configuration class, rather than as a component with one of the `@Component` annotations. The `@Bean` annotation can be used to customize the behavior of the bean and its dependencies, such as setting properties, registering event listeners, and configuring the behavior of other beans.

In summary, the `@Bean` annotation in Spring Boot is used to define a bean that is managed by the Spring framework, and provides a way to customize the behavior of the bean and its dependencies in a configuration class.

4. What is Setter Injection?

Setter injection is a form of dependency injection in which the dependencies are injected into a bean using setter methods. In setter injection, the dependencies are passed to the bean using the setter methods rather than the constructor.

For example, consider the following code:

```
@Service
public class MyService {

    private MyRepository myRepository;

    @Autowired
    public void setMyRepository(MyRepository myRepository) {
        this.myRepository = myRepository;
    }

    // ...
}

@Repository
public class MyRepository {
    // ...
}
```

In this example, the `MyService` class is annotated with `@Service`, indicating that it is a service bean. The `MyService` class has a setter method, `setMyRepository`, that takes an instance of `MyRepository` as an argument. The `MyRepository` class is annotated with `@Repository`, indicating that it is a repository bean.

The `@Autowired` annotation on the `setMyRepository` method of the `MyService` class indicates that the `MyRepository` bean should be automatically injected into the `MyService` bean using setter injection.

In this example, the `MyService` bean is dependent on the `MyRepository` bean. The `MyRepository` bean is created and managed by the Spring container, and an instance of it is automatically injected into the `MyService` bean using the `setMyRepository` setter method when it is created.

In summary, setter injection is a form of dependency injection in which the dependencies are injected into a bean using setter methods. In setter injection, the dependencies are passed to the bean using the setter methods rather than the constructor, and is used to manage the dependencies between beans that are managed by the Spring container.

5. What is Spring Bean's Lifecycle?

The life cycle of a Spring bean refers to the different stages a bean goes through from its creation to its destruction. A Spring bean can be thought of as an object that is managed by the Spring framework, and the life cycle of a Spring bean is managed by the Spring container.

The main stages in the life cycle of a Spring bean are:

- Bean instantiation: The Spring container creates an instance of the bean by using one of the following mechanisms:
- Constructor injection: The container creates an instance of the bean by calling its constructor.
    a. Static factory method: The container calls a static factory method to create an instance of the bean
    b. Instance factory method: The container calls a factory method on an existing bean to create an instance of the bean.

- Bean wiring: The container wires up the dependencies between beans by injecting the dependencies into the appropriate bean. This process is done using either constructor injection, setter injection, or field injection.
- Bean post-processing: After the bean has been wired up, the container can perform post-processing on the bean. This stage is used for tasks such as setting bean properties, performing custom initialization, and registering event listeners.
- Bean initialization: Once the post-processing has been completed, the container initializes the bean. This stage is used for tasks such as starting threads, connecting to databases, or opening network connections.
- Bean destruction: Finally, when the container shuts down, it destroys the beans. This stage is used for tasks such as releasing resources, closing network connections, and stopping threads.

The life cycle of a Spring bean is managed by the Spring container, and the programmer can customize the life cycle by using various hooks, such as `@PostConstruct` and `@PreDestroy`, to perform custom initialization and destruction logic.

In summary, the life cycle of a Spring bean refers to the different stages a bean goes through from its creation to its destruction, and is managed by the Spring container. The life cycle of a Spring bean can be customized by using various hooks to perform custom initialization and destruction logic.

6. How can we set the spring bean scope and what supported scopes does it have?

In Spring, the scope of a bean determines the number of instances of a bean that exist in the Spring container and how they are shared among the different components that depend on them.

Spring supports the following scopes for beans:

- singleton: This is the default scope. Only one instance of the bean will be created for the entire application and will be shared among all components that depend on it.
- prototype: A new instance of the bean is created every time a component requests it.
- request: A new instance of the bean is created for each HTTP request in a web application.
- session: A new instance of the bean is created for each HTTP session in a web application.
- application: A single instance of the bean is created for a single ServletContext in a web application.
- websocket: A new instance of the bean is created for each WebSocket session in a web application.

To set the scope of a bean, you can use the `@Scope` annotation on the class definition or in the XML configuration file. For example:

```
@Scope("singleton")
@Component
public class MyBean {
//   ...
}
```

In summary, the scope of a bean in Spring determines the number of instances of a bean that exist in the Spring container and how they are shared among the different components that depend on them. Spring supports several scopes, including singleton, prototype, request, session, application, and websocket. The scope of a bean can be set using the `@Scope` annotation or in the XML configuration file.

7. What is the difference between new based object creation and defining a class as bean in spring boot?

The difference between creating an object using the new operator and defining a class as a bean in Spring Boot lies in how the object is managed and how its dependencies are resolved.

When an object is created using the new operator, it is a standalone object that has complete control over its own lifecycle. It must manually manage its dependencies, either by constructing them itself or by receiving them as constructor arguments. The resulting object is tightly coupled to its dependencies, which can make it difficult to change or test.

In contrast, when a class is defined as a bean in Spring Boot, the object is managed by the Spring framework. The Spring container is responsible for creating and managing the lifecycle of the bean, including resolving its dependencies. When a bean is created, its dependencies can be automatically injected by the Spring container, either through constructor arguments or through setter methods.

This separation of concerns makes it easier to manage and maintain the objects in a Spring Boot application. It also enables the use of other features provided by the Spring framework, such as aspect-oriented programming, event handling, and configuration management.

In summary, creating an object using the new operator is a direct and straightforward way to instantiate an object, but it does not provide any flexibility in terms of how the object and its dependencies are managed. Defining a class as a bean in Spring Boot provides more control over the lifecycle of the object and its dependencies, making it easier to manage and maintain the application.

8. What is spy in java testing?

In Java testing, a spy is a type of test double that provides a partial implementation of the object being tested. A spy can be used to control the behavior of the object in question, to verify the interactions between objects during a test, or to capture information about how the object is being used.

Spies are often used in combination with mock objects to provide a more fine-grained control over the behavior of objects in a test environment.

For example, if you have a service class with a method that makes an HTTP request to an external API, you can use a spy to intercept the call to the API and return a fake response, without actually sending a real request. This allows you to test the behavior of your code without relying on external services and without having to worry about the stability and reliability of those services.

In Java testing frameworks such as JUnit and Mockito, you can create a spy using the spy method. The spy will typically be created from an instance of the class being tested, and the spy's behavior can then be controlled and verified through various methods provided by the testing framework.

Here's an example using Mockito, a popular Java testing framework:

Consider a simple service class CalculatorService:

```
public class CalculatorService {

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

We can write a test for this class using a spy as follows:

```
@Test
public void testAdd() {
    // Create a spy of the CalculatorService
    CalculatorService calculatorService = spy(new CalculatorService());

    // Control the behavior of the add method
    when(calculatorService.add(1, 2)).thenReturn(10);

    // Verify the behavior of the add method
    int result = calculatorService.add(1, 2);
    assertEquals(10, result);
    verify(calculatorService).add(1, 2);
}
```

In this example, we first create a spy of the CalculatorService using the spy method. We then control the behavior of the add method by telling the spy to return a fixed value of 10 when called with arguments 1 and 2. Finally, we verify the behavior of the add method by asserting that the result is equal to 10 and by using the verify method to ensure that the add method was indeed called with arguments 1 and 2.

Note that when using a spy, only the methods that are being controlled or verified will be executed, while other methods will be executed normally. This allows you to test only the behavior that you're interested in, without having to worry about the other parts of the code.

9. What is `@Qualifier` annotation?

The `@Qualifier` annotation in Spring is used to specify a specific implementation of a bean when multiple implementations of the same interface or class are present. The `@Qualifier` annotation can be applied to a constructor argument, a method parameter, or a field, and it works in conjunction with the `@Autowired` annotation to resolve the bean that should be injected.

For example, consider the following scenario:

```
public interface Engine {
    void start();
}

@Component
@Qualifier("diesel")
public class DieselEngine implements Engine {
    // implementation details
}

@Component
@Qualifier("petrol")
public class PetrolEngine implements Engine {
    // implementation details
}
```

Here, we have two implementations of the Engine interface, DieselEngine and PetrolEngine. By annotating each implementation with `@Qualifier`, we give them a specific name, "diesel" and "petrol", respectively. Now, when we want to inject one of these engines into a class, we can use the `@Autowired` and `@Qualifier` annotations to specify which engine we want:

```
@Component
public class Car {

    private final Engine engine;

    @Autowired
    public Car(@Qualifier("diesel") Engine engine) {
        this.engine = engine;
    }

    // other methods
}
```

In this example, the constructor of the Car class takes an Engine as an argument and annotates it with `@Autowired` and `@Qualifier("diesel")`. This tells Spring to inject the DieselEngine into the Car class, because it is the only engine implementation with the "diesel" qualifier. If we wanted to inject the PetrolEngine instead, we would simply change the `@Qualifier` value to "petrol".

10. What is a stateless and stateful bean?

In the context of Spring Framework, a bean can be either stateless or stateful.

A stateless bean is a bean that doesn't maintain any state between method invocations. For example, a simple utility class that doesn't store any information between invocations would be considered a stateless bean. These types of beans are generally thread-safe and can be shared among multiple clients without issue.

A stateful bean, on the other hand, does maintain state between method invocations. For example, a shopping cart that keeps track of items added to it would be considered a stateful bean. These types of beans are generally not thread-safe and cannot be shared among multiple clients.

In general, it is recommended to use stateless beans whenever possible, as they are easier to design, implement, and maintain than stateful beans.

11. What are different ways to create a bean?

There are three main ways to create a bean in Spring Framework:

- XML Configuration: You can create a bean by defining it in an XML file (usually named as "applicationContext.xml").

```
<bean id="exampleBean" class="com.example.ExampleBean"/>
```

- Annotation-based Configuration: You can create a bean using annotations on a class, such as `@Component` or `@Bean.`

```
@Component
public class ExampleBean {
  // ...
}
```

- Java-based Configuration: You can create a bean using Java code, such as a configuration class annotated with `@Configuration`.

```
@Configuration
public class BeanConfig {
  @Bean
  public ExampleBean exampleBean() {
    return new ExampleBean();
  }
}
```

Note: These are the traditional methods to create a bean in Spring. With the introduction of Spring Boot, you can also use auto-configuration and starter dependencies to create and manage beans.

12. How does mockito work?

Mockito is a popular Java library for creating test doubles, also known as mocks, for testing purposes. It allows you to isolate the code you're testing from the dependencies it interacts with.

Here's how it works:

- First, you need to create a mock object using the Mockito.mock method or the @Mock annotation:

```
ExampleDependency dependency = Mockito.mock(ExampleDependency.class);
```

- Next, you can define the behavior of the mock object. For example, you can specify what it should return when a certain method is called:

```
Mockito.when(dependency.getData()).thenReturn("test data");
```

- Finally, you can use the mock object in your test cases, just like you would use a real instance of the dependency:

```
ExampleService service = new ExampleService(dependency);
String result = service.getResult();
Assert.assertEquals("test data", result);
```

- After the test case has run, you can verify that the mock object was used as expected:

```
Mockito.verify(dependency).getData();
```

This way, you can test your code in isolation, without having to worry about the behavior of its dependencies. Additionally, you can use mock objects to test error conditions, edge cases, and other scenarios that are difficult to reproduce with real instances.

13. How to test a method which has void return type?

To test a method with a void return type, you need to verify the behavior of the method, such as the state of the object after the method has been called or the interactions with its dependencies. Here are some ways to do that:

- Verify the state of the object: You can use assertions to verify that the state of the object has changed as expected after the method has been called.

```
ExampleClass example = new ExampleClass();
example.voidMethod();
assertTrue(example.isStateChanged());
```

- Verify interactions with dependencies: If the void method interacts with other objects, you can use mock objects to verify that the interactions happened as expected.

ExampleDependency dependency = Mockito.mock(ExampleDependency.class);
ExampleClass example = new ExampleClass(dependency);
example.voidMethod();
Mockito.verify(dependency).doSomething();

- Verify exceptions: If the void method is expected to throw an exception under certain circumstances, you can use try-catch blocks or the expected attribute of the @Test annotation to verify that the correct exception is thrown.

```
ExampleClass example = new ExampleClass();
try {
  example.voidMethod();
  fail("Exception expected");
} catch (ExampleException e) {
  assertEquals("Exception message", e.getMessage());
}
```

These are just a few examples of how to test void methods. The approach you choose depends on the specific requirements of your method and the behavior you want to verify.

14. What is Power Mocito and How does it work?

Power Mockito is a library that extends the functionality of the popular Mockito mocking framework for Java. It provides additional features that allow developers to mock and test complex code, even in situations where the code uses final or static methods, or private constructors.

Mockito is a Java mocking framework that allows developers to create mock objects for testing purposes. Mock objects are objects that simulate the behavior of real objects, allowing developers to isolate and test specific parts of their code without needing to rely on external dependencies.

While Mockito is a powerful tool for mocking Java objects, it has some limitations. For example, it cannot mock final or static methods, and it cannot create mock objects for classes with private constructors. This is where Power Mockito comes in - it provides additional functionality that allows developers to mock these types of objects.

Power Mockito works by using bytecode manipulation to modify the behavior of the Java virtual machine (JVM) at runtime. When the developer writes a test case that uses Power Mockito, the library generates bytecode that modifies the behavior of the JVM to allow the mocking of final and static methods, or to create mock objects for classes with private constructors.

Here's an example of how Power Mockito can be used to mock a final method:

```
public class MyClass {
    public final String getMyString() {
        return "Real string";
    }
}

@Test
public void testFinalMethodMocking() {
    MyClass myClass = mock(MyClass.class);
    when(myClass.getMyString()).thenReturn("Mocked string");

    assertEquals("Mocked string", myClass.getMyString());
}
```

In this example, we have a class MyClass with a final method getMyString(). Normally, this method cannot be mocked using Mockito. However, by using Power Mockito, we can mock this method and specify its return value using the when method.

By using Power Mockito, developers can test their code more thoroughly, even in situations where the code uses final or static methods, or private constructors. This can help to ensure that the code is robust and free from bugs, even in complex applications with many dependencies.

15. What is a prototype bean, how is it configured?

A prototype bean in Spring is a bean definition that returns a new instance of the bean every time it is requested. This is in contrast to singleton beans, which return the same instance every time.

Here's how you can configure a prototype bean in Spring:

```
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@Component
public class PrototypeBean {
  // ...
}
```

Note that when you configure a bean as a prototype, each time it is requested from the application context, a new instance of the bean is created and returned. This allows you to create multiple instances of the same bean, each with its own state and behavior.

16. How does spring boot implements serialization of object internally?

In Spring Boot, serialization of objects is implemented using Java's built-in serialization mechanism. When an object is serialized, its state is converted into a stream of bytes that can be transmitted over a network or stored in a file. The process of serialization involves converting the object's fields into a binary format that can be reconstructed into the original object when it is deserialized.

Spring Boot uses the Java Serialization API to perform serialization and deserialization of objects. The ObjectOutputStream class is used to serialize objects, and the ObjectInputStream class is used to deserialize them.

Here's an example of how serialization can be performed in Spring Boot:

```
public class MyObject implements Serializable {
    private String name;
    private int age;

    public MyObject(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

// Serializing an object
MyObject myObject = new MyObject("John", 30);
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
ObjectOutputStream objectOutputStream = new ObjectOutputStream(outputStream);
objectOutputStream.writeObject(myObject);
byte[] bytes = outputStream.toByteArray();

// Deserializing an object
ByteArrayInputStream inputStream = new ByteArrayInputStream(bytes);
ObjectInputStream objectInputStream = new ObjectInputStream(inputStream);
MyObject deserializedObject = (MyObject) objectInputStream.readObject();
```

In this example, we have a class MyObject that implements the Serializable interface. This indicates to the Java Serialization API that this class can be serialized and deserialized.

To serialize an object of type MyObject, we create an ObjectOutputStream and write the object to a ByteArrayOutputStream. The writeObject method serializes the object and writes it to the output stream.

To deserialize an object, we create an ObjectInputStream and read the serialized data from a ByteArrayInputStream. The readObject method reads the serialized data and reconstructs the original object.

By using Java's built-in serialization mechanism, Spring Boot can serialize and deserialize objects with ease, allowing developers to transmit data over a network or store it in a file in a binary format. However, it's worth noting that there are other serialization mechanisms available that may be more efficient or offer additional features, depending on the use case.

17. What is Jackson Library?

Jackson is a popular Java-based library for JSON serialization and deserialization. It provides a set of tools and annotations that allow developers to convert Java objects to JSON format and vice versa.

Jackson provides a fast, efficient, and flexible way to work with JSON data in Java applications. It can handle complex JSON structures and supports a wide range of features such as:

- Support for JSON parsing and generation
- Annotations for controlling serialization and deserialization behavior

- Support for handling polymorphic types
- Support for streaming data processing
- Support for handling of date/time formats

Jackson can be used with various Java frameworks such as Spring, Hibernate, and JAX-RS, making it a popular choice for developers working on web applications and APIs.

Here's an example of how to use Jackson to serialize a Java object to JSON:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

// Serialization example
ObjectMapper objectMapper = new ObjectMapper();
Person person = new Person("John", 30);
String jsonString = objectMapper.writeValueAsString(person);
System.out.println(jsonString); // Output: {"name":"John","age":30}
```

In this example, we have a class Person with two fields name and age. To serialize an object of type Person, we create an instance of ObjectMapper and call the writeValueAsString method, passing in the object to be serialized. The method returns a JSON string representing the object.

Jackson is widely used in the Java community and has become a de facto standard for working with JSON data in Java applications.

18. What is spring security and what library do we import to use it?

Spring Security is a framework for securing Java applications. It provides a flexible and comprehensive security solution for both web-based and method-level security. With Spring Security, you can secure your application by defining authentication and authorization rules, such as who is allowed to access what resources and what actions they can perform.

To use Spring Security in your application, you need to import the spring-security-core library. This library provides the core security features, such as authentication and authorization. You can also import additional libraries, such as spring-security-web for web security and spring-security-config for configuration-based security.

Here's an example of how to include Spring Security in your project using Gradle:

```
dependencies {
  implementation 'org.springframework.security:spring-security-core:<version>'
  implementation 'org.springframework.security:spring-security-web:<version>'
  implementation 'org.springframework.security:spring-security-config:<version>'
}
```

Replace `<version>` with the desired version of Spring Security.

Once you have the library imported, you can configure Spring Security in your application by defining authentication and authorization rules, setting up user authentication and access control, and securing your web application or method-level security.

19. How to create a custom annotation using spring boot framework Java?

Creating a custom annotation in Spring Boot is a two-step process:

- Define the annotation: Create a new Java class and use the @Retention and @Target annotations to specify the retention policy and target element types for the annotation.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface CustomAnnotation {
  String value() default "";
}
```

In this example, the CustomAnnotation annotation can be used on methods, and it has an optional value parameter.

- Use the annotation: Apply the custom annotation to the methods or classes in your application that you want to annotate.

```
import org.springframework.stereotype.Service;

@Service
public class ExampleService {
  @CustomAnnotation("Example Value")
  public void exampleMethod() {
    // ...
  }
}
```

In this example, the ExampleService class has a method annotated with the CustomAnnotation annotation.

You can use reflection to retrieve the annotations and their values at runtime. This allows you to perform custom processing based on the annotations in your application. For example, you could use the custom annotation to trigger custom behavior in response to certain events or to configure certain components.

20. What is Reflection?

Reflection is a feature in Java that allows for the inspection of classes, interfaces, enums, annotations, and their properties at runtime. With reflection, we can obtain information about the methods, fields, and constructors of a class, and even create new instances of a class dynamically.

Reflection can be used in a variety of ways, such as creating dynamic proxies, implementing dependency injection frameworks, or creating custom serialization mechanisms. However, reflection can also introduce performance overhead and security risks, so it should be used judiciously.

Suppose you have a class called Person which has private attributes such as name, age, and address. Now, you want to access and modify these private attributes from outside of the class. This can be done using reflection in Java.

Here's an example code snippet that uses reflection to get and set the private attributes of the Person class:

```
import java.lang.reflect.Field;

public class Person {
    private String name;
    private int age;
    private String address;

    public Person(String name, int age, String address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }

    public static void main(String[] args) throws Exception {
        Person person = new Person("John Doe", 30, "123 Main St.");

        // Accessing private field name
        Field nameField = Person.class.getDeclaredField("name");
        nameField.setAccessible(true);
        String name = (String) nameField.get(person);
        System.out.println("Name: " + name);

        // Modifying private field age
        Field ageField = Person.class.getDeclaredField("age");
        ageField.setAccessible(true);
        ageField.setInt(person, 35);
        System.out.println("Age: " + person.age);
    }
}
```

In this example, we use reflection to access and modify the private fields of the Person class. We first use getDeclaredField() to get a reference to the private field, then we use setAccessible(true) to allow us to access and modify the private field. Finally, we use the get() and setInt() methods to get and modify the value of the field, respectively.

21. What is the difference between @Service, @Repository and @Controller annotation in Spring Boot and can either of these can be used interchangeably in any case?

In Spring Boot, the @Service, @Repository, and @Controller annotations are used to provide different functionality and to help organize your code. These annotations should not be used interchangeably because they have specific purposes.

@Service

The @Service annotation is used to indicate that a class is a service component in your application. A service is typically used to encapsulate business logic, perform data validation, and interact with data sources, such as databases or web services.

Here's an example of a service class with the @Service annotation:

```
@Service
public class ProductService {
    // ...
}
```

@Repository

The @Repository annotation is used to indicate that a class is a repository component in your application. A repository is typically used to interact with data sources, such as databases or web services, to retrieve or store data.

Here's an example of a repository class with the @Repository annotation:

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
    // ...
}
```

@Controller

The @Controller annotation is used to indicate that a class is a web controller component in your application. A controller is typically used to handle requests and return a response to a client, such as a web browser or mobile app.

Here's an example of a controller class with the @Controller annotation:

```
@Controller
public class ProductController {
    // ...
}
```

So, in summary, the main difference between @Service, @Repository, and @Controller annotations is their intended use. @Service is used for encapsulating business logic, @Repository is used for interacting with data sources, and @Controller is used for handling requests and returning a response.

While these annotations should not be used interchangeably, there are situations where a class can have more than one of these annotations. For example, a service class may also have the @Repository annotation if it needs to interact with a data source. Similarly, a controller class may also have the @Service annotation if it needs to perform business logic. However, it is important to ensure that the purpose of each annotation is clear and that each annotation is used for its intended purpose.

22. If spring creates singleton bean, and different apis are calling it all at once then how it is handled?

When a singleton bean in Spring is accessed by multiple threads, it is managed by the container in a thread-safe manner. The Spring framework provides synchronization mechanisms to ensure that only one thread can modify the state of the bean at a time, which prevents data corruption and race conditions.

In the case of a web application where different APIs are calling the same singleton bean, the container will create only one instance of the bean and share it among all the requests. Each request will be processed by a separate thread and the container will ensure that each thread has its own copy of any request-specific data, such as request parameters or session information.

The thread-safe behavior of a Spring singleton bean is achieved through the use of locks and synchronized blocks. When a method in the bean is called, the container will acquire a lock on the bean instance, which prevents other threads from accessing it. Once the method completes, the lock is released and the next thread can access the bean.

For example, consider the following singleton bean that maintains a counter:

```
@Service
public class CounterService {
    private int count = 0;

    public synchronized int increment() {
        return ++count;
    }
}
```

In this example, the increment() method is marked as synchronized, which means that only one thread can execute the method at a time. This ensures that the count is incremented atomically, preventing any race conditions that might occur if multiple threads were to access the count variable simultaneously.

So in summary, when multiple APIs access a singleton bean in Spring, the container manages the bean in a thread-safe manner using synchronization mechanisms to prevent data corruption and race conditions.

23. How does caching works in Spring boot? Give an example.

Caching is a powerful technique used in software development to improve application performance by reducing the response time for frequently accessed data. In Spring Boot, caching can be easily implemented using annotations and is supported by many caching providers, including Ehcache, Hazelcast, and Redis.

To use caching in a Spring Boot application, we first need to enable caching by adding the `@EnableCaching` annotation to a configuration class. This tells Spring to create a cache manager bean and to start caching annotations processing.

```
@Configuration
@EnableCaching
public class CacheConfig {
    @Bean
    public CacheManager cacheManager() {
        // create and configure a cache manager
    }
}
```

Next, we can add caching annotations to methods that should be cached. For example, the `@Cacheable` annotation can be used to indicate that a method's result should be cached. The first time the method is called with a given set of parameters, the method will be executed and its result will be stored in the cache. Subsequent calls to the same method with the same parameters will return the cached result without executing the method again.

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository repository;

    @Cacheable("products")
    public List<Product> getAllProducts() {
        return repository.findAll();
    }
}
```

In this example, we define a ProductService class that has a getAllProducts method that retrieves all products from a ProductRepository. We add the @Cacheable("products") annotation to the method to indicate that its result should be cached with the name "products". The first time the method is called, it will retrieve all products from the repository and store the result in the cache. Subsequent calls with the same parameters will return the cached result, which can significantly improve performance.

We can also use other caching annotations, such as @CachePut to update the cache, @CacheEvict to remove entries from the cache, and @CacheConfig to configure the caching behavior of a class or method.

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository repository;

    @Cacheable("products")
    public List<Product> getAllProducts() {
        return repository.findAll();
    }

    @CachePut(value = "products", key = "#product.id")
    public Product saveProduct(Product product) {
        return repository.save(product);
    }

    @CacheEvict(value = "products", key = "#id")
    public void deleteProductById(Long id) {
        repository.deleteById(id);
    }

    @CacheConfig(cacheNames = "products")
    @Cacheable(key = "#id")
    public Product getProductById(Long id) {
        return repository.findById(id).orElse(null);
    }
}
```

In this example, we add additional caching annotations to the ProductService class. The saveProduct method uses @CachePut to update the cache with the newly saved product. The deleteProductById method uses @CacheEvict to remove the product with the specified ID from the cache. The getProductById method uses @CacheConfig to specify the cache name and @Cacheable to cache individual products by ID.

By using caching in this way, we can significantly improve the performance of our Spring Boot application by reducing the time it takes to retrieve frequently accessed data.

24. Can we use yaml and properties files simutaneously in Spring Boot?

Yes, it is possible to use YAML and properties files simultaneously in a Spring Boot application.

Spring Boot supports multiple external configuration file formats, including YAML and properties files. By default, Spring Boot looks for a file named application.yml or application.properties in the classpath, and it loads the configuration from the file.

If you have both a application.yml and a application.properties file in the classpath, Spring Boot will load both files, but the properties in the application.yml file will take precedence over the properties in the application.properties file.

You can also specify additional configuration files using the spring.config.name and spring.config.location properties in your application's application.properties or application.yml file.

For example, you can have the following files in your classpath:

```
- application.yml
- application-dev.properties
```

The application.yml file will contain the default configuration for your application, while the application-dev.properties file will contain the configuration specific to the "dev" profile. When you start your application with the dev profile, Spring Boot will load both files and apply the configurations accordingly.

So, in summary, you can use both YAML and properties files in Spring Boot simultaneously, and you can even use them together to provide different configurations for different profiles or environments.

25. What to do if we dont want to use tomcat server in spring boot and instead what are the options and how to define it?

Spring Boot is designed to be flexible, so you can choose to use a different server instead of the default Tomcat server. Here are the steps to use a different server in Spring Boot:

Add the dependency for the server that you want to use in your pom.xml file or build.gradle file. For example, if you want to use Jetty server, you can add the following dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Note: The exact configuration may vary depending on the server you choose.

In your application class, remove the @EnableAutoConfiguration annotation and replace it with

`@SpringBootApplication(exclude = {EmbeddedServletContainerAutoConfiguration.class}).`

Add a `@Bean` method to your application class to configure the embedded server. For example, if you want to use Jetty server, you can add the following method:

```
@Bean
public JettyServletWebServerFactory jettyServletWebServerFactory() {
    JettyServletWebServerFactory factory = new JettyServletWebServerFactory();
    // configure the factory
    return factory;
}
```

Note: The exact configuration may vary depending on the server you choose.

Run your application as usual.

That's it! Spring Boot will now use the server that you specified instead of the default Tomcat server.

In summary, to use a different server in Spring Boot, you need to add the dependency for the server, remove the `@EnableAutoConfiguration` annotation, add a `@Bean` method to configure the server, and run your application.

26. When do we use `@Primary`?

The `@Primary` annotation is used in Spring to indicate that a particular bean should be given preference when there are multiple beans of the same type.

When there are multiple beans of the same type in the application context, Spring needs to know which bean to use. This can be done by using the `@Qualifier` annotation to specify the name of the bean, but it can become cumbersome when there are many beans of the same type. In this case, the `@Primary` annotation can be used to provide a default bean that should be used when there are multiple beans of the same type.

Here's an example to illustrate how `@Primary` can be used. Suppose you have two beans that implement the same interface:

```
public interface Animal {
   public String makeSound();
}

@Component
public class Dog implements Animal {
   @Override
   public String makeSound() {
      return "Woof";
   }
}

@Component
@Primary
public class Cat implements Animal {
   @Override
   public String makeSound() {
      return "Meow";
   }
}
```

In this example, we have two beans, Dog and Cat, that both implement the Animal interface. The `@Primary` annotation is used on the Cat bean to indicate that it should be given preference when there are multiple Animal beans in the application context.

If you use the `@Autowired` annotation to inject an Animal bean into a class, the Cat bean will be used by default, because it has the `@Primary` annotation. If you want to use the Dog bean instead, you can use the `@Qualifier` annotation to specify the name of the bean:

```
@Component
public class AnimalService {
   @Autowired
   public AnimalService(@Qualifier("dog") Animal animal) {
      // ...
   }
}
```
In this example, the `@Qualifier` annotation is used to specify the name of the bean to inject.

So, in summary, you can use the `@Primary` annotation to provide a default bean that should be used when there are multiple beans of the same type.

27. How do we test static method?

Testing a static method can be challenging, especially if the static method interacts with external systems or other static methods. However, there are a few strategies that you can use to test static methods effectively.

- Use PowerMockito: PowerMockito is a testing framework that allows you to mock static methods. With PowerMockito, you can mock the behavior of a static method, so that you can test the code that calls the static method without actually executing the method.

Here's an example of how to use PowerMockito to mock a static method:

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(MyClass.class)
public class MyClassTest {

    @Test
    public void testMyStaticMethod() {
        PowerMockito.mockStatic(MyClass.class);
        when(MyClass.myStaticMethod()).thenReturn("mocked response");

        String result = MyClass.myStaticMethod();

        assertEquals("mocked response", result);
    }
}
```
In this example, the `myStaticMethod()` method in the MyClass class is mocked using PowerMockito. The `when()` method is used to specify the return value of the mocked method. Then, the method is called and the result is verified.

- Refactor the static method: One way to test a static method is to refactor it into an instance method of a class. Then, you can create an instance of the class in your test and call the instance method. This is particularly useful if the static method interacts with external systems, as you can mock the external systems in your test.

Here's an example of how to refactor a static method into an instance method:

```
public class MyClass {
    public static String myStaticMethod() {
        // ...
    }

    public String myInstanceMethod() {
        // ...
    }
}
```
In this example, the `myStaticMethod()` method is refactored into an instance method called `myInstanceMethod()`.

Here's an example of how to test the refactored method:

```
public class MyClassTest {
    @Test
    public void testMyInstanceMethod() {
        MyClass myClass = new MyClass();
        String result = myClass.myInstanceMethod();

        assertEquals("expected result", result);
    }
}
```
In this example, an instance of the MyClass class is created, and the `myInstanceMethod()` method is called.

In summary, testing a static method can be challenging, but you can use frameworks like PowerMockito or refactor the static method into an instance method to make testing easier.

28. What is `@InjectMock`?

`@InjectMocks` is a Mockito annotation that can be used to inject mock or spy objects into a class that you want to test.

In Mockito, you create mock or spy objects using the `@Mock` or `@Spy` annotations. These annotations create instances of the mock or spy objects, but they do not automatically inject them into the class you want to test.

To inject the mock or spy objects into the class you want to test, you can use the `@InjectMocks` annotation. This annotation tells Mockito to inject the mock or spy objects into the fields of the class you want to test.

Here's an example of how to use the `@InjectMocks` annotation:

```
public class MyClassTest {

    @Mock
    private MyDependency myDependency;

    @InjectMocks
    private MyClass myClass;

    @Test
    public void testMyMethod() {
        when(myDependency.doSomething()).thenReturn("mocked response");

        String result = myClass.myMethod();

        assertEquals("expected result", result);
    }
}
```

In this example, the `@Mock` annotation is used to create a mock object of the MyDependency class. The `@InjectMocks` annotation is used to inject the mock object into the myDependency field of the MyClass class.

When the `myMethod()` method is called on the myClass object, it uses the mocked myDependency object, which has been configured with the `when()` method to return a specific value.

The `assertEquals()` method is then used to verify that the myMethod() method returns the expected result.

In summary, the `@InjectMocks` annotation is used to inject mock or spy objects into a class that you want to test, and it can help simplify your test code by automatically injecting the dependencies that you need to test the class.

29. What are doReturn and thenReturn?

In Mockito, `doReturn()` and `thenReturn()` are two different methods that can be used to specify the behavior of a mocked object.

`thenReturn()` is used to specify the return value of a method call on a mock object. Here's an example:

```
List<String> myList = mock(List.class);
when(myList.get(0)).thenReturn("first");

String result = myList.get(0);

assertEquals("first", result);
```

In this example, we're creating a mock object of the List class, and then using the when() method to specify that when the get(0) method is called on the mock object, it should return the string "first". We then call the get(0) method on the mock object and verify that it returns the expected value.

doReturn() is similar to thenReturn(), but it's used to specify the behavior of a void method or a method that returns void. Here's an example:

```
List<String> myList = mock(List.class);
doReturn("first").when(myList).set(0, "new first");

myList.set(0, "new first");

assertEquals("first", myList.get(0));
```

In this example, we're creating a mock object of the List class, and then using the doReturn() method to specify that when the set(0, "new first") method is called on the mock object, it should have no effect and return the string "first". We then call the set(0, "new first") method on the mock object, and then verify that the value of the list at index 0 is still "first".

In summary, thenReturn() is used to specify the return value of a method call on a mock object, while doReturn() is used to specify the behavior of a void method or a method that returns void. Both methods can be used to specify the behavior of mock objects in your tests.

30. What is dependency injection?

Dependency Injection (DI) is a design pattern used in software engineering that allows you to decouple the components of your application by removing the responsibility of instantiating dependencies from the class that needs them. Instead of a class creating its dependencies directly, the dependencies are injected into the class from an external source, such as a dependency injection framework or a factory.

In DI, classes are designed to receive their dependencies as constructor parameters or through setter methods, which are then injected by the dependency injection framework. By using this approach, the class is not tightly coupled to its dependencies, and can be easily tested in isolation by replacing its dependencies with mock objects.

Dependency injection has several benefits, including:

- Decoupling: By removing the responsibility of instantiating dependencies from the class that needs them, you can reduce coupling and make your code more modular and flexible.
- Testability: By injecting mock objects or stubs in place of real dependencies during testing, you can test classes in isolation and improve the reliability of your tests.
- Reusability: By making your classes more modular and flexible, you can increase the reuse of your code and reduce development time.
- Maintainability: By reducing coupling and improving modularity, you can make your code easier to maintain and modify.

There are several popular dependency injection frameworks available for Java, such as Spring and Guice, which automate the process of dependency injection and provide additional features such as lifecycle management and aspect-oriented programming.

31. What is Transaction Propagation in Spring Boot?

Transaction propagation in Spring Boot refers to the way transactions are handled and propagated between different layers of an application. When multiple methods are involved in a single transaction, they need to be executed as a single unit of work to ensure consistency and integrity of the data. The transaction manager in Spring Boot manages the lifecycle of transactions and propagates them from one method to another based on the propagation behavior specified in the code.

Spring Boot supports the following propagation behaviors:

- REQUIRED: If a transaction exists, the current transaction is used. Otherwise, a new transaction is created.
- SUPPORTS: If a transaction exists, the current transaction is used. Otherwise, the method is executed without a transaction.
- MANDATORY: A transaction must exist for the method to execute. If there is no transaction, an exception is thrown.
- REQUIRES_NEW: A new transaction is created for the method, and any existing transaction is suspended.
- NOT_SUPPORTED: The method is executed without a transaction, and any existing transaction is suspended.
- NEVER: The method is executed without a transaction, and an exception is thrown if an existing transaction is active.
- NESTED: A new transaction is created within an existing transaction. If there is no existing transaction, a new transaction is created.

By specifying the appropriate propagation behavior, developers can control how transactions are propagated and managed across different layers of the application, ensuring data consistency and integrity.

Let's say we have two methods, methodA and methodB, both of which interact with the database and need to be executed as part of the same transaction.

Here's an example of how we can use transaction propagation in Spring Boot to ensure that these methods execute within the same transaction:

```
@Service
@Transactional
public class MyService {

    @Autowired
    private MyRepository repository;
    public void methodA() {
        // some database operations
        repository.save(someObject);

        // calling methodB to perform more database operations within the same transaction
        methodB();
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void methodB() {
        // some more database operations
        repository.save(anotherObject);
    }
}
```

In this example, methodA is annotated with @Transactional, indicating that it should be executed within a transaction. When methodA is called, Spring Boot checks whether there is an active transaction already. If there is no active transaction, a new transaction is created, and methodA and any methods it calls are executed within that transaction.

In methodA, we call methodB which is annotated with @Transactional(propagation = Propagation.REQUIRES_NEW). This indicates that methodB should always execute within a new transaction, even if there is an active transaction when it is called. So, a new transaction is created for methodB, and any database operations it performs are executed within that transaction.

By using transaction propagation in this way, we can ensure that both methodA and methodB execute within the same transaction, while also ensuring that methodB always executes within its own separate transaction. This helps to maintain data consistency and integrity, even in complex applications with multiple layers of logic.

32. Is it sufficient to annotate the classes with @Transactional annotation in Spring's declarative transaction implementation?

Annotating classes with the @Transactional annotation in Spring's declarative transaction implementation is not always sufficient to ensure that transactions are properly managed in your application.

While the @Transactional annotation is a powerful tool for managing transactions in Spring, it is important to understand how it works and what its limitations are.

One potential issue is that the @Transactional annotation is only effective when it is applied to public methods of Spring-managed beans. If you have non-public methods or methods in non-Spring-managed beans that require transaction management, you may need to use programmatic transaction management instead.

Another issue is that the @Transactional annotation can have complex behavior depending on the configuration of your application. For example, you need to configure the transaction manager, set the transaction propagation, specify the rollback rules, and set the isolation level to ensure that transactions behave as expected.

Additionally, using the @Transactional annotation does not guarantee that your transactions are properly designed or optimized for performance. It is important to carefully design and test your transactions to ensure that they are efficient, effective, and provide the necessary data consistency and integrity.

In summary, while the @Transactional annotation is a powerful tool for declarative transaction management in Spring, it is important to understand its limitations and to carefully design and test your transactions to ensure that they meet your requirements.

33. What is isolation level at transaction management?

Isolation level is a concept in transaction management that defines the degree of isolation between concurrent transactions in a database system. The isolation level determines how much one transaction is allowed to see the changes made by other concurrent transactions, and how much changes made by the current transaction are visible to other concurrent transactions.

There are four standard isolation levels defined in the ANSI SQL standard, which are supported by most database management systems. These isolation levels are:

- READ UNCOMMITTED: This is the lowest isolation level, where a transaction can see uncommitted changes made by other concurrent transactions. This level provides the least level of data consistency but the highest level of concurrency.

- READ COMMITTED: In this isolation level, a transaction can only see the committed changes made by other concurrent transactions. This level provides a higher level of data consistency than READ UNCOMMITTED, but a lower level of concurrency.
- REPEATABLE READ: In this isolation level, a transaction can see only the data that existed at the start of the transaction, and any changes made by other concurrent transactions are not visible to the current transaction. This level provides a higher level of data consistency than READ COMMITTED but can still result in phantom reads.
- SERIALIZABLE: This is the highest isolation level, where a transaction can see only the data that existed at the start of the transaction and all other concurrent transactions are prevented from accessing the same data until the transaction is completed. This level provides the highest level of data consistency but the lowest level of concurrency.

In summary, isolation level defines the degree of consistency and concurrency in a database system, and it is an important factor to consider when designing transactional systems that require high data consistency and high concurrency. The appropriate isolation level to use depends on the specific requirements of the application and the trade-offs between data consistency and concurrency that are acceptable

34. What is the function used in WebSecurityConfigurerAdapter? How does it work?

The WebSecurityConfigurerAdapter is a class in Spring Security that provides a convenient base class for configuring the security of a web application. It is typically used to configure security settings such as authentication, authorization, and session management.

One of the key functions of WebSecurityConfigurerAdapter is to provide a mechanism for configuring the security of the HTTP endpoints in your application. This is done through the configure(HttpSecurity http) method.

The configure(HttpSecurity http) method is used to specify which HTTP endpoints in your application should be secured and how they should be secured. It allows you to define security rules and access policies for different URLs, HTTP methods, and user roles.

For example, you can use the authorizeRequests() method to define access rules for your endpoints, such as allowing only authenticated users to access certain endpoints. You can also use the formLogin() method to configure form-based authentication, which enables users to log in using a username and password.

Here is an example of a basic WebSecurityConfigurerAdapter class:

```java
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/admin/**").hasRole("ADMIN")
                .antMatchers("/user/**").hasAnyRole("USER", "ADMIN")
                .anyRequest().authenticated()
                .and()
            .formLogin()
                .loginPage("/login")
                .permitAll()
                .and()
            .logout()
                .permitAll();
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .inMemoryAuthentication()
                .withUser("user").password("password").roles("USER")
                .and()
                .withUser("admin").password("password").roles("USER", "ADMIN");
    }
}
```

In this example, the configure(HttpSecurity http) method is used to define the security rules for the application's HTTP endpoints. The configureGlobal(AuthenticationManagerBuilder auth) method is used to configure the authentication mechanism for the application, which in this case is an in-memory user database.

Overall, the WebSecurityConfigurerAdapter class is a powerful and flexible tool for configuring the security of your web application in Spring. It provides a convenient way to define the security policies and rules that are required to protect your application and its users.

35. What is GOLD standards for API?

GOLD (General OpenAPI-Linked Data) is a set of best practices and standards for designing and implementing APIs (Application Programming Interfaces) that facilitate the sharing of data and services across different systems and organizations.

The GOLD standard includes the following principles:

- RESTful architecture: The API should follow the REST (Representational State Transfer) architecture, which is based on the principles of simplicity, scalability, and loose coupling between components. This means that the API should use HTTP verbs (GET, POST, PUT, DELETE) to perform operations on resources identified by URIs (Uniform Resource Identifiers).
- OpenAPI specification: The API should use the OpenAPI (formerly Swagger) specification to define the API's structure, endpoints, parameters, responses, and documentation. The OpenAPI specification provides a standardized format for describing RESTful APIs, which can be used by developers to build clients and automate tests.
- Linked data: The API should use linked data principles to represent and connect data in a machine-readable format that allows for the integration and reuse of data across different applications and domains. Linked data uses unique identifiers (URIs) to identify and link resources, and RDF (Resource Description Framework) to represent the data and relationships between resources.
- Design for reuse: The API should be designed with the goal of facilitating reuse and interoperability across different systems and domains. This means that the API should have a clear and consistent structure, use standard data formats, and provide documentation and examples that make it easy for developers to understand and use the API.
- Discovery and access: The API should provide mechanisms for discovery and access, such as a directory of available APIs, authentication and authorization mechanisms, and rate limiting and throttling policies to ensure fair and efficient access to the API.

Overall, the GOLD standard aims to provide a framework for designing and implementing APIs that promote interoperability, reusability, and accessibility, and that can be easily integrated into existing systems and workflows.

36. What are RESTful Webservice?

REST (Representational State Transfer) is a set of architectural principles for building web services that are scalable, flexible, and platform-independent. RESTful web services are based on these principles and use the HTTP (Hypertext Transfer Protocol) to communicate between the client and the server.

A RESTful web service uses HTTP methods (GET, POST, PUT, DELETE) to interact with resources, which are identified by URIs (Uniform Resource Identifiers). Each resource has a state, which can be represented in different formats, such as JSON (JavaScript Object Notation), XML (Extensible Markup Language), or plain text.

An example of a RESTful web service is the GitHub API, which allows developers to access and manipulate data from GitHub repositories. The GitHub API provides a set of endpoints (URIs) that represent different resources, such as repositories, issues, comments, and users. Developers can use HTTP methods to interact with these resources and retrieve or modify their state.

For example, to retrieve information about a repository, a developer can send an HTTP GET request to the repository endpoint, which has the following format:

```
https://api.github.com/repos/:owner/:repo
```

where :owner and :repo are placeholders for the owner and name of the repository. The response will be a JSON object that contains information about the repository, such as its name, description, URL, and owner.

Similarly, to create a new issue in a repository, a developer can send an HTTP POST request to the issues endpoint, which has the following format:

```
https://api.github.com/repos/:owner/:repo/issues
```

with a JSON object in the request body that contains the details of the issue, such as its title, body, labels, and assignee.

Overall, the GitHub API is a good example of a RESTful web service that follows the principles of simplicity, scalability, and interoperability, and provides a flexible and powerful way to interact with GitHub repositories.

37. How to write ApplicationListenser Interface to track email that comes into inbox and generate unique id for each mail event?

To track email that comes into an inbox and generate a unique ID for each mail event, you can create an implementation of the ApplicationListener interface in Java. Here's an example implementation that uses JavaMail API to connect to an email server, monitor the inbox folder for new emails, and generate a unique ID for each email:

```
import javax.mail.*;
import javax.mail.event.MessageCountAdapter;
import javax.mail.event.MessageCountEvent;
import java.util.UUID;

public class EmailListener implements ApplicationListener<ContextRefreshedEvent> {

    private final String host;
    private final String username;
    private final String password;

    public EmailListener(String host, String username, String password) {
        this.host = host;
        this.username = username;
        this.password = password;
    }

    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        Properties properties = new Properties();
        properties.setProperty("mail.imap.host", host);
        properties.setProperty("mail.imap.port", "993");
        properties.setProperty("mail.imap.ssl.enable", "true");

        try {
            Session session = Session.getInstance(properties);
            Store store = session.getStore("imap");
            store.connect(host, username, password);

            Folder inbox = store.getFolder("INBOX");
            inbox.open(Folder.READ_ONLY);

            inbox.addMessageCountListener(new MessageCountAdapter() {
                @Override
                public void messagesAdded(MessageCountEvent event) {
                    Message[] messages = event.getMessages();
                    for (Message message : messages) {
                        String id = generateUniqueId();
                        System.out.println("New email received with ID: " + id);
                        // do something with the message, such as extract data or save to a database
                    }
                }
            });

            // keep the listener running indefinitely
            while (true) {
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private String generateUniqueId() {
        return UUID.randomUUID().toString();
    }
}
```

In this implementation, the EmailListener class implements the ApplicationListener interface and overrides the onApplicationEvent method to connect to an email server, monitor the inbox folder for new emails, and generate a unique ID for each email event.

The generateUniqueId method uses the java.util.UUID class to create a random UUID string that can be used as the unique ID for each email event.

To use the EmailListener, you can instantiate it with the email server hostname, username, and password, and then register it with your Spring Boot application context:

```
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

    @Bean
    public EmailListener emailListener() {
        return new EmailListener("mail.example.com", "user@example.com", "password");
    }
}
```
In this example, the EmailListener is instantiated with the email server hostname, username, and password, and registered as a Spring bean using the @Bean annotation. When the Spring Boot application starts up, the EmailListener will connect to the email server, start listening for new emails, and generate a unique ID for each email event.

38. How to detect Spring MVC controller and startup Apache Tomcat through Spring Boot? How will we enable the HTTPS Web Services?

To detect Spring MVC controller and startup Apache Tomcat through Spring Boot, we can use the @SpringBootApplication annotation on our main class. This annotation combines the @Configuration, @EnableAutoConfiguration, and @ComponentScan annotations, which are needed to configure and bootstrap the Spring Boot application.

For example:

```
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```
To enable HTTPS web services in Spring Boot, we can do the following:

Generate a self-signed SSL certificate:

```
keytool -genkey -alias myapp -keyalg RSA -keystore keystore.jks -keysize 2048
```

Configure the SSL properties in application.properties:

```
server.port=8443
server.ssl.key-store=classpath:keystore.jks
server.ssl.key-store-password=changeit
server.ssl.key-password=changeit
```

Enable HTTPS by adding the @EnableWebSecurity annotation on a security configuration class:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requiresChannel().anyRequest().requiresSecure();
    }
}
```
With these configurations in place, when we start the Spring Boot application, it will automatically detect the Spring MVC controllers and start up the Apache Tomcat web server. HTTPS web services will be enabled on the specified port with the specified SSL certificate.

39. What annotations does a single spring boot annotation contains?

The @SpringBootApplication annotation in Spring Boot is a meta-annotation that includes several other annotations, which are:

- @Configuration: Indicates that the class contains one or more bean definitions that the Spring container should manage.
- @EnableAutoConfiguration: Tells Spring Boot to automatically configure the application based on its dependencies, classpath and other beans that have been defined.
- @ComponentScan: Scans the package and its sub-packages where the application class is located to find and register other components or beans.

These three annotations work together to enable the Spring Boot features such as auto-configuration, component scanning, and configuration properties.

40. What is Hibernate?

Hibernate is an open-source Object-Relational Mapping (ORM) framework for Java that simplifies the development of database-driven applications by providing a high-level, object-oriented API for interacting with relational databases.

In traditional database-driven applications, developers have to write complex SQL queries and map the results to Java objects, which can be a time-consuming and error-prone task. Hibernate simplifies this process by providing a set of classes and interfaces that allow developers to map Java objects to database tables and perform common database operations, such as inserting, updating, and querying data, using simple object-oriented methods.

Hibernate uses a technique called ORM to map the Java objects to the database tables. With ORM, a developer can define a set of mappings that describe how each Java class and its properties are mapped to database tables and columns. This allows the developer to work with Java objects and let Hibernate take care of the underlying database operations.

Hibernate supports a wide range of database systems, including Oracle, MySQL, PostgreSQL, and Microsoft SQL Server, and provides a high-level, vendor-neutral API that shields the developer from the underlying database details.

Overall, Hibernate is a powerful and flexible ORM framework that simplifies the development of database-driven applications by providing a high-level, object-oriented API for interacting with relational databases.

41. What is JPA? What is the difference between JPA and Hibernate?

JPA stands for Java Persistence API, which is a standard Java framework for ORM (Object-Relational Mapping). JPA provides a set of interfaces and annotations that allows Java developers to map Java objects to relational database tables and perform common database operations, such as inserting, updating, and querying data, using a high-level, object-oriented API.

JPA is a specification that defines the standard Java API for ORM and is implemented by various ORM frameworks, such as Hibernate, EclipseLink, OpenJPA, and TopLink. JPA defines a set of standard interfaces that ORM providers must implement to be compliant with the JPA specification.

JPA is designed to be vendor-neutral, which means that it can work with any relational database that provides a JDBC driver. JPA provides a set of annotations that allow developers to specify the mapping between Java objects and database tables, such as @Entity, @Table, @Column, @Id, and @GeneratedValue.

JPA also provides a query language called JPQL (Java Persistence Query Language), which is similar to SQL but operates on Java objects instead of database tables. JPQL allows developers to query the database using object-oriented criteria, such as object relationships and inheritance, and provides a type-safe and portable way to access the database.

Overall, JPA is a powerful and flexible framework that simplifies the development of database-driven applications by providing a high-level, object-oriented API for interacting with relational databases, and allows developers to focus on the business logic of their applications instead of the low-level database operations.

JPA and Hibernate are both ORM frameworks for Java, and while they share some similarities, there are also some key differences between the two.

- Standardization: JPA is a specification while Hibernate is an implementation of that specification. JPA is a set of interfaces and annotations that define the API for ORM in Java, while Hibernate is a specific ORM implementation that supports the JPA specification.
- Vendor neutrality: JPA is designed to be vendor-neutral, meaning that it can work with any ORM provider that supports the JPA specification. On the other hand, Hibernate is a specific ORM implementation that only works with Hibernate.
- Learning curve: JPA is considered to have a steeper learning curve than Hibernate, as it has a larger number of annotations and interfaces that need to be understood. Hibernate is generally considered to be more approachable and easier to use.
- Feature set: Hibernate is known for providing many features beyond those provided by the JPA specification. For example, Hibernate provides a second-level cache, a Criteria API, and advanced mapping options. JPA, on the other hand, is a more lightweight specification that only includes the core ORM functionality.
- Performance: While both JPA and Hibernate provide good performance, some benchmarks have shown that Hibernate can be faster than JPA in some scenarios. This may be due to Hibernate's more advanced caching features and its ability to optimize SQL queries.
- Community support: Hibernate has a larger and more active community than JPA, and as a result, it may be easier to find help and resources for Hibernate than for JPA.

In summary, JPA is a standard specification for ORM in Java, while Hibernate is a specific implementation of that specification. Hibernate provides more features and is generally easier to use, while JPA is more lightweight and vendor-neutral.

42. What are other JPA implementations?

In addition to Hibernate, there are several other implementations of the Java Persistence API (JPA). Some of the most popular ones include:

- EclipseLink: EclipseLink is a JPA implementation that is developed by the Eclipse Foundation. It is a feature-rich and high-performance implementation that provides support for both JPA and JAXB.
- Apache OpenJPA: Apache OpenJPA is a JPA implementation that is developed by the Apache Software Foundation. It is a robust and mature implementation that provides a high level of performance and scalability.
- DataNucleus: DataNucleus is a JPA implementation that supports both JPA and JDO (Java Data Objects) standards. It is a flexible and extensible implementation that provides support for a wide range of data stores and technologies.
- ObjectDB: ObjectDB is a JPA implementation that is designed for use with object databases. It provides a high level of performance and scalability and is a good choice for applications that need to manage large amounts of complex data.
- TopLink: TopLink is a JPA implementation that is developed by Oracle. It provides a rich set of features and is known for its high performance and scalability.

Overall, these JPA implementations provide a range of features and performance characteristics, and the choice of implementation may depend on the specific requirements of your application.

43. What is the main element in Hibernate and what are different strategies in Hibernate?

The main element in Hibernate is the mapping between Java classes and database tables. Hibernate provides a set of mapping mechanisms that allow developers to define how Java classes are mapped to database tables, and how Java objects are persisted to and retrieved from the database.

Some of the key mapping mechanisms provided by Hibernate include:

- Annotations: Hibernate provides a set of annotations that can be used to annotate Java classes and properties, and define the mapping between them and database tables and columns.
- XML configuration: Hibernate also provides an XML configuration mechanism that can be used to define the mapping between Java classes and database tables.
- Programmatic configuration: Hibernate allows developers to define the mapping between Java classes and database tables programmatically, using the Hibernate API.

In addition to the mapping mechanisms, Hibernate provides a set of strategies for how data is persisted to and retrieved from the database. These strategies are used to define how Hibernate should manage the persistence and retrieval of objects, and how transactions should be handled.

Some of the key strategies provided by Hibernate include:

- Identity: The identity strategy generates a unique identifier for each persistent object, using an underlying database sequence, identity column, or other mechanism.
- Sequence: The sequence strategy uses an underlying database sequence to generate unique identifiers for each persistent object.
- Table: The table strategy uses a database table to generate unique identifiers for each persistent object.
- Auto: The auto strategy allows Hibernate to choose the appropriate strategy based on the underlying database.
- Increment: The increment strategy generates identifiers by incrementing a counter in the database.
- UUID: The UUID strategy uses a universally unique identifier to generate identifiers for each persistent object.

Overall, Hibernate provides a flexible and powerful set of mapping mechanisms and persistence strategies, allowing developers to define the exact behavior they need for their application.

44. What is the difference between Spring and Spring Boot?

Spring and Spring Boot are two related but distinct frameworks in the Java ecosystem.

Spring is a popular framework for building enterprise applications in Java. It provides a wide range of features and modules, such as inversion of control (IoC), aspect-oriented programming (AOP), data access, web development, security, and more. Spring is highly configurable and extensible, and allows developers to build complex and scalable applications with relative ease.

Spring Boot, on the other hand, is a framework built on top of Spring that aims to simplify and streamline the process of building and deploying Spring-based applications. Spring Boot provides a number of opinionated defaults and auto-configurations that can help developers get started quickly and reduce boilerplate code. It also includes a number of production-ready features, such as health checks, metrics, and logging, that can make it easier to monitor and manage deployed applications.

Here are some of the key differences between Spring and Spring Boot:

- Configuration: Spring requires a lot of manual configuration to get started, such as defining beans, wiring dependencies, and configuring application properties. Spring Boot, by contrast, provides a number of auto-configurations that can automatically set up common features and dependencies based on sensible defaults.
- Convention over configuration: Spring Boot relies heavily on convention over configuration, which means that it makes a lot of assumptions about how an application should be configured and behaves. This can simplify development by reducing the amount of code that needs to be written, but it can also limit flexibility in some cases.
- Opinionated: Spring Boot is often described as an opinionated framework, which means that it makes decisions for you and provides sensible defaults that are designed to work well in most scenarios. This can be a big advantage for developers who want to get started quickly and don't need a lot of customization.
- Production-ready features: Spring Boot includes a number of production-ready features out of the box, such as health checks, metrics, and logging, that can make it easier to monitor and manage deployed applications. Spring provides many of these features as well, but they require more manual configuration.

In summary, while Spring and Spring Boot are related frameworks, they have different focuses and approaches. Spring is a powerful and flexible framework that can be used to build complex enterprise applications, while Spring Boot is an opinionated framework that aims to simplify and streamline the process of building and deploying Spring-based applications.

45. What is Tomcat? How is it configured and used?

Apache Tomcat is an open-source web server and servlet container that is widely used in Java web development. In Spring Boot, Tomcat is the default embedded web server and servlet container. This means that you can run and deploy your Spring Boot application without having to install and configure a separate web server.

To use Tomcat in Spring Boot, you simply need to include the "spring-boot-starter-web" dependency in your project. This will bring in all the necessary libraries and configurations to run Tomcat as the embedded web server.

By default, Spring Boot will configure Tomcat to run on port 8080. You can change this port by setting the "server.port" property in your application.properties or application.yml file.

If you need to configure Tomcat further, you can do so by creating a bean of type "TomcatServletWebServerFactory" in your configuration class, and setting its properties as needed. For example, you can configure the maximum number of connections, set SSL properties, or add additional servlets or filters.

Spring Boot also provides a number of features that make it easy to deploy your application to a Tomcat server. For example, you can create an executable WAR file that includes the embedded Tomcat server, which can be deployed to an external Tomcat server. You can also create a "TomcatEmbeddedServletContainerFactory" bean that can be used to configure a Tomcat server that is running outside of Spring Boot.

Overall, Tomcat is a powerful and flexible web server and servlet container that is widely used in Java web development, and it is easy to configure and use in Spring Boot.

46. What is embedded db? When was it introduced? What are it's types and how to configure it?

Spring Boot provides support for several embedded databases, which are lightweight, self-contained databases that run in the same JVM process as the application. This allows for easy development and testing, as well as simpler deployment, as there is no need to set up and maintain a separate database server.

The embedded database support in Spring Boot was introduced in version 1.4, released in August 2016. Since then, it has been improved and expanded with each new release.

There are several types of embedded databases supported by Spring Boot, including:

- H2 Database: A lightweight, in-memory database that supports SQL and JDBC.

- HSQLDB: Another in-memory database that supports SQL and JDBC.
- Derby: A Java-based relational database that supports SQL and JDBC.
- Embedded MySQL: A version of the MySQL database that runs embedded within the application.

To configure an embedded database in a Spring Boot application, you simply need to include the appropriate dependencies in your project's build file (such as Maven or Gradle), and configure the datasource in your application.properties or application.yml file. For example, here is how you would configure an H2 database:

```
# application.properties
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

This configuration sets up an in-memory H2 database with a single testdb database instance, using the default sa user with no password.

Once the embedded database is configured, you can use it just like any other database in your Spring Boot application, by creating entities and repositories and using them to perform database operations. When you run your application, the embedded database will be started automatically and the necessary tables and data will be created as needed.

47. What is H2? How is it configured and used?

H2 is an open-source in-memory database that can be used in Java applications, including those built with Spring Boot. It is a lightweight and fast database that is often used in testing and development environments.

In Spring Boot, you can use H2 by including the "spring-boot-starter-data-jpa" and "h2" dependencies in your project. The "spring-boot-starter-data-jpa" dependency includes the necessary libraries for using JPA with Spring Boot, while the "h2" dependency provides the H2 database engine.

To configure H2, you can add the following properties to your application.properties or application.yml file:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

These properties set the JDBC URL for the H2 database, as well as the driver class name, username, and password. The last two properties enable the H2 web console and set its path.

Once H2 is configured, you can use it in your Spring Boot application just like any other database. You can define JPA entities that map to database tables, and use the JPA repositories to interact with the database.

When your application is running, you can also access the H2 web console by navigating to the URL http://localhost:8080/h2-console (assuming you have not changed the server port). From the web console, you can view the database schema, execute SQL queries, and perform other database management tasks.

Overall, H2 is a convenient and easy-to-use in-memory database that is well-suited for use in Spring Boot applications, especially in testing and development environments.

48. What is the difference between JPA and JDBC?

JDBC (Java Database Connectivity) and JPA (Java Persistence API) are both Java technologies for accessing and manipulating databases. However, they differ in several ways:

- Level of abstraction: JDBC provides a low-level API for interacting with databases, whereas JPA provides a high-level API for object-relational mapping (ORM) between Java objects and database tables. JPA abstracts away much of the low-level JDBC code that is required to interact with a database.
- Object-relational mapping: As mentioned, JPA provides an ORM layer that maps Java objects to database tables, whereas JDBC does not provide any built-in object-relational mapping.
- Portability: JPA provides a database-independent interface, which means that the same code can be used with different databases without modification. JDBC, on the other hand, requires database-specific code to be written.
- Ease of use: JPA provides a simpler and more intuitive API than JDBC, making it easier to work with databases in a Java application.

In summary, JDBC provides a lower-level API for interacting with databases, while JPA provides a higher-level API for object-relational mapping, making it easier to work with databases in a Java application.

49. What is the difference between @RequestMapping and @GetMapping?

In Spring MVC, @RequestMapping is a more generic annotation that can be used to handle different types of HTTP requests, such as GET, POST, PUT, DELETE, etc. The @GetMapping annotation is a specific type of @RequestMapping that is used to handle HTTP GET requests.

The main difference between @RequestMapping and @GetMapping is that @GetMapping is more concise and specific, as it is only used to handle GET requests. Using @GetMapping can make the code more readable and easier to understand, as it clearly indicates that the method is only intended to handle GET requests.

Here is an example:

```
// using @RequestMapping
@RequestMapping(value = "/users", method = RequestMethod.GET)
public List<User> getUsers() {
    // code to retrieve and return list of users
}

// using @GetMapping
@GetMapping("/users")
public List<User> getUsers() {
    // code to retrieve and return list of users
}
```

In the example above, both methods handle GET requests to the /users endpoint, but @GetMapping is more concise and easier to read.

However, it's worth noting that @RequestMapping is more flexible, as it can be used to handle different types of HTTP requests and can accept more parameters, such as headers, media types, and path variables. In some cases, it may be necessary to use @RequestMapping instead of @GetMapping to handle a more complex scenario.

50. What is Spring MVC?

Spring MVC is a popular web framework built on top of the Spring Framework that helps to create web applications in Java. It follows the Model-View-Controller (MVC) design pattern to separate the concerns of data, presentation, and processing.

In the Spring MVC architecture, the Model is responsible for managing the data and the business logic, the View is responsible for rendering the data to the user interface, and the Controller is responsible for receiving the user input and directing the data flow.

The Spring MVC framework provides many features to make it easier to develop web applications, including:

- Request handling: Spring MVC provides a DispatcherServlet that receives incoming HTTP requests and sends them to the appropriate controller based on the URL.
- Data binding: Spring MVC provides data binding between the model and view. It allows the data to be easily converted and validated for display in the view.
- View resolution: Spring MVC supports various view technologies, including JSP, Thymeleaf, and Velocity. The view resolution process is configurable, and you can add new view resolvers to support other technologies.
- Validation: Spring MVC provides a validation framework that can be used to validate form input on the server-side.
- Error handling: Spring MVC provides various mechanisms to handle exceptions and errors that may occur during the request processing.
- Interceptors: Spring MVC allows you to add interceptors that can be used to perform pre- and post-processing tasks for requests.

RESTful web services: Spring MVC supports the creation of RESTful web services through the use of the @RestController annotation.

Overall, Spring MVC is a powerful web framework that provides many features and benefits for creating web applications in Java.
Model-View-Controller (MVC) is an architectural pattern that separates an application into three interconnected components: the model, the view, and the controller.

- Model: This component represents the data and the business logic of the application. It is responsible for storing and retrieving data and performing operations on that data.

- View: This component is responsible for presenting the data to the user. It is the user interface of the application and can be a web page, a mobile app screen, or any other form of visual representation.
- Controller: This component acts as an intermediary between the view and the model. It receives input from the user through the view, processes that input by invoking methods on the model, and updates the view with the results.

The main idea behind the MVC pattern is to separate the concerns of an application into three distinct components, each with a specific responsibility. This separation makes the application easier to maintain, test, and extend. It also allows for more flexible and reusable code, since each component can be modified or replaced without affecting the others.

In the context of a Spring MVC application, the controller is typically implemented as a Spring controller, the view is typically implemented as a JSP or Thymeleaf template, and the model is typically implemented as a Java object that is passed between the controller and the view.

51. What is included in spring boot starter dependency?

Spring Boot starter dependencies are a set of opinionated, pre-configured dependencies that aim to make it easier to get started with Spring Boot applications.

The starter dependencies typically include the following:

- Core Spring Framework dependencies: These are the core dependencies required for Spring Boot, such as the Spring Context, Spring Boot autoconfiguration, and Spring Boot starter test dependencies.
- Spring Boot-specific dependencies: These dependencies provide additional features and functionalities specific to Spring Boot, such as embedded server dependencies, logging dependencies, and properties-based configuration dependencies.
- Third-party dependencies: These are the commonly used libraries that are often used in Spring Boot applications, such as database drivers, security libraries, and template engines.

Here are some examples of commonly used Spring Boot starter dependencies:

- spring-boot-starter-web: This starter dependency includes all the necessary dependencies to build a web application with Spring Boot, such as Spring MVC and embedded Tomcat server.
- spring-boot-starter-data-jpa: This starter dependency includes all the necessary dependencies for using Spring Data JPA, such as Hibernate and Spring Data Commons.
- spring-boot-starter-security: This starter dependency includes all the necessary dependencies for Spring Security, such as Spring Security Core, Spring Security Configuration, and Spring Security Web.

Using Spring Boot starter dependencies helps simplify project configuration, as it provides a consistent set of libraries and configurations that work together out of the box. Developers can easily add the required starter dependencies to their project and start building their application without worrying about the underlying dependencies or configurations.

52. What is the difference between @RestController and @Controller?

In Spring, @Controller is a class-level annotation that marks a class as a Spring MVC controller. The class is capable of handling HTTP requests, processing input, and producing output. The @Controller annotation is typically used to build web applications that provide server-side rendering.

On the other hand, @RestController is a specialized version of the @Controller annotation that combines the functionality of @Controller and @ResponseBody. This means that classes annotated with @RestController are capable of returning the JSON or XML representation of the resources directly, without requiring a separate view resolver.

To summarize, the key difference between @Controller and @RestController is that the former is used for generating web pages that are meant to be displayed in a browser, while the latter is used for generating machine-readable data in the form of JSON or XML that can be consumed by client applications.

53. How to define different environments for application?

In Spring Boot, you can define different environments for your application by using different configuration files or properties. Here's how to do it:

- Create configuration files:

Create a separate configuration file for each environment you want to define. For example, you could create a application.properties file for your default configuration, and then create application-dev.properties and application-prod.properties files for your development and production environments respectively.

- Define active profile:

Set the active profile for your application by adding the following line to your application.properties file:

```
spring.profiles.active=dev
```

This will set the active profile to dev, meaning that Spring Boot will use the application-dev.properties file for configuration.

- Configure different environments:

In each of your configuration files, you can define different properties or override properties from the default configuration file. For example, you could define a different database URL or log level for your development and production environments.

Here's an example of how you could configure a different database URL for your production environment:

```
# application-prod.properties
spring.datasource.url=jdbc:mysql://localhost/mydb_prod
```

- Load different environments:

When you run your Spring Boot application, it will automatically load the configuration file for the active profile. For example, if you set the active profile to prod, Spring Boot will load the application-prod.properties file. You can also specify the active profile when running your application by passing the following command-line argument:

```
java -jar myapp.jar --spring.profiles.active=prod
```

This will set the active profile to prod and load the application-prod.properties file.

By defining different environments in this way, you can easily switch between different configurations for your application depending on the environment it's running in.

54. Define Employee class entity and Department where each department has multiple employees.

Employee.java

```
import javax.persistence.*;

import lombok.*;

@Entity
@Table(name = "employees")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
    private String phone;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "department_id")
    private Department department;
}
```

In this example, the Employee class is annotated with `@Entity` and `@Table` annotations to map the class to a database table named "employees". The `@Data`, `@NoArgsConstructor`, and `@AllArgsConstructor` annotations are from Lombok and provide automatic generation of getters, setters, constructors, and other boilerplate code.

The Employee class also has a Many-to-One relationship with the Department class, which is defined by the `@ManyToOne` and `@JoinColumn` annotations. This allows multiple employees to be associated with a single department.

Department.java

```
import java.util.*;

import javax.persistence.*;

import lombok.*;

@Entity
@Table(name = "departments")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @OneToMany(mappedBy = "department")
    private List<Employee> employees = new ArrayList<>();
}
```

In this example, the Department class is annotated with `@Entity` and `@Table` annotations to map the class to a database table named "departments". The `@Data`, `@NoArgsConstructor`, and `@AllArgsConstructor` annotations are from Lombok and provide automatic generation of getters, setters, constructors, and other boilerplate code.

The Department class has a One-to-Many relationship with the Employee class, which is defined by the `@OneToMany` and `@JoinColumn` annotations. This allows a department to have multiple employees associated with it.

Note that the actual implementation may vary depending on the specific needs of your application and database.

Java Interview Questions:

1. What are different ways to create an object?

In Java, there are several ways to create an object:

- Using the new keyword: This is the most common way of creating an object. You use the new keyword followed by the name of the class and any arguments that the class constructor requires.

Example:

```
MyClass obj = new MyClass();
```

- Using a static factory method: Some classes provide static factory methods to create objects. These methods return an instance of the class.

Example:

```
MyClass obj = MyClass.createInstance();
```

- Using object cloning: Object cloning is the process of creating a new object that is an exact copy of an existing object. This can be done using the clone() method defined in the Object class.

Example:

```
MyClass obj1 = new MyClass(); MyClass obj2 = (MyClass) obj1.clone();
```

- Using deserialization: Deserialization is the process of creating an object from its serialized form. This is done using the ObjectInputStream class.

Example:

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("myobject.ser")); MyClass obj = (MyClass)
in.readObject();
```

- Using reflection: Reflection is a powerful feature in Java that allows you to inspect and manipulate objects at runtime. You can create objects using the newInstance() method provided by the Class class.

Example:

```
MyClass obj = (MyClass) Class.forName("com.example.MyClass").newInstance();
```

2. What are intermediate and terminal functions in streams? Which is one eager and which in lazy loading?

In Java, the java.util.stream package provides a way to perform functional-style operations on streams of data. These operations are divided into two categories: intermediate operations and terminal operations.

- Intermediate operations: Intermediate operations are operations that are performed on a stream and return another stream as the result. Intermediate operations are typically used to transform or filter the elements in a stream. Examples of intermediate operations include map(), filter(), flatMap(), etc.

- Terminal operations: Terminal operations are operations that are performed on a stream and produce a non-stream result, such as a count, a summary statistic, or a single value. Terminal operations are used to produce a final result from a stream. Examples of terminal operations include forEach(), count(), collect(), reduce(), etc.

Intermediate operations are lazy, meaning that they do not actually process the elements in the stream until a terminal operation is called. This allows the intermediate operations to be combined and optimized into a single operation, reducing the number of operations that need to be performed on the data. Terminal operations, on the other hand, cause the elements in the stream to be processed and the final result to be produced.

3. What is ThreadLocal class?

The ThreadLocal class in Java is a utility class that provides thread-local variables. These are variables whose values are specific to the current thread and are not shared among multiple threads.

Each thread that accesses a thread-local variable has its own, independently initialized copy of the variable. This makes it possible for each thread to have its own unique value for the variable, even if the variable is defined in a shared scope.

ThreadLocal is often used to store per-thread information, such as transaction IDs, session IDs, or user IDs. By using a ThreadLocal, you can ensure that each thread has its own private data that is not visible to or affected by other threads. Here's an example of how to use the ThreadLocal class:

```
ThreadLocal<Integer> threadLocal = new ThreadLocal<>();

threadLocal.set(1);
System.out.println(threadLocal.get()); // prints "1"

threadLocal.remove();
```

In this example, the threadLocal variable is created as a ThreadLocal<Integer>, which means it can store an integer value. The set method is used to assign a value to the thread-local variable, and the get method is used to retrieve the value. The remove method is used to remove the value from the thread-local variable.

4. What is mutex?

A mutex (short for mutual exclusion) is a synchronization mechanism that is used to control access to a shared resource.

A mutex is designed to ensure that only one thread at a time can access the resource.
In a multithreaded environment, multiple threads may attempt to access a shared resource simultaneously. To prevent race conditions and other synchronization problems, a mutex is used to enforce exclusive access to the resource. A mutex is essentially a lock that is associated with a shared resource and is used to synchronize access to that resource.

A thread that wants to access the shared resource must first acquire the mutex. Once it has acquired the mutex, it has exclusive access to the resource and can perform its operations. The thread then releases the mutex when it has finished accessing the resource. Other threads that attempt to access the resource will be blocked until the mutex is released, ensuring that only one thread at a time can access the resource.

Mutexes are commonly used in multithreaded programming to synchronize access to shared data structures and to ensure that only one thread at a time can access a critical section of code. In Java, the synchronized keyword is used to implement mutex-like behavior. The synchronized keyword can be used to declare a method or block of code as a critical section, ensuring that only one thread at a time can execute that code.

```
class SharedResource {
    private int value;
    public synchronized int getValue() {
        return value;
    }
    public synchronized void setValue(int value) {
        this.value = value;
    }
}
class MutexExample {
    public static void main(String[] args) {
        SharedResource sharedResource = new SharedResource();
        Thread thread1 = new Thread(() -> {
            sharedResource.setValue(1);
            System.out.println("Thread 1: value = " + sharedResource.getValue());
        });
        Thread thread2 = new Thread(() -> {
            sharedResource.setValue(2);
            System.out.println("Thread 2: value = " + sharedResource.getValue());
        });
        thread1.start();
        thread2.start();
    }
}
```

In this example, the SharedResource class represents a shared resource that is accessed by multiple threads. The value field is protected by a mutex, which is implemented by declaring the getValue and setValue methods as synchronized. This means that only one thread at a time can access these methods, ensuring that the value field is accessed in a thread-safe manner.

The MutexExample class creates two threads that both access the shared resource. The first thread sets the value of the resource to 1, and the second thread sets the value to 2. Because a mutex protects the access to the shared resource, the output of the program is guaranteed to be correct and to avoid race conditions.

The output of the above program is non-deterministic and may vary each time the program is run. However, it will look something like this:

```
Thread 1: value = 1
Thread 2: value = 2
or
Thread 2: value = 2
Thread 1: value = 1
```

The order in which the threads are executed and the values they print is not guaranteed and may change each time the program is run. However, the critical section of code that accesses the shared resource is protected by a mutex, so the values of the resource are guaranteed to be correct and to avoid race conditions.

5. What is fail fast method and fail safe behavior?

"Fail-fast" and "fail-safe" are terms used to describe different behavior in data structures or algorithms when they encounter errors.

"Fail-fast" refers to a method that immediately throws an exception when an error occurs. For example, if a collection (such as an ArrayList or HashMap) is being modified while an iteration is being performed, the iteration may throw a `ConcurrentModificationException` to indicate that the collection has been modified during the iteration. The purpose of this behavior is to alert the programmer of potential data corruption or incorrect results and to prevent further processing.

On the other hand, "fail-safe" refers to a method that doesn't throw an exception when an error occurs, but instead handles the error in a safe way. For example, a fail-safe iterator for a collection may return a default value instead of throwing an exception or may block until the modification is complete and then continue with the iteration. The purpose of this behavior is to allow the program to continue executing without interruption, even in the presence of errors.

In Java, fail-fast collections are typically found in the `java.util` package, while fail-safe collections are typically found in the `java.util.concurrent` package. Some examples of fail-fast and fail-safe data structures in Java:

Fail-fast data structures:

- ArrayList: ArrayList is a resizable array implementation of the List interface. It is a fail-fast data structure, meaning that any structural modification (addition, removal, or modification of elements) made while iterating over the list will cause a `ConcurrentModificationException` to be thrown.

- HashMap: HashMap is a hash-table-based implementation of the Map interface. It is also a fail-fast data structure, meaning that any structural modification made while iterating over the map will cause a `ConcurrentModificationException` to be thrown.

Fail-safe data structures:

- ConcurrentHashMap: ConcurrentHashMap is a hash-table-based implementation of the Map interface, designed for use in multi-threaded environments. It is a fail-safe data structure, meaning that it does not throw an exception when structural modifications are made during iteration. Instead, it uses lock-free algorithms to ensure that all read operations are atomic and consistent.
- CopyOnWriteArrayList: CopyOnWriteArrayList is a thread-safe, fail-safe implementation of the List interface. It is designed for use in multi-threaded environments, where multiple threads may modify the list simultaneously. When a structural modification is made to the list, a new copy of the list is created, ensuring that all read operations are performed on a consistent view of the data.

6. What is CopyOnWriteArrayList and ConcurrentHashMap? Name few of it's operations and when do we use it?

CopyOnWriteArrayList and ConcurrentHashMap are two concurrent data structures in Java that provide thread-safe access to their elements.

CopyOnWriteArrayList is a thread-safe variant of the ArrayList class. It guarantees that all read operations are thread-safe, but write operations are expensive as they involve creating a new copy of the entire underlying array. When an element is added or modified, a new copy of the entire array is created, and the modification is made on the new copy. This means that iterators that were created before the modification still refer to the old copy of the array, ensuring that they remain consistent and do not throw ConcurrentModificationException.

Some common operations in CopyOnWriteArrayList are:

- add(E e): Adds the element e to the end of the list.
- remove(int index): Removes the element at the specified index from the list.
- set(int index, E element): Replaces the element at the specified index with the specified element.

CopyOnWriteArrayList is particularly useful when there are many reads and few writes. It is not recommended to use CopyOnWriteArrayList when the size of the list is very large or when write operations are frequent.

ConcurrentHashMap is a thread-safe implementation of the Map interface. It allows multiple threads to read and write to the map concurrently without the need for external synchronization. It does not block read operations, and write operations are only blocked for the affected segment of the map. ConcurrentHashMap achieves this concurrency by dividing the map into segments and locking each segment separately.

Some common operations in ConcurrentHashMap are:

- put(K key, V value): Inserts the specified key-value mapping into the map.
- get(Object key): Returns the value to which the specified key is mapped, or null if the map contains no mapping for the key.
- remove(Object key): Removes the key-value mapping for the specified key from the map.

ConcurrentHashMap is useful when multiple threads need to read and write to the same map. It is generally faster than using a synchronized Map or Hashtable because it avoids the overhead of locking the entire map for every operation. However, it is not recommended to use ConcurrentHashMap when the number of writes to the map is very high, as this can cause contention and slow down performance.

7. What is an abstract class and abstract methods and what is the difference between abstract class and interface? Also how do we inherit from abstract class and implement interface, how is it different and give use cases for each and code examples?

An abstract class in Java is a class that cannot be instantiated on its own, but is designed to be extended by subclasses. An abstract class can contain both abstract methods (methods without a body) and concrete methods (methods with a body).

An abstract method is a method that is declared in an abstract class, but does not have a body. An abstract method acts as a blueprint for subclasses to follow, as the subclasses must provide a concrete implementation for each abstract method in the superclass.

The difference between an abstract class and an interface in Java is as follows:

- Abstract class: An abstract class can contain both abstract and concrete methods. Abstract methods must be overridden by subclasses, while concrete methods can be used directly. An abstract class can have instance variables and provide a common behavior to its subclasses.
- Interface: An interface only contains abstract methods, and all methods declared in an interface must be overridden by subclasses. An interface cannot have instance variables, but can be used to define a set of methods that must be implemented by classes that implement the interface.
- Multiple inheritance: An abstract class can extend only one class, but can implement multiple interfaces. An interface, on the other hand, can be implemented by multiple classes, providing a form of multiple inheritance in Java.
- Implementation: An abstract class provides a partial implementation for its subclasses, but the subclasses must provide the complete implementation for the abstract methods. An interface does not provide any implementation, and the classes that implement the interface must provide a complete implementation for all the methods declared in the interface.

To inherit from an abstract class, a subclass must extend the abstract class and provide an implementation for all abstract methods declared in the abstract class. To implement an interface, a class must use the implements keyword followed by the interface name and provide an implementation for all methods declared in the interface.

Here are some code examples:

Example of an abstract class in Java:

```
public abstract class Shape {
    private String color;

    public Shape(String color) {
        this.color = color;
    }
    public String getColor() {
        return color;
    }
    public abstract double getArea();
}
```

Example of a subclass that extends the Shape class:

```
public class Circle extends Shape {
    private double radius;

    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }
    public double getRadius() {
        return radius;
    }
    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

Example of an interface in Java:

```
public interface Printable {
    void print();
}
```

Example of a class that implements the Printable interface:

```
public class Document implements Printable {
    private String content;

    public Document(String content) {
        this.content = content;
    }
    public String getContent() {
        return content;
    }
    @Override
    public void print() {
        System.out.println(content);
    }
}
```

In summary, the choice between using an abstract class and an interface depends on the design of the system and the needs of the programmer. Abstract classes are typically used when a base class needs to provide some common functionality and behavior to its subclasses, whereas interfaces are typically used when a group of related classes need to provide a common set of method signatures.

8. Why do we need Abstract class?

Abstract classes are a way of creating a template or a blueprint for a class hierarchy that defines common behavior or properties for a group of related classes. They cannot be instantiated directly but are meant to be extended by other classes that provide their own implementations of the abstract methods defined in the abstract class.

Here are some reasons why we need abstract classes:

- Code Reusability: Abstract classes provide a way to define a common interface and implementation for a group of related classes. This makes it easier to reuse code and avoid duplication.
- Abstraction: Abstract classes allow us to abstract away implementation details and focus on the essential features of a class hierarchy. This makes the code more maintainable and easier to understand.
- Polymorphism: Abstract classes can be used to achieve polymorphism. Because they can be extended by other classes, we can write code that can work with objects of different classes as long as they implement the same abstract methods.
- Encapsulation: Abstract classes can be used to encapsulate implementation details and hide them from clients. This allows us to change the implementation of a class hierarchy without affecting clients that depend on it.
- Frameworks: Abstract classes are commonly used in frameworks and libraries to provide a common interface and implementation for clients to build on. This makes it easier for developers to build applications using the framework or library.

In summary, abstract classes are a powerful tool for creating class hierarchies that define common behavior or properties for a group of related classes. They provide a way to achieve code reusability, abstraction, polymorphism, encapsulation, and are commonly used in frameworks and libraries.

9. Give a brief overview of each of these data structures: LinkedHashMap, IdentityHashMap, WeakHashMap, EnumMap, LinkedHashSet, EnumSet, TreeMap, ConcurrentSkipListMap and ConcurrentSkipListSet

- LinkedHashMap: LinkedHashMap is a Map implementation that maintains the order of its entries. It does this by maintaining a doubly-linked list of its entries in the order they were inserted. This makes it possible to iterate over the entries in the order they were added.

Use case: Maintaining the order of entries in a Map.

Example:

```
LinkedHashMap<String, Integer> map = new LinkedHashMap<>();
map.put("John", 28);
map.put("Mary", 24);
map.put("Tom", 32);

for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

Output:

```
John: 28
Mary: 24
Tom: 32
```

- IdentityHashMap: IdentityHashMap is a Map implementation that uses reference equality (==) instead of object equality (equals()) to compare keys. This means that two objects with the same value but different memory addresses will be treated as distinct keys.

Use case: Storing keys based on object reference equality, rather than object value equality.

Example:

```
IdentityHashMap<String, Integer> map = new IdentityHashMap<>();
String key1 = new String("John");
String key2 = new String("John");
map.put(key1, 28);
map.put(key2, 32);
```

```
System.out.println(map.size()); // Output: 2
```

- WeakHashMap: WeakHashMap is a Map implementation in which keys are weakly referenced. This means that if a key is no longer strongly referenced (i.e., if no other objects have a strong reference to it), it will be garbage collected. This makes WeakHashMap useful for caching or for managing data that is not critical to the application.

Use case: Caching data that is not critical to the application.

Example:

```
WeakHashMap<String, Integer> cache = new WeakHashMap<>();
String key = "cached_data";
cache.put(key, 1234);

System.out.println(cache.get(key)); // Output: 1234

key = null; // Remove the strong reference to the key
System.gc(); // Run the garbage collector

// The entry will be removed from the cache because the key is no longer strongly referenced
System.out.println(cache.get("cached_data")); // Output: null
```

- EnumMap: EnumMap is a specialized Map implementation that uses enum values as keys. Because enum values are unique and fixed at compile time, EnumMap can be more efficient than other Map implementations.

Use case: Storing data based on enum values.

Example:

```
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

EnumMap<Day, String> schedule = new EnumMap<>(Day.class);
schedule.put(Day.MONDAY, "9am - 5pm");
schedule.put(Day.TUESDAY, "9am - 5pm");
schedule.put(Day.WEDNESDAY, "9am - 5pm");
schedule.put(Day.THURSDAY, "9am - 5pm");
schedule.put(Day.FRIDAY, "9am - 5pm");
schedule.put(Day.SATURDAY, "closed");
schedule.put(Day.SUNDAY, "closed");

System.out.println(schedule.get(Day.MONDAY)); // Output: 9am - 5pm
```

- LinkedHashSet: LinkedHashSet is a Set implementation that maintains the order of its elements. It does this by maintaining a doubly-linked list of its elements in the order they were inserted. This makes it possible to iterate over the elements in the order they were added.

Use case: Maintaining the order of elements in a Set.

Example:

```
LinkedHashSet<Integer> set = new LinkedHashSet<>();
set.add(10);
set.add(5);
set.add(20);

for (int i : set) {
    System.out.println(i);
}
```

Output:

```
10
5
20
```

- EnumSet: EnumSet is a specialized Set implementation that uses enum values as its elements. Like EnumMap, EnumSet can be more efficient than other Set implementations because enum values are unique and fixed at compile time.

Use case: Storing data based on enum values.

Example:

```
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

EnumSet<Day> weekdays = EnumSet.of(Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY, Day.THURSDAY, Day.FRIDAY);

System.out.println(weekdays.contains(Day.MONDAY)); // Output: true
```

- TreeMap: TreeMap is a Map implementation that maintains its entries in sorted order based on their keys. This makes it possible to iterate over the entries in sorted order.

Use case: Maintaining entries in sorted order based on their keys.

Example:

```
TreeMap<String, Integer> map = new TreeMap<>();
map.put("John", 28);
map.put("Tom", 32);
map.put("Mary", 24);
```

Output:

```
John: 28
Mary: 24
Tom: 32
```

- ConcurrentSkipListMap: ConcurrentSkipListMap is a concurrent Map implementation that maintains its entries in sorted order. It uses a skip list data structure to achieve this, which allows for efficient concurrent access and modification.

Use case: Maintaining entries in sorted order in a concurrent environment.

Example:

```
ConcurrentSkipListMap<String, Integer> map = new ConcurrentSkipListMap<>();
map.put("John", 28);
map.put("Mary", 24);
map.put("Tom", 32);

for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

Output:

```
John: 28
Mary: 24
Tom: 32
```

- ConcurrentSkipListSet: ConcurrentSkipListSet is a concurrent Set implementation that maintains its elements in sorted order. Like ConcurrentSkipListMap, it uses a skip list data structure to achieve efficient concurrent access and modification.

Use case: Maintaining elements in sorted order in a concurrent environment.

Example:

```
ConcurrentSkipListSet<Integer> set = new ConcurrentSkipListSet<>();
set.add(10);
set.add(5);
set.add(20);

for (int i : set) {
    System.out.println(i);
}
```

Output:

```
5
10
20
```

Each of these data structures has unique characteristics and is useful in different situations. By understanding the strengths and weaknesses of each one, you can choose the right data structure for your specific use case.

10. How to make a class immutable?

An immutable class is a class whose state cannot be changed once it is created. Here are the steps to create an immutable class in Java:

- Make the class final: To prevent subclassing and modification, mark the class final.
- Make the fields private and final: All the fields in the class should be private and final to prevent direct access and modification.
- Don't provide setters: Since the state of an immutable object cannot change, do not provide setters for the fields.
- Provide a constructor that sets all the fields: Provide a constructor that takes all the fields as parameters and sets them in the object.
- Use defensive copying for mutable fields: If the class has any mutable fields, make sure to perform a defensive copy of them in the constructor to prevent any changes to the original object from affecting the immutable object.
- Override methods like toString, equals, and hashCode: Override these methods to make the class complete.

Here is an example of an immutable class

```
final public class ImmutableClass {
    private final int id;
    private final String name;

    public ImmutableClass(int id, String name) {
        this.id = id;
        this.name = new String(name);
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return new String(name);
    }

    @Override
    public String toString() {
        return "ImmutableClass [id=" + id + ", name=" + name + "]";
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        ImmutableClass other = (ImmutableClass) obj;
        if (id != other.id)
            return false;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
}
```

In this example, the toString, equals, and hashCode methods have been added to the ImmutableClass so that it can be properly printed, compared, and used as a key in a hash-based collection, respectively. Additionally, a defensive copy of the name field is made in the constructor, and in the getName method, to ensure that the caller doesn't modify the internal state of the class.

11. Why is String class immutable?

In Java, the String class is immutable, which means that once a String object is created, it cannot be modified. This is because the contents of the String object are stored in a character array, and the array is marked as final. This means that once the String object is created, the contents of the array cannot be changed.

There are several reasons why the String class is immutable:

- Security: Immutable objects are more secure because they cannot be modified by other code. For example, if a String object is used to store a password, making it immutable ensures that the password cannot be changed by malicious code.
- Thread-safety: Immutable objects can be safely shared between threads without the need for synchronization. This is because the contents of the String object cannot be modified, so there is no risk of race conditions or other concurrency issues.
- Performance: Because String objects are immutable, they can be cached and reused by the JVM, which can improve performance by reducing the amount of memory allocation and garbage collection that is needed.
- Design simplicity: Immutable objects are simpler to design and reason about because their state cannot be changed. This makes them easier to use and less error-prone.

For example, consider the following code:

```
String s1 = "Hello";
String s2 = s1.toUpperCase();
```

In this example, calling the toUpperCase() method on the String object s1 does not modify the original String object. Instead, it creates a new String object with the uppercase version of the original String. This is possible because String objects are immutable.

In summary, the immutability of the String class in Java provides several benefits, including increased security, thread-safety, performance, and design simplicity.

12. What is the difference between String and StringBuilder?

In Java, String and StringBuilder are both classes used to represent textual data. However, there are some key differences between the two:

- Immutability: String objects are immutable, which means that once a String object is created, it cannot be modified. In contrast, StringBuilder objects are mutable, which means that their contents can be modified.
- Performance: Because String objects are immutable, every time a String is modified, a new String object is created. This can lead to performance issues if a lot of modifications need to be made to a String. StringBuilder, on the other hand, is designed for efficient string manipulation, as it allows for in-place modifications.
- Thread-safety: String objects are thread-safe, meaning that multiple threads can safely access the same String object at the same time. StringBuilder, however, is not thread-safe.

In summary, String is suitable for representing fixed textual data that won't be modified, while StringBuilder is suitable for representing text that will be frequently modified.

13. What is the difference between StringBuilder and StringBuffer?

StringBuilder and StringBuffer are two classes in Java that are used for manipulating strings. Both classes provide similar functionality, but there are some differences between them.

- Thread-safety: The main difference between StringBuilder and StringBuffer is thread-safety. StringBuffer is thread-safe, which means that multiple threads can access the same instance of a StringBuffer object without interfering with each other. StringBuilder, on the other hand, is not thread-safe, which means that it should only be used in single-threaded environments.
- Performance: Because StringBuffer is thread-safe, it is slower than StringBuilder, which is not thread-safe. This is because StringBuffer has to synchronize access to its internal state to ensure thread-safety, which adds overhead.
- Mutability: Both StringBuilder and StringBuffer are mutable, which means that you can modify their content after they are created. This makes them useful for situations where you need to concatenate strings or modify their content.
- Usage: StringBuilder was introduced in Java 5 as a faster alternative to StringBuffer. It is recommended to use StringBuilder in single-threaded environments, where thread-safety is not a concern. StringBuffer should be used in multi-threaded environments, where thread-safety is important.

In summary, StringBuffer and StringBuilder are both used for manipulating strings in Java, but they differ in their thread-safety and performance characteristics. StringBuffer is thread-safe but slower, while StringBuilder is not thread-safe but faster. StringBuilder is recommended for single-threaded environments, while StringBuffer should be used in multi-threaded environments.

14. How to make method thread safe and how do two threads access the same instance variable?

To make a method thread-safe, you need to ensure that the method can be executed concurrently by multiple threads without causing any issues such as race conditions or deadlocks. There are several ways to achieve this:

- Synchronization: You can use the synchronized keyword to lock the method, so that only one thread can execute it at a time. For example:

```
public synchronized void incrementCounter() {
      counter++;
}
```

- Locks: You can use the java.util.concurrent.locks.Lock interface to explicitly lock and unlock a section of code. For example:

```
private final Lock lock = new ReentrantLock();
public void incrementCounter() {
      lock.lock();
      try {
            counter++;
      } finally {
            lock.unlock();
      }
}
```

- Atomic variables: You can use java.util.concurrent.atomic.AtomicInteger or similar classes to ensure that updates to the shared state are atomic and therefore thread-safe. For example:

```
private final AtomicInteger counter = new AtomicInteger();
public void incrementCounter() {
      counter.incrementAndGet();
}
```

When two threads access the same instance variable, it is possible for them to interfere with each other's updates, leading to incorrect results. To avoid this, you should make sure to synchronize access to the shared state. For example, you can use the synchronized keyword, locks, or atomic variables, as described above.

15. What are different ways to create threads?

In Java, there are several ways to create threads:

- Extending the Thread class: This involves creating a class that extends the Thread class and overriding the run() method.

Example:

```
public class MyThread extends Thread {
    public void run() {
        // thread code here
    }
}

MyThread t = new MyThread();
t.start();
```

- Implementing the Runnable interface: This involves creating a class that implements the Runnable interface and defining the run() method.

Example:

```
public class MyRunnable implements Runnable {
    public void run() {
        // thread code here
    }
}

Thread t = new Thread(new MyRunnable());
t.start();
```

- Using a lambda expression: This involves creating a lambda expression that defines the thread code and passing it to the Thread constructor.

Example:

```
Thread t = new Thread(() -> {
    // thread code here
});
t.start();
```

- Using an anonymous inner class: This involves creating an anonymous inner class that extends the Thread class or implements the Runnable interface and defining the run() method.

Example:

```
Thread t = new Thread(new Runnable() {
    public void run() {
        // thread code here
    }
});
t.start();
```

16. What is the difference between pass by reference and pass by value?

In Java, method arguments are passed by value. This means that when you pass an object to a method, a copy of the reference to the object is created and passed to the method. The method can modify the object's state, but it cannot change the reference to the object itself.

Pass by reference, on the other hand, means that a reference to the actual object is passed to the method, rather than a copy of the reference. This allows the method to change the reference itself, as well as the state of the object.

It's important to note that Java does not support pass by reference, but it is often described as pass by value of reference, which can be misleading.

Here is an example to illustrate the difference:

```
class Example {
    int x;
}

void modify(Example e) {
    e.x = 20;
    e = new Example();
}

public static void main(String[] args) {
    Example example = new Example();
    example.x = 10;
    modify(example);
    System.out.println(example.x); // Outputs: 20
}
```

In this example, modify method takes an Example object as an argument and modifies its x value. Even though e is reassigned to a new object inside the modify method, the change to the original object's x value is still preserved. This shows that Java passes method arguments by value of reference. Java only supports pass-by-value for all arguments, including object references. However, the behavior of object references can sometimes create the illusion of pass-by-reference.

Here is an example of how object references can create the illusion of pass-by-reference:

```
public class PassByReferenceExample {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Hello");
        updateString(sb);
        System.out.println(sb); // Output: Hello World
    }

    public static void updateString(StringBuilder sb) {
        sb.append(" World");
    }
}
```

In this example, a StringBuilder object is created with the value "Hello". This object reference is passed as an argument to the updateString method. Inside the updateString method, the append method is called on the StringBuilder object, which updates the value of the object. When the updateString method returns, the value of the StringBuilder object is printed, and the output is "Hello World".

Although the StringBuilder object is being modified inside the updateString method, it is not being passed by reference. The object reference is still being passed by value, but the behavior of object references allows the method to modify the object's state.

17. What are functional interfaces?

Functional interfaces are interfaces in Java that define exactly one abstract method. They are used as the basis for functional programming in Java 8 and later, and they provide a way to represent a function as an object.

Functional interfaces are identified by the @FunctionalInterface annotation, which is optional but recommended. Here's an example of a functional interface:

```
@FunctionalInterface
public interface Converter<F, T> {
  T convert(F from);
}
```

In this example, the Converter interface defines a single abstract method convert that takes an object of type F and returns an object of type T.

Functional interfaces can be used as the target type for lambda expressions, method references, or constructor references, allowing you to pass behavior as an argument to a method. For example:

```
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted); // 123
```

In this example, the lambda expression `(from) -> Integer.valueOf(from)` implements the Converter interface, and it can be assigned to a variable of type Converter. This allows you to pass behavior as an argument to another method or store it in a data structure for later use.

18. Can a functional interface which is used as lambda have more than two functions?

Yes, a functional interface used as a lambda can have more than two functions, but only if the functional interface itself specifies more than one abstract method. In Java, a functional interface is an interface that has exactly one abstract method.

However, since Java 8, default methods and static methods are allowed in functional interfaces, as long as they do not override the abstract method. This means that a functional interface can have multiple default methods and static methods, in addition to the one abstract method.

If a functional interface has multiple abstract methods, it is no longer a functional interface and cannot be used with lambda expressions. In this case, you would need to create a separate interface for each abstract method or use a regular interface with named classes instead of lambda expressions.

Here's an example of a functional interface with multiple abstract methods:

```
@FunctionalInterface
public interface Calculator {
    int add(int a, int b);

    int subtract(int a, int b);
}
```

This interface has two abstract methods: add and subtract. However, it is not a valid functional interface because it has more than one abstract method.

If you try to use a lambda expression with this interface, you will get a compilation error:

```
Calculator calculator = (a, b) -> a + b;
```

This will result in a compilation error because the interface has more than one abstract method.

To make this interface a valid functional interface, you can use the @FunctionalInterface annotation and declare one of the methods as default:

```
@FunctionalInterface
public interface Calculator {
    int add(int a, int b);

    default int subtract(int a, int b) {
        return a - b;
    }
}
```

Now this interface is a valid functional interface because it has exactly one abstract method (add) and one default method (subtract). You can use a lambda expression with this interface:

```
Calculator calculator = (a, b) -> a + b;
int result = calculator.add(10, 5); // result = 15
result = calculator.subtract(10, 5); // result = 5
```

19. How does stream API work? Give an example of filter, map, flatMap and reduce.

The Stream API in Java is a powerful tool for processing collections of data. It provides a functional and declarative way to manipulate collections, allowing you to express complex data processing operations in a concise and readable manner.

Here are examples of the common operations in the Stream API:

- filter: The filter operation allows you to filter the elements in a stream based on a given condition. For example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
List<Integer> evenNumbers = numbers.stream()
                                   .filter(n -> n % 2 == 0)
                                   .collect(Collectors.toList());
System.out.println(evenNumbers); // [2, 4, 6, 8, 10]
```

- map: The map operation allows you to transform the elements in a stream into a new form. For example:

```
List<String> words = Arrays.asList("hello", "world");
List<String> uppercaseWords = words.stream()
                                  .map(String::toUpperCase)
                                  .collect(Collectors.toList());
System.out.println(uppercaseWords); // [HELLO, WORLD]
```

- flatMap: The flatMap operation allows you to transform each element in a stream into a new stream, and then flatten the result into a single stream. For example:

```
List<List<Integer>> listOfLists = Arrays.asList(Arrays.asList(1, 2, 3), Arrays.asList(4, 5, 6));
List<Integer> flattened = listOfLists.stream()
                                     .flatMap(List::stream)
                                     .collect(Collectors.toList());
System.out.println(flattened); // [1, 2, 3, 4, 5, 6]
```

- reduce: The reduce operation allows you to combine the elements in a stream into a single value. For example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
System.out.println(sum); // 15
```

In this example, the reduce operation takes an initial value of 0 and a binary operator (a, b) -> a + b that adds the elements in the stream. The result is the sum of all the elements in the stream.

These are just a few examples of the many operations available in the Stream API. By combining these operations in various ways, you can create powerful and efficient data processing pipelines in your applications.

20. What is the difference between internal and external iteration?

In Java, internal iteration and external iteration are two approaches to traverse a collection of elements, such as an array or a list.

Internal iteration refers to a scenario where the iteration control is within the collection being iterated. In other words, the collection provides a method to iterate over its elements, and the caller of the method only needs to provide a function to be applied to each element. Examples of internal iteration in Java include the use of the forEach method in the Stream class or the forEachRemaining method in the Iterator interface.

On the other hand, external iteration refers to a scenario where the iteration control is outside of the collection being iterated. In this approach, the caller manually retrieves each element from the collection and processes it as desired. Examples of external iteration in Java include the use of a for loop to iterate over the elements of an array or the use of a while loop with the next method in the Iterator interface to iterate over the elements of a list.

In summary, the main difference between internal and external iteration in Java is that the former relies on the collection to provide the iteration control, while the latter requires the caller to explicitly manage the iteration control.

Here are examples of both internal and external iteration in Java:

Internal Iteration Example:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Using internal iteration with the forEach method of the Stream class
names.stream().forEach(name -> System.out.println(name));
```

In this example, the forEach method is used for internal iteration to iterate over the elements of the names list and print each element to the console. The forEach method takes a lambda expression as an argument, which is applied to each element of the list.

External Iteration Example:

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

// Using external iteration with a for loop
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

In this example, external iteration is used with a for loop to iterate over the elements of the numbers array and print each element to the console. The for loop explicitly manages the iteration control by using the index variable i to access each element of the array.

21. What are Supplier and Consumer functions?

In Java, Supplier and Consumer are functional interfaces that represent functions that take in no input but return a value, and functions that take in an input and return no output, respectively.

Supplier is a functional interface with a single method get() that takes no arguments and returns a value of a specified type. It can be used to represent a factory function that generates a new object of the specified type.

For example:

```
Supplier<String> supplier = () -> "Hello, World!";
String result = supplier.get(); // returns "Hello, World!"
```

Consumer is a functional interface with a single method accept(T t) that takes an argument of type T and returns no output. It can be used to represent a function that performs some operation on the argument of type T.

For example:

```
Consumer<String> consumer = (s) -> System.out.println(s);
consumer.accept("Hello, World!"); // prints "Hello, World!"
```

22. What is Predicate function?

In Java, a Predicate is a functional interface that represents a function that takes in an input parameter of type T and returns a boolean value. It is typically used to filter or test whether a given input satisfies a certain condition. The Predicate interface contains a single abstract method test(T t) that takes in an input parameter of type T and returns a boolean.

Here is an example of using Predicate to filter a list of integers based on a certain condition:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Define a predicate to filter even numbers
```

```
Predicate<Integer> evenPredicate = n -> n % 2 == 0;

// Use the predicate to filter the list of numbers
List<Integer> evenNumbers = numbers.stream()
                                   .filter(evenPredicate)
                                   .collect(Collectors.toList());
```

In the above example, the evenPredicate is used to filter the list of numbers to only include even numbers. The filter() method takes in the predicate as an argument and returns a new stream containing only the elements that satisfy the predicate.

Predicates can be combined using logical operators such as and(), or(), and negate() to create more complex conditions.

23. What is LinkedHashSet and what is the difference between HashMap and HashSet?

- LinkedHashSet: LinkedHashSet is a concrete implementation of the Set interface that uses a hash table and linked list to store its elements. The linked list provides an order to the elements that is maintained as they are inserted, which makes LinkedHashSet well-suited for cases where you need to preserve the order of elements. Like HashSet, LinkedHashSet provides constant-time performance for the basic operations (add, contains, and remove), and it allows the storage of null elements.
- HashMap vs HashSet:
    a. HashMap is a concrete implementation of the Map interface that uses a hash table to store its elements, mapping keys to values. HashMap allows you to store multiple key-value pairs, and it provides constant-time performance for the basic operations (put, get, and remove). HashMap also allows the storage of null keys and values.
    b. HashSet is a concrete implementation of the Set interface that uses a hash table to store its elements. HashSet stores unique elements, and it provides constant-time performance for the basic operations (add, contains, and remove). HashSet also allows the storage of null elements. Element addition in HashSet internally calls put method of HashMap with key as the element and value as PRESENT(Object PRESENT=new Object()).

In summary, HashMap is used to store key-value pairs, while HashSet is used to store unique elements. Additionally, LinkedHashSet is a variant of HashSet that preserves the order of elements, while HashMap and HashSet do not.

24. What are the Java 8 features?

Java 8 is a major release of the Java programming language that was released in March 2014. It introduced several new features and enhancements to the Java platform, including:

- Lambda expressions: This feature allows you to write code in a functional style, using compact, anonymous functions. You can use lambda expressions to write code that's more concise and readable, especially for operations that require simple processing of elements in a collection.
- Stream API: The Stream API is a powerful tool for processing collections of data. It provides a functional and declarative way to manipulate collections, allowing you to express complex data processing operations in a concise and readable manner.
- Functional interfaces: Java 8 introduced the concept of functional interfaces, which are interfaces that define a single abstract method. You can use functional interfaces as the target type of lambda expressions and method references, making it easier to pass behavior as data.
- Default methods: Java 8 added the ability to define default methods in interfaces. This allows you to provide a default implementation of an interface method in an interface, rather than in an abstract class.
- Date and Time API: Java 8 introduced a new Date and Time API, which is designed to be more flexible, readable, and easier to use than the previous date and time APIs.
- Optional class: Java 8 introduced the Optional class, which is a container object used to represent the presence or absence of a value. The Optional class can help you to avoid null pointer exceptions and make your code more readable and safer.
- CompletableFuture class: Java 8 introduced the CompletableFuture class, which is a powerful and versatile class that represents the result of an asynchronous computation. CompletableFuture makes it easy to perform complex, asynchronous operations in a concise and readable manner.

These are some of the main new features and enhancements introduced in Java 8. By using these features, you can write code that's more concise, readable, and easier to maintain, while also taking advantage of the latest advances in the Java platform.

25. What are the Date Time API improvements introduced in Java 8?

Java 8 introduced several improvements to the Date Time API, which is part of the java.time package. Some of the key improvements are:

- New classes: Java 8 introduced several new classes to represent date and time, including LocalDate, LocalTime, LocalDateTime, Instant, ZonedDateTime, OffsetDateTime, and OffsetTime. These classes provide a more flexible and comprehensive way to work with date and time.
- Fluent API: The new Date Time API introduced a fluent API, which allows for more natural and intuitive code. For example, you can use methods such as plusDays, minusMinutes, and withZone to manipulate dates and times.
- Time zones: Java 8 introduced a new time zone database that includes more than 600 time zones, making it easier to work with dates and times across different time zones.
- Immutable classes: The new Date Time API classes are immutable, which means that once you create an instance, you cannot modify its state. This can help avoid many common programming errors related to mutable objects.
- Parsing and formatting: The new Date Time API includes a powerful parsing and formatting engine that can handle a wide range of date and time formats.
- Integration with the Stream API: The new Date Time API integrates seamlessly with the Stream API, making it easy to work with large collections of date and time objects.

Overall, the improvements to the Date Time API in Java 8 provide a more modern and comprehensive way to work with date and time, making it easier to write reliable and maintainable code.

26. What is CompletableFuture?

CompletableFuture is a class in Java that represents a promise to return a result in the future. It is a type of Future that can be explicitly completed with a value or an exception. It supports chaining of multiple asynchronous computations, and provides a wide range of methods for creating and combining asynchronous operations.

CompletableFuture was introduced in Java 8 and is part of the java.util.concurrent package. It is widely used in modern Java applications for implementing non-blocking and reactive programming paradigms.

Using CompletableFuture, you can execute a long-running operation asynchronously and return a future object immediately. You can then perform other operations in the meantime, and later retrieve the result of the operation when it becomes available.

Here's an example of how to create and use a CompletableFuture:

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    // Perform a long-running operation asynchronously
    return "Hello, world!";
});

// Perform other operations while the future is being computed
...

// Retrieve the result of the future
String result = future.get();
System.out.println(result); // "Hello, world!"
```

In this example, supplyAsync() is used to create a CompletableFuture that executes a long-running operation asynchronously. The future is then used to perform other operations in the meantime, and the result is retrieved later using the get() method.

27. What is the difference between Future and CompletableFuture?

Future and CompletableFuture are both classes in Java that represent asynchronous computations, but they have some important differences.

A Future represents a result that may not yet be available, and provides methods for checking if the computation is complete and retrieving the result once it is available. However, it does not provide any way to explicitly complete the computation or to combine multiple futures.

On the other hand, a CompletableFuture is a more powerful implementation of Future that allows you to explicitly complete a computation, combine multiple futures, and perform various other operations. It also provides many methods for handling errors and cancellation of computations.

Some key differences between Future and CompletableFuture are:

- Completion: A Future can only be completed externally, i.e., by the operation that created it. In contrast, a CompletableFuture can be completed explicitly by calling its complete() method.
- Chaining: A CompletableFuture can be chained with other futures using methods like thenApply(), thenCompose(), and thenCombine(), allowing you to create a pipeline of computations. A Future does not have any built-in support for chaining.
- Exception Handling: CompletableFuture provides many methods to handle exceptions and errors, such as exceptionally(), handle(), and whenComplete(). In contrast, Future does not have any built-in exception handling support.
- Composition: CompletableFuture provides methods for combining multiple futures, such as allOf() and anyOf(). These methods can be used to wait for multiple futures to complete before continuing.

Overall, CompletableFuture is a more powerful and flexible implementation of Future that provides many more features for handling asynchronous computations.

Here are some examples of using Future and CompletableFuture:

Example of Future:

```
import java.util.concurrent.*;

public class FutureExample {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Future<String> future = executorService.submit(() -> {
            Thread.sleep(2000);
            return "Hello from Future!";
        });
        System.out.println("Waiting for result...");
        while (!future.isDone()) {
            Thread.sleep(500);
            System.out.println("Still waiting...");
        }
        System.out.println(future.get());
        executorService.shutdown();
    }
}
```

In this example, we create a Future by submitting a task to an ExecutorService. We then wait for the result by repeatedly checking if the future is done and sleeping for a short period of time. Once the future is done, we retrieve the result using its get() method.

Example of CompletableFuture:

```
import java.util.concurrent.CompletableFuture;

public class CompletableFutureExample {
    public static void main(String[] args) throws Exception {
        CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            return "Hello";
        });

        CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            return "World!";
        });

        CompletableFuture<String> combinedFuture = future1.thenCombine(future2, (result1, result2) -> result1
+ " " + result2);

        System.out.println("Waiting for results...");
        System.out.println(combinedFuture.get());
    }
}
```

In this example, we create two CompletableFutures using the supplyAsync() method, which runs a task asynchronously and returns a future. We then use the thenCombine() method to combine the results of the two futures into a single future that contains the concatenated string. Finally, we wait for the result of the combined future using its get() method.

28. What happens when you declare a method static in java?

A static method in Java is a method that is associated with the class, rather than with a specific instance of the class. When you declare a method as static, you can access it directly from the class, without having to create an instance of the class. The following are the main characteristics of static methods in Java:

- Accessibility: Static methods can be accessed from anywhere in your code, without having to create an instance of the class. You can access static methods by using the class name followed by the method name, for example: `ClassName.methodName()`.
- No access to instance variables: Because static methods don't have a reference to an instance of the class, they cannot access instance variables. Instead, they can only access static variables.
- No use of `this` keyword: Static methods cannot use the `this` keyword to refer to the current instance of the class, because they don't have a reference to an instance.
- Class-level methods: Because they are associated with the class, rather than with an instance, static methods are often used to define class-level methods. For example, you might use a static method to implement a utility method that is used by several instances of the class.

In summary, when you declare a method as static, you are making it a class-level method that can be accessed directly from the class, without having to create an instance of the class. Static methods are useful for defining utility methods that don't need to access instance variables or use the this keyword.

29. What happens when we declare a method final in java?

When a method is declared as final in Java, it means that the method cannot be overridden in any subclass.

Here are some key points to keep in mind when a method is declared as final:

- A final method cannot be overridden in a subclass, which means that the subclass cannot provide its own implementation of the method.
- Attempting to override a final method in a subclass will result in a compile-time error.
- A final method can be inherited by a subclass, but the subclass cannot change the implementation of the method.
- A final method can be called from a subclass, just like any other inherited method.
- Declaring a method as final does not prevent the class from being extended. It only prevents the subclass from overriding the final method.

Here is an example to illustrate the use of the final keyword on a method:

```
public class Vehicle {
    public final void startEngine() {
        // implementation for starting engine
    }
}

public class Car extends Vehicle {
    // Attempting to override the final method in the superclass will result in a compile-time error
    // public void startEngine() { // this will result in a compile-time error
    //    // implementation for starting engine in a car
    // }
}
```

In the above example, the startEngine() method in the Vehicle class is marked as final, which means that it cannot be overridden in any subclass such as the Car class. Attempting to override the startEngine() method in the Car class will result in a compile-time error.

30. Give an example of lambda function and what could be possible use cases for using it and not using it?

A lambda function is a concise and anonymous function that can be written in place of a traditional anonymous inner class. In Java 8, lambda functions can be used as a more concise and readable way to define and pass behavior as data.

Here's an example of a lambda function in Java 8:

```
(int x, int y) -> x + y
```

This lambda function takes two int parameters and returns their sum. You can use this lambda function as the target of a method reference or as the argument to a method that expects a functional interface.

Use cases for using lambda functions:

- Simplifying event handling: You can use lambda functions to simplify event handling code in GUI applications. For example, you can use a lambda function as the event handler for a button click, instead of having to write an anonymous inner class.
- Processing collections: The Stream API in Java 8 provides a functional and declarative way to manipulate collections. You can use lambda functions to define operations such as filtering, mapping, and reducing that are applied to elements in a collection.
- Concise and readable code: Lambda functions can make your code more concise and readable by allowing you to express complex operations in a concise and readable manner.

Use cases for not using lambda functions:

- Performance: Because lambda functions are compiled to anonymous inner classes, they may have a performance overhead compared to traditional anonymous inner classes. If performance is a concern, you may want to avoid using lambda functions for performance-critical operations.
- Debugging: Debugging code that uses lambda functions can be more difficult than debugging code that uses traditional anonymous inner classes, because the stack traces for lambda functions may not be as readable. If you need to debug complex operations, you may want to avoid using lambda functions.

In summary, lambda functions are a powerful and flexible feature in Java 8 that can make your code more concise and readable, especially when working with collections and event handling. However, they may have a performance overhead and can make debugging more difficult, so you may want to use them judiciously

31. How to sort HashMap by its keys?

You can sort a HashMap by its keys by using the following steps:

1. Convert the HashMap to a TreeMap: The TreeMap class implements the SortedMap interface, which provides a sorted view of the underlying map. By converting the HashMap to a TreeMap, you can sort its keys in natural order.
2. Use the forEach method to iterate over the sorted TreeMap and access its keys and values:

Here's an example:

```
HashMap<Integer, String> map = new HashMap<>();
map.put(3, "C");
map.put(1, "A");
map.put(2, "B");

// Convert the HashMap to a TreeMap
TreeMap<Integer, String> sortedMap = new TreeMap<>(map);

// Use forEach to iterate over the sorted TreeMap
sortedMap.forEach((key, value) -> System.out.println(key + ": " + value));

// Output:
// 1: A
// 2: B
// 3: C
```

In this example, the HashMap is first converted to a TreeMap, and then the forEach method is used to iterate over the sorted TreeMap and access its keys and values. The keys are sorted in natural order, so the output is sorted by the keys.

32. Why we cannot inherit from two classes?

In Java, a class can only extend one class, and not multiple classes. This is because of the concept of single inheritance in Java, which means that a class can inherit from only one direct superclass. The reason for this design decision is to prevent the complexity and confusion that can arise from multiple inheritance of implementation. When a class inherits from multiple classes, it can lead to ambiguity in cases where the same method or variable is present in both superclasses. This can result in difficult-to-resolve conflicts and make the code more difficult to maintain.

To overcome the limitations of single inheritance, Java provides alternative mechanisms for reusing code, such as interfaces and composition. By using these mechanisms, you can achieve the same level of code reuse and flexibility that you can get from multiple inheritance, without introducing the ambiguity and complexity that can arise from it.

Consider the following example:

```
class A {
    public void foo() {
        System.out.println("foo from A");
    }
}

class B {
    public void foo() {
        System.out.println("foo from B");
    }
}

class C extends A, B {
    // error: java does not support multiple inheritance
}
```

In this example, we have three classes: A, B, and C. Both A and B have a foo method with the same name, but with different implementations. We then attempt to define C as a subclass of both A and B, which is not allowed in Java.

If Java allowed multiple inheritance, it would be unclear which foo method to use when calling foo on an instance of C. This could lead to ambiguities and conflicts that would be difficult to resolve.

To work around this limitation, Java provides the ability to implement multiple interfaces. An interface defines a set of methods that a class can implement, but it does not provide any implementation itself. A class can implement multiple interfaces, which allows it to provide different implementations of the methods defined in each interface.

Here is an example of how to use interfaces to achieve similar functionality as multiple inheritance:

```
interface A {
    void foo();
}

interface B {
    void foo();
}

class C implements A, B {
    public void foo() {
        System.out.println("foo from C");
    }
}
```

In this example, we define A and B as interfaces that both have a foo method with the same signature. We then define C as a class that implements both A and B. We must provide an implementation for the foo method in C, which allows us to specify which implementation of foo to use.

This approach allows us to achieve similar functionality to multiple inheritance, while avoiding the potential conflicts and ambiguities that can arise when inheriting from multiple classes.

33. What are the different types of memory in java?

In Java, there are several types of memory:

- Heap Memory: Heap memory is the runtime data area from which the memory for all objects and classes is allocated. The Java Virtual Machine (JVM) creates a heap when it starts up, and all objects created in the application are stored in the heap.
- Stack Memory: Stack memory is used to store method invocations and local variables. Each time a method is invoked, a new frame is created on the stack to store the method arguments and local variables. When the method returns, the frame is removed from the stack.
- Method Area: The method area is used to store class metadata and class-level data, such as class variables and method bytecode.
- Native Method Stack: The native method stack is used to store native method invocations, which are methods implemented in native code (e.g., C or C++).
- PC Registers: PC (Program Counter) registers are used to store the address of the next instruction to be executed by the JVM.

It's important to note that the JVM uses a garbage collector to manage heap memory and automatically reclaim memory that is no longer in use by the application, so you don't have to worry about freeing memory explicitly. The stack memory, on the other hand, is managed by the JVM and automatically released when a method returns.

34. How does a garbage collector work?

A garbage collector is a mechanism in a programming language runtime that automatically manages the memory used by an application. It automatically frees up memory that is no longer in use by the program, which can help prevent memory leaks and improve performance.

Garbage collectors work by tracking the objects that are created by the program and monitoring their usage. They identify objects that are no longer being used by the program and reclaim their memory.

There are several different algorithms that garbage collectors can use to determine which objects are still being used and which can be freed. One common algorithm is called mark and sweep, which works in two phases:

- Mark phase: In this phase, the garbage collector traverses all the objects in memory, starting from a set of known "roots" (such as global variables or the call stack), and marks each object that is still in use.
- Sweep phase: In this phase, the garbage collector frees the memory of all the unmarked objects, which are no longer in use by the program.

Java has its own garbage collector, which runs periodically in the background to free up memory used by objects that are no longer referenced by the program. The Java Virtual Machine (JVM) is responsible for running the garbage collector, and it uses different algorithms depending on the configuration of the JVM.

An example of how garbage collection works in Java:

```
public class GarbageCollectionExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        for (int i = 0; i < 1000000; i++) {
            list.add("String " + i);
        }

        // The list is no longer needed, so we set it to null
        list = null;

        // Call the garbage collector to free up the memory used by the list
        System.gc();
    }
}
```

In this example, we create a large list of strings and fill it with one million elements. Once we're done with the list, we set it to null to indicate that it's no longer needed. Finally, we call the System.gc() method to suggest that the JVM run the garbage collector to free up the memory used by the list. Note that the System.gc() method doesn't guarantee that the garbage collector will run immediately or even at all, but it's a suggestion to the JVM to run it.

35. What is a parallel stream? Give an example.

A parallel stream is a type of stream in Java that allows multiple stream operations to be performed in parallel across multiple threads. The main idea behind parallel streams is to take advantage of the processing power of modern multi-core processors by dividing a large task into smaller tasks and executing them simultaneously.

Here's an example of how you can use a parallel stream in Java:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Using a sequential stream
int sum = numbers.stream().filter(n -> n % 2 == 0).mapToInt(Integer::intValue).sum();
System.out.println("Sequential sum: " + sum); //30

// Using a parallel stream
int parallelSum = numbers.parallelStream().filter(n -> n % 2 == 0).mapToInt(Integer::intValue).sum();
System.out.println("Parallel sum: " + parallelSum); //30
```

In this example, we create a list of numbers and calculate the sum of all even numbers in the list using both a sequential stream and a parallel stream. The parallel stream will divide the stream into multiple substreams and execute the filter and map operations in parallel, while the sequential stream will execute these operations one after the other.

The advantage of using a parallel stream is that it can make the execution of certain operations faster by utilizing the processing power of multiple cores. However, it's important to keep in mind that parallel streams are not always faster than sequential streams, as the overhead of creating and managing the multiple threads can sometimes offset the performance gains. It's also important to consider the specific requirements of your application, such as the size of the data, the complexity of the operations, and the desired degree of parallelism, when deciding whether to use a parallel stream.

36. What is the difference between map and flatMap?

In Java 8 and above, map and flatMap are two commonly used intermediate operations in Stream API.

map is used to transform each element of a Stream into another element by applying a function to it. The output of the map operation is a new Stream with the transformed elements.

flatMap, on the other hand, is used to flatten a Stream of Streams into a single Stream by first applying a function to each element of the Stream, and then flattening the resulting Streams into a single Stream.

To illustrate the difference, consider the following example:

```
List<List<Integer>> nestedList = Arrays.asList(
        Arrays.asList(1, 2, 3),
        Arrays.asList(4, 5, 6),
        Arrays.asList(7, 8, 9)
);

// Using map to get the square of each number in the nested list
List<List<Integer>> mappedList = nestedList.stream()
        .map(list -> list.stream()
                .map(num -> num * num)
                .collect(Collectors.toList()))
        .collect(Collectors.toList());
// Output: [[1, 4, 9], [16, 25, 36], [49, 64, 81]]

// Using flatMap to get a single list of squares from the nested list
List<Integer> flattenedList = nestedList.stream()
        .flatMap(list -> list.stream()
                .map(num -> num * num))
        .collect(Collectors.toList());
// Output: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In the first example, the map operation returns a new nested list where each element is the square of the corresponding element in the original nested list.

In the second example, the flatMap operation first applies the map function to each nested list to get a Stream of Integer values, and then flattens the resulting Streams into a single Stream of Integer values.

37. Difference between ConcurrentMap and HashTable?

ConcurrentMap and Hashtable are both collections in Java used to store key-value pairs, but they have some differences:

- Synchronization: Hashtable is synchronized, meaning that all of its methods are thread-safe. This means that only one thread can access the Hashtable at a time, which can result in reduced performance. On the other hand, ConcurrentMap provides concurrent access to the map, allowing multiple threads to access it at the same time.
- Null values: Hashtable does not allow null keys or values, while ConcurrentMap allows null values, but not null keys.
- Performance: Hashtable is slower than ConcurrentMap due to its synchronization overhead, while ConcurrentMap is faster as it allows concurrent access to the map.
- Iteration: Hashtable can throw a ConcurrentModificationException if the map is modified while it is being iterated, while ConcurrentMap provides a fail-safe iterator that does not throw this exception.
- Methods: Hashtable has a few methods that are not available in ConcurrentMap, such as elements and keys, while ConcurrentMap provides additional methods, such as putIfAbsent and remove, that are not available in Hashtable.

Overall, if you need a thread-safe map in your application, it is generally recommended to use ConcurrentMap instead of Hashtable as it provides better performance and more features.

38. Which is faster, streams or for loop?

It depends on the specific use case and the size of the data being processed. Generally speaking, a stream can be slower than a for loop for small data sets, because of the overhead of creating and executing a stream. However, for larger data sets, a stream can be faster than a for loop, especially when the operations being performed are CPU-bound, as streams can take advantage of parallelism to process the data faster.

Additionally, if the operations being performed are complex and involve multiple steps, a stream can be more readable and easier to maintain than a for loop, as it provides a functional, declarative way to process data.

So, in short, it's not possible to say which one is always faster, as it depends on the specific use case and the data being processed. It's important to evaluate the performance of both options and choose the one that best fits the requirements of the specific scenario.

39. Between intermediate streams like filter & map and terminal like count & collect which are lazy loading & eager loading?

Intermediate streams (such as filter and map) are lazy loaded in Java 8. This means that they do not perform any operations until a terminal operation (such as count or collect) is executed on the stream. The intermediate operations are executed only when they are needed to produce the result of the terminal operation. This allows for efficient processing of large data sets, as the intermediate operations are not performed on all elements of the stream, but only on the elements that are needed to produce the result.

On the other hand, terminal operations (such as count and collect) are eager loaded, meaning that they perform the operations on the stream as soon as they are called. Terminal operations trigger the execution of the intermediate operations and return the result. Terminal operations consume the stream, making it impossible to use it again.

In summary, intermediate operations are lazy loaded and terminal operations are eager loaded. The lazy loading of intermediate operations allows for efficient processing of large data sets, while the eager loading of terminal operations triggers the execution of the intermediate operations and produces the result.

40. What is lazy loading and eager loading?

In the context of programming, lazy loading and eager loading are two approaches used in loading data from a database or any other source of data.

Lazy loading is a technique where an object is loaded only when it is needed, instead of loading all the objects at once. This means that when a request is made for a particular object, only that object is loaded from the database, and not the entire set of related objects. This approach is commonly used to improve performance and reduce memory consumption.

On the other hand, eager loading is the technique where all the related objects are loaded at once, even if they are not immediately needed. This approach is useful in situations where the related objects are expected to be used soon after the initial object is loaded, and it can also help to reduce the number of database calls.

To summarize, lazy loading is a technique used to defer the loading of an object until it is needed, while eager loading is the technique where all the related objects are loaded at once.

41. What are equals and hashcode and how to sort objects using it?

equals and hashCode are methods that are part of the Object class in Java and can be overridden in custom classes. They are used to compare objects and determine their equality and uniqueness.

The equals method is used to compare two objects for equality. It returns true if two objects are equal, and false otherwise. The default implementation of equals in the Object class compares objects based on reference equality, but it can be overridden in custom classes to define custom equality criteria.

The hashCode method returns a hash code that is used to identify an object. The hash code is used by data structures like HashMap and HashSet to quickly determine if an object is present in the data structure. The hashCode method must be overridden along with the equals method, so that if two objects are equal, they return the same hash code.

Objects can be sorted using equals and hashCode by implementing the Comparable or Comparator interfaces. The Comparable interface requires a class to define its own natural ordering, by implementing the compareTo method. The Comparator interface allows a class to be sorted by an external comparison method, by implementing the compare method.

Here is an example of how to sort a list of custom objects based on their natural ordering defined by the Comparable interface:

```
class Employee implements Comparable<Employee> {
    private String name;
    private int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Employee o) {
        return this.age - o.age;
    }
}

List<Employee> employees = new ArrayList<>();
employees.add(new Employee("John", 25));
employees.add(new Employee("Jane", 30));
employees.add(new Employee("Jim", 20));

Collections.sort(employees); // Jim, John, Jane
```

In this example, the Employee class implements the Comparable interface and overrides the compareTo method to define the natural ordering based on the employee's age. The Collections.sort method can then be used to sort the list of employees based on their age.

42. In the lambda function’s interface implementation is the function specified abstract?

Yes, the single abstract method in a functional interface is specified as abstract. When implementing a functional interface using a lambda expression, you only provide the implementation for the single abstract method, and it is not necessary to declare the method as abstract.

Here is an example of a functional interface and its implementation using a lambda expression:

```
@FunctionalInterface
interface MyFunctionalInterface {
  void run();
}

MyFunctionalInterface runner = () -> System.out.println("Running");
```

In this example, the MyFunctionalInterface is a functional interface with a single abstract method run(). The implementation of the run method is provided as a lambda expression that simply prints "Running".

43. What is the join method in thread class?

The join method in the Thread class allows one thread to wait for the completion of another thread. The calling thread (the current thread) is suspended until the specified thread (the target thread) terminates.

Here's an example of using the join method:

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

public class Main {
  public static void main(String[] args) {
    MyThread myThread = new MyThread();
    myThread.start();
    try {
      myThread.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Main thread finished");
  }
}
```

In this example, the Main class creates an instance of the MyThread class and starts it. The Main thread then calls the join method on the MyThread instance, causing it to wait until the MyThread thread terminates. When the MyThread thread has finished, the Main thread resumes and outputs "Main thread finished".

44. Explain the Java thread lifecycle.

The Java thread lifecycle consists of several stages, each of which has a different state. The states are as follows:

- New: A new thread is in the new state when it has been instantiated, but the start method has not yet been called.
- Runnable: A runnable thread is a thread that is eligible to run but may be waiting for resources such as the CPU.
- Blocked: A blocked thread is a thread that is waiting for a lock or a condition to be satisfied.
- Waiting: A waiting thread is a thread that is waiting for another thread to complete a specific action or for a certain time period to elapse.
- Timed waiting: A timed waiting thread is a thread that is waiting for a specific time period to elapse or for another thread to complete a specific action.
- Terminated: A terminated thread is a thread that has completed its execution or has been stopped by an external force such as an exception.

A thread moves from one state to another based on the operations performed by the thread and the underlying system. For example, when the start method is called on a new thread, it enters the runnable state. When the thread acquires a lock, it enters the blocked state, and when the lock is released, it reenters the runnable state. When the thread has completed its task or has been interrupted, it enters the terminated state.

45. What is a Memory Leak? What are the common causes of it?

A memory leak is a situation where a computer program retains a memory allocation that is no longer needed. This causes the memory to be occupied indefinitely, leading to a depletion of available memory over time and eventually causing the program to crash.

There are several common causes of memory leaks in Java:

- Unclosed resources: Failing to close resources such as streams, sockets, and database connections can result in memory leaks.
- Improper use of Singleton objects: Singleton objects can cause memory leaks if they hold references to other objects that are no longer needed.
- Cached data: Cached data that is no longer needed can result in memory leaks if the cache is not periodically cleaned.
- Listener and Callback Objects: Listener and callback objects can result in memory leaks if they hold references to objects that are no longer needed.
- Threads: Threads that are not properly managed can result in memory leaks. For example, if a thread is started but not stopped, it will continue to consume memory even after its task has been completed.
- Large object heap: The large object heap, which is used to store objects that are too large to fit on the normal heap, can result in memory leaks if objects are not properly cleaned up.

46. What is Object Cloning and how to achieve it in Java?

Object cloning is a process of creating a duplicate object from an existing object. This process creates a new instance of the object with all its fields, methods and properties being an exact copy of the original object.

There are two ways to achieve object cloning in Java:

- Shallow Cloning: Shallow cloning is the process of creating a new instance of an object and copying the values of its fields into the new object. This type of cloning creates a new reference to the same objects that the original object refers to, rather than creating new objects.
- Deep Cloning: Deep cloning is the process of creating a new instance of an object and copying the values of its fields, as well as the values of its referenced objects, into the new object. This type of cloning creates new objects, rather than new references to the same objects.

To achieve object cloning in Java, the class must implement the Cloneable interface and override the clone() method. The clone() method should create a new instance of the object and copy the values of its fields into the new object. In the case of deep cloning, the referenced objects should also be cloned and their values copied into the new object.

47. What is a thread safe block?

A thread-safe block is a section of code that can be safely executed by multiple threads simultaneously without any interference between the threads. In other words, a thread-safe block ensures that the code within it is executed in a way that prevents concurrent access to shared resources from causing unpredictable results or data corruption.

To make a block of code thread-safe, synchronization mechanisms such as the `synchronized` keyword or `lock` objects can be used. When a block of code is marked as synchronized, only one thread can enter and execute the code within the block at a time, preventing other threads from accessing the same resources until the first thread has finished executing.

For example, consider a shared resource that is updated by multiple threads. To prevent the resource from becoming corrupt, the code that updates the resource can be enclosed within a `synchronized` block. This ensures that only one thread can access the resource at a time, and the resource is updated consistently and correctly.

48. What is the difference between wait() and sleep() in Thread class?

wait() and sleep() are two methods in the Java Thread class that allow a thread to temporarily pause its execution. However, they have different purposes and behaviors:

- wait(): The wait() method is used to release the lock on a shared resource and allow another thread to acquire the lock and modify the shared resource. A thread that calls wait() goes into a waiting state and remains in that state until another thread calls notify() or notifyAll() on the same object. Once the waiting thread is notified, it re-acquires the lock and continues its execution. The wait() method must be called from within a synchronized block to ensure that the lock on the shared resource is held by the current thread.
- sleep(): The sleep() method causes the current thread to temporarily pause its execution for a specified period of time. The thread does not release the lock on any shared resources during this time. Once the specified time has elapsed, the thread resumes its execution. The sleep() method does not require any synchronization, and can be called from anywhere in the code.

In summary, wait() is used to temporarily release the lock on a shared resource and allow another thread to modify it, while sleep() is used to temporarily pause the execution of a thread without releasing any locks.

49. What is hashCode function and why do we need it? Which class does this function belong to?

hashCode() is a method defined in the Object class in Java. It provides a numerical representation (hash code) of an object. The hash code is used for various purposes, including:

- Object identity: The hash code of an object can be used to identify the object, especially when the object is used as a key in a HashMap, HashSet, or any other data structure that relies on the hash code for object equality.
- Object comparison: The hash code can be used to quickly compare two objects to see if they are equal. If two objects have the same hash code, it is likely that they are equal, but not necessarily. If two objects have different hash codes, it is guaranteed that they are not equal.
- Object indexing: The hash code can be used to efficiently index objects in a data structure, allowing for faster access times.

To properly implement the hashCode() method, it must satisfy the following rules:

- Consistency: If two objects are equal, their hash codes must be the same. If the hash code of an object changes over time, the object should not be used as a key in a hash-based data structure.
- Uniqueness: The hash code of an object must be unique. If two objects have the same hash code, it is not guaranteed that they are equal.
- Distribution: The hash code should be distributed evenly across the range of possible hash codes to ensure efficient use of the data structure.

By default, the hashCode() method provided by the Object class returns a unique identifier for each object, but this is not useful for most purposes. The hashCode() method should be overridden in each class to provide a useful hash code that reflects the object's state.

50. What is serialization and deserialization in Java?

Serialization and deserialization are mechanisms in Java that allow an object's state to be saved to a storage medium, such as a file or network stream, and later restored.

Serialization is the process of converting an object's state into a byte stream so that it can be persisted or transmitted over a network. The byte stream can be saved to a file or sent over a network, and later used to recreate the object. During serialization, the object's state, including the values of its instance variables, is written to the byte stream.

Deserialization is the reverse process of serialization, where the byte stream is read and used to recreate an object. The deserialized object is a new instance of the original object and has the same state as the original object when it was serialized.

In Java, objects can be serialized by implementing the Serializable interface. To serialize an object, it is passed to an instance of the ObjectOutputStream, which writes the object's state to a stream. To deserialize an object, it is read from the stream by an instance of the ObjectInputStream.

Serialization is a powerful mechanism that allows objects to be saved and restored easily, but it also has some limitations. For example, not all objects can be serialized. Classes that are not serializable or have members that are not serializable will not be saved or restored correctly. Additionally, serialization can have performance overhead, as the process of writing and reading the byte stream can be slow.

Here's a simple example of serialization and deserialization in Java:

```
import java.io.*;

class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        // Serialize the object
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.ser"))) {
            Person person = new Person("John Doe", 30);
            oos.writeObject(person);
            System.out.println("Serialization successful");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialize the object
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("person.ser"))) {
            Person person = (Person) ois.readObject();
            System.out.println("Deserialization successful: " + person);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

In this example, we have a Person class that implements the Serializable interface. The Person class has a name and an age, and implements the toString() method for printing its state.

The SerializationExample class contains the main() method, which performs the serialization and deserialization. First, we create a Person object and write it to a file using an ObjectOutputStream. Next, we read the object from the file using an ObjectInputStream and cast it to a Person object. Finally, we print the state of the deserialized object to show that it has been restored correctly.

51. What is inheritance, polymorphism and abstraction?

Inheritance is a mechanism in Object Oriented Programming (OOP) that allows a new class to inherit properties and behavior from an existing class. The existing class is known as the parent class or superclass, while the new class is known as the child class or subclass.

Polymorphism is the ability of an object to take on multiple forms. In Java, polymorphism is achieved through method overriding and method overloading.

Abstraction is a mechanism that enables the developer to hide the implementation details and show only the necessary information to the user. This is achieved through abstract classes and interfaces in Java. Abstract classes are classes that cannot be instantiated, but can be subclassed, while interfaces are completely abstract and provide only a blueprint for the classes that implement them.

52. What is the difference between runtime and compile time polymorphism?

Polymorphism in Java can be of two types: compile-time polymorphism and runtime polymorphism.

Compile-time polymorphism, also known as method overloading, occurs when multiple methods with the same name but different parameter list exist in the same class. The correct method to be called is determined at compile-time based on the arguments passed to the method.

Runtime polymorphism, also known as method overriding, occurs when a subclass provides a new implementation for a method defined in the parent class. The correct method to be called is determined at runtime based on the actual type of the object, rather than the type of the reference variable.

In summary, compile-time polymorphism is determined during compile-time based on the method signature, while runtime polymorphism is determined during runtime based on the actual type of the object.

53. Why is composition preferred over inheritance? What is the difference?

Composition and inheritance are two design patterns in object-oriented programming. They are used to reuse code and model relationships between objects.

```
public class Car {
    private Engine engine;
    private String make;
    private String model;

    public Car(Engine engine, String make, String model) {
        this.engine = engine;
        this.make = make;
        this.model = model;
    }

    public void start() {
        engine.start();
        System.out.println("The " + make + " " + model + " has started.");
    }
}

public class Engine {
    public void start() {
        System.out.println("The engine has started.");
    }
}
```

In this example, the Car class contains an instance of the Engine class. When we create a new Car object, we pass in an Engine object to the constructor. The start() method of the Car class calls the start() method of the Engine class to start the engine.

Inheritance is a relationship between classes where a subclass inherits all the attributes and methods of its parent class. Inheritance allows creating new classes that are built on existing classes, creating a hierarchy of classes.

```
public class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }

    public void speak() {
        System.out.println(name + " makes a noise.");
    }
}

public class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    public void speak() {
        System.out.println(name + " barks.");
    }
}
```

In this example, the Dog class extends the Animal class to inherit its name property and speak() method. The Dog class overrides the speak() method to make the dog bark instead of making a generic noise. When we create a new Dog object and call its speak() method, it will print "Dog's name barks."

Composition is a relationship between classes where an object of one class is composed of objects of other classes. It allows creating complex objects by combining simpler objects. Composition is achieved by creating instance variables of other objects in the class.

Composition is generally preferred over inheritance because it offers more flexibility and control over the object's behavior. Composition enables a more dynamic and modular way to build objects, as opposed to inheritance where the object's behavior is determined by its parent class. Composition also allows for easier testing and refactoring, as the objects can be easily replaced with other objects.

In summary, inheritance models a relationship between classes that is determined at compile-time, while composition models a relationship between objects that can be determined at runtime.

54. What is Java MetaSpace? What is the difference between MetaSpace and PermGen?

In Java, Metaspace is a part of the JVM memory where class metadata is stored. The metadata includes information about classes, methods, and other JVM-specific details that are needed to execute the code.

The main difference between Metaspace and PermGen (which was used in earlier versions of Java) is the way that they store the metadata. PermGen was a part of the Java Heap, while Metaspace is a native memory space outside of the Java Heap. As a result, Metaspace can be resized dynamically and does not have a fixed maximum size like PermGen. This makes Metaspace more flexible and less prone to OutOfMemoryErrors due to lack of space.

Additionally, Metaspace allows for faster class loading and unloading, as it uses native memory management instead of the garbage collection mechanism used by PermGen. This can result in better performance and reduced pause times for applications that load and unload classes frequently.

55. What is defaut method in interface?

A default method in an interface is a method that is declared with a default implementation in the interface definition itself. This was introduced in Java 8 and is also known as a defender method or virtual extension method.

Default methods allow interfaces to provide a default implementation of a method so that implementing classes don't have to provide their own implementation. This can be useful in situations where you need to add a new method to an interface, but don't want to break all the classes that already implement the interface.

Here's an example of a default method in an interface:

```
public interface Shape {
    double getArea();

    default void printArea() {
        System.out.println("The area of the shape is " + getArea());
    }
}
```

In this example, the Shape interface has a getArea() method that must be implemented by any class that implements the interface. It also has a printArea() method that provides a default implementation. If a class that implements the Shape interface does not provide its own implementation of printArea(), it will use the default implementation provided by the interface.

It is important to note that default methods can only be defined in interfaces and not in classes. Also, if a class inherits the same method signature from multiple interfaces with default implementations, it must explicitly override the method to provide its own implementation.

Default methods are a powerful feature of Java 8 that allow for more flexible and backwards-compatible interfaces. They can be used to add new functionality to existing interfaces without breaking existing code, and can make it easier to maintain and update complex interfaces.

56. What were the features introducted in java 8 for interfaces?

Java 8 introduced several new features for interfaces, making them more powerful and flexible than before. Some of the key features introduced in Java 8 for interfaces include:

- Default methods: Interfaces can now contain methods with a default implementation. Default methods can provide a default implementation for an interface method, which can be overridden by implementing classes.
- Static methods: Interfaces can now contain static methods, which can be called using the interface name rather than an implementing class.
- Functional interfaces: A functional interface is an interface that contains a single abstract method. Functional interfaces can be used with lambda expressions and method references, making it easier to write code in a functional style.
- Method references: Method references provide a way to refer to an existing method by name, rather than defining a new lambda expression.
- Optional: The Optional class provides a way to handle null values more elegantly, reducing the risk of NullPointerExceptions.

These features have made interfaces more versatile and powerful, and have enabled developers to write more concise and expressive code. Default and static methods have made it easier to add new functionality to interfaces without breaking existing code, and functional interfaces and method references have made it easier to write code in a functional style. The Optional class has also helped to improve code quality by reducing the risk of NullPointerExceptions.

57. What is method reference? Give an example.

Method reference is a shorthand syntax for writing lambda expressions. Instead of explicitly defining the input and output parameters for a lambda expression, method reference allows you to reference an existing method and use it as a lambda expression.

Here is an example of method reference in Java:

```
import java.util.Arrays;

public class Example {
    public static void main(String[] args) {
        String[] words = {"Hello", "World", "Java", "Programming"};

        // Using lambda expression
        Arrays.sort(words, (s1, s2) -> s1.compareToIgnoreCase(s2));
        System.out.println(Arrays.toString(words));

        // Using method reference
        Arrays.sort(words, String::compareToIgnoreCase);
        System.out.println(Arrays.toString(words));
    }
}
```

In the above example, the Arrays.sort() method is used to sort an array of Strings. In the first call to Arrays.sort(), a lambda expression is used to define the comparison function for sorting. In the second call to Arrays.sort(), a method reference is used instead of a lambda expression. The String::compareToIgnoreCase method reference refers to the compareToIgnoreCase() method of the String class, which takes another String as an argument and returns an int. This method is used as the comparison function for sorting the array.

58. What is Class Loader and how does it work?

A class loader is a component of the Java Runtime Environment (JRE) that loads class files into memory at runtime. The Java virtual machine (JVM) uses class loaders to locate and load classes on demand, as they are referenced by executing code.

The class loader works in a hierarchical manner, with each class loader responsible for loading classes from a specific source or location. The three types of class loaders in Java are:

- Bootstrap Class Loader: The bootstrap class loader is responsible for loading the core Java API classes, which are located in the Java runtime environment (JRE).
- Extension Class Loader: The extension class loader is responsible for loading classes from the Java extension directories, which are located in the JRE.
- Application Class Loader: The application class loader is responsible for loading classes from the application classpath, which is a list of directories and JAR files specified by the CLASSPATH environment variable.

When a class is requested by an executing program, the JVM first searches the bootstrap class loader for the class. If the class is not found, the extension class loader is searched. If the class is still not found, the application class loader is searched. If the class is not found by any of these class loaders, a ClassNotFoundException is thrown.

The class loader loads the class file into memory and performs a series of verification steps to ensure that the code is safe to execute. Once the class is loaded and verified, it is ready to be executed by the JVM.

The class loader is a critical component of the Java runtime environment, and is responsible for loading classes on demand at runtime. The hierarchical nature of the class loader allows for different sources of classes to be loaded in a secure and controlled manner, ensuring the safety and security of the executing code.

59. What is JVM, JDK, JRE?

JVM, JDK, and JRE are all important components of the Java platform. Here is a brief description of each:

- JVM: The Java Virtual Machine (JVM) is the virtual machine that executes Java bytecode. It is the foundation of the Java platform and is responsible for running Java programs on a variety of hardware and operating system configurations. The JVM includes several components, including the class loader, bytecode verifier, and execution engine.
- JDK: The Java Development Kit (JDK) is a software development kit used to develop Java applications. It includes the Java compiler, which compiles Java source code into bytecode that can be executed on the JVM. It also includes a variety of development tools, such as the debugger and profiler, as well as the JRE.
- JRE: The Java Runtime Environment (JRE) is a runtime environment for executing Java programs. It includes the JVM and a set of standard libraries, and provides a platform-independent environment for running Java programs. The JRE does not include development tools, such as the Java compiler or debugger.

In summary, the JVM is responsible for executing Java bytecode, the JDK is used to develop Java applications, and the JRE is used to run Java programs. These components work together to provide a robust, platform-independent environment for developing and running Java applications.

60. What is transient keyword where do we use it?

The transient keyword is used in Java to indicate that a field should not be serialized when an object is saved or transmitted. When an object is serialized, its state is written to a stream, so that it can be reconstructed later. The transient keyword can be used to exclude certain fields from the serialization process, so that they are not saved or transmitted along with the object's other state.

Here's an example of how the transient keyword might be used:

```
public class MyClass implements Serializable {
  private String name;
  private transient int age;
  // constructor, getters, setters, etc.

  private void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject();
    out.writeInt(age);
  }

  private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    age = in.readInt();
  }
}
```

In this example, the age field is marked as transient, so it will not be serialized when the MyClass object is saved or transmitted. Instead, the age field is handled explicitly in the writeObject() and readObject() methods, which are called when the object is serialized or deserialized. This allows us to customize the serialization process and include only the fields we want to save or transmit.

Overall, the transient keyword is used to exclude fields from the serialization process in Java, and is typically used for fields that contain sensitive or unnecessary information that should not be saved or transmitted along with the object's other state.

61. What is try-with-resource?

try-with-resources is a language feature introduced in Java 7 that provides a convenient and safe way to manage resources that need to be closed after they have been used, such as files, network connections, and database connections.

Traditionally, when working with resources, you would need to use a try-finally block to ensure that the resource is closed properly, even in the event of an exception. For example:

```
InputStream in = new FileInputStream("file.txt");
try {
  // use the input stream
}
finally {
  in.close();
}
```

With try-with-resources, you can simplify this code and ensure that the resource is always closed, without needing to use a finally block. Here's an example of how try-with-resources can be used to read the contents of a file:

```
try (InputStream in = new FileInputStream("file.txt")) {
  // use the input stream
}
```

In this example, the InputStream is opened and assigned to a variable in the try statement. The InputStream will automatically be closed at the end of the try block, whether an exception is thrown or not. This makes the code simpler, shorter, and less error-prone.

The try-with-resources statement can be used with any resource that implements the AutoCloseable interface, which includes most I/O classes in the Java API. When a try-with-resources block is executed, the resources are closed in the reverse order in which they were opened. If any of the resources fail to close, the exceptions are added to a SuppressedException list, which can be retrieved with the getSuppressed() method of the caught exception.

62. What are different types of garbage collectors?

In Java, there are several types of garbage collectors available, each with its own approach to memory management.

- Serial Garbage Collector: This is the default garbage collector that runs on a single thread and is best suited for small applications that do not have high demands on the system.
- Parallel Garbage Collector: This collector is similar to the serial garbage collector, but it runs on multiple threads, making it faster for applications that have larger heap sizes.
- Concurrent Mark Sweep (CMS) Garbage Collector: This collector runs concurrently with the application, so it tries to minimize pauses caused by garbage collection. It's suitable for applications that require low latency and quick response times.

- Garbage-First (G1) Garbage Collector: This collector is designed for large heap sizes and uses multiple parallel threads to reduce pause times. It monitors and prioritizes the garbage collection process to ensure the most frequently accessed objects are kept in memory, while less frequently accessed objects are discarded.
- Z Garbage Collector (ZGC): This collector is designed for applications that require large heap sizes, high scalability, and low latency. It can handle heaps up to 16 terabytes in size and tries to keep pause times under 10 milliseconds.

Each type of garbage collector has its own strengths and weaknesses, and the optimal garbage collector for a particular application depends on factors such as the size of the heap, the type of application, and the desired response time.

63. What is the difference between ClassNotFoundException and NoClassDefFoundError?

Both ClassNotFoundException and NoClassDefFoundError occur when the JVM can not find a requested class on the classpath. Although they look familiar, there are some core differences between these two.

ClassNotFoundException is a checked exception which occurs when an application tries to load a class through its fully-qualified name and can not find its definition on the classpath.

This occurs mainly when trying to load classes using Class.forName(), ClassLoader.loadClass() or ClassLoader.findSystemClass(). Therefore, we need to be extra careful of java.lang.ClassNotFoundException while working with reflection.

For example, let’s try to load the JDBC driver class without adding necessary dependencies which will get us ClassNotFoundException:

```
@Test(expected = ClassNotFoundException.class)
public void givenNoDrivers_whenLoadDriverClass_thenClassNotFoundException()
  throws ClassNotFoundException {
      Class.forName("oracle.jdbc.driver.OracleDriver");
}
```

NoClassDefFoundError is a fatal error. It occurs when JVM can not find the definition of the class while trying to:

Instantiate a class by using the new keyword. Load a class with a method call. The error occurs when a compiler could successfully compile the class, but Java runtime could not locate the class file. It usually happens when there is an exception while executing a static block or initializing static fields of the class, so class initialization fails.

Let's consider a scenario which is one simple way to reproduce the issue. ClassWithInitErrors initialization throws an exception. So, when we try to create an object of ClassWithInitErrors, it throws ExceptionInInitializerError. If we try to load the same class again, we get the NoClassDefFoundError:

```
public class ClassWithInitErrors {
    static int data = 1 / 0;
}

public class NoClassDefFoundErrorExample {
    public ClassWithInitErrors getClassWithInitErrors() {
        ClassWithInitErrors test;
        try {
            test = new ClassWithInitErrors();
        } catch (Throwable t) {
            System.out.println(t);
        }
        test = new ClassWithInitErrors();
        return test;
    }
}
```

Let us write a test case for this scenario:

```
@Test(expected = NoClassDefFoundError.class)
public void givenInitErrorInClass_whenloadClass_thenNoClassDefFoundError() {

    NoClassDefFoundErrorExample sample
     = new NoClassDefFoundErrorExample();
    sample.getClassWithInitErrors();
}
```

Sometimes, it can be quite time-consuming to diagnose and fix these two problems. The main reason for both problems is the unavailability of the class file (in the classpath) at runtime.

Let's take a look at few approaches we can consider when dealing with either of these:

We need to make sure whether class or jar containing that class is available in the classpath. If not, we need to add it

If it's available on application's classpath then most probably classpath is getting overridden. To fix that we need to find the exact classpath used by our application

Also, if an application is using multiple class loaders, classes loaded by one classloader may not be available by other class loaders. To troubleshoot it well, it's essential to know how classloaders work in Java

While both of these exceptions are related to classpath and Java runtime unable to find a class at run time, it's important to note their differences.

Java runtime throws ClassNotFoundException while trying to load a class at runtime only and the name was provided during runtime. In the case of NoClassDefFoundError, the class was present at compile time, but Java runtime could not find it in Java classpath during runtime.

64. Where do we use volatile keyword?

In Java, the volatile keyword is used to indicate that a variable's value may be modified by multiple threads. When a variable is marked as volatile, the Java Virtual Machine (JVM) ensures that all threads see the latest value of the variable. Specifically, the volatile keyword has the following effects:

- Visibility: When a thread reads the value of a volatile variable, the JVM guarantees that the thread sees the latest value of the variable. In other words, changes made by one thread to the volatile variable are immediately visible to all other threads that access the same variable.
- Atomicity: Reading and writing to a volatile variable is atomic. This means that the operation completes in a single step, without the possibility of interference from other threads.

The most common use of the volatile keyword is to ensure that a flag variable, which is used to signal between threads, is visible to all threads. For example, consider a program with two threads: one thread sets a flag when a computation is complete, and the other thread checks the flag to see if the computation is done. If the flag variable is not marked as volatile, the second thread may not see the updated value of the flag and may continue to wait indefinitely.

Here's an example of using volatile to ensure visibility:

```
public class Flag {
    private volatile boolean done = false;

    public void setDone() {
        done = true;
    }

    public boolean isDone() {
        return done;
    }
}
```

In this example, the done variable is marked as volatile to ensure that any updates to the variable are immediately visible to all threads that access it. The setDone() method sets the done variable to true, and the isDone() method returns the value of the done variable.

65. What is ConcurrentHashMap and ConcurrentHashTable? Can we add null values to any of it? How does HashMap work?

ConcurrentHashMap and ConcurrentHashTable are both thread-safe versions of the HashMap and HashTable data structures, respectively.

ConcurrentHashMap is a high-performance concurrent hash table implementation that allows multiple threads to read and write concurrently without the need for explicit synchronization. It provides better concurrency and scalability compared to HashTable and synchronized HashMap, and also supports atomic operations like putIfAbsent and remove.

ConcurrentHashTable is an older thread-safe version of HashMap that uses synchronization to ensure thread safety. However, it is generally slower and less efficient than ConcurrentHashMap and is now considered to be a legacy class.

Both ConcurrentHashMap and ConcurrentHashTable allow null values, but not null keys. If a null key is added, a NullPointerException will be thrown.

In HashMap, the elements are stored in an array, and the hash code of the key is used to determine the index of the array where the element should be stored. When two keys have the same hash code, they are stored in a linked list at the corresponding array index. Retrieving an element involves computing its hash code, finding the corresponding array index, and traversing the linked list (if necessary) to find the desired element.

Note that the HashMap class is not thread-safe and is not recommended for use in multi-threaded environments. If you need a thread-safe version of HashMap, you can use ConcurrentHashMap.

66. When do we use ArrayList and when do we use LinkedList?

In Java, ArrayList and LinkedList are two commonly used implementations of the List interface, and they have different characteristics that make them more suitable for certain use cases.

ArrayList is implemented using an array, and it is good for scenarios where you need to do a lot of random access to elements in the list. Because the underlying data structure is an array, you can access elements by their index in constant time. ArrayList is also more memory efficient than LinkedList when you need to store a large number of elements, because it does not require as much overhead for each element.

LinkedList, on the other hand, is implemented using a doubly-linked list, and it is good for scenarios where you need to do a lot of insertion and deletion of elements in the list. Because a linked list is made up of nodes that contain references to the previous and next nodes, adding or removing an element is an O(1) operation, whereas it can be O(n) for an ArrayList. However, accessing elements by index in a LinkedList is an O(n) operation, because you have to traverse the list from the beginning or end to find the element.

In summary, you should use an ArrayList when you need to do a lot of random access to elements in the list, and a LinkedList when you need to do a lot of insertion or deletion of elements.

67. What happens when we do not override hashcode of a class in java? How does it compares two objects then?

When we don't override the hashCode() method of a class in Java, the default implementation from the Object class is used. The default implementation of hashCode() returns a unique integer for each object instance based on the memory address of the object. This means that each object will get a different hash code, even if they have the same values for their fields.

If we use such objects as keys in hash-based data structures like HashMap, we may face problems. When we put an object into a HashMap, it uses the object's hashCode() method to determine its bucket in the hash table, and then uses the equals() method to compare keys within that bucket. If two keys have the same hash code, but are not equal according to equals(), then the HashMap cannot distinguish between them, and we may get unexpected results.

To avoid such problems, we should always override the hashCode() method along with the equals() method when creating custom classes that will be used as keys in hash-based data structures. The hashCode() method should generate the same hash code for two objects that are equal according to equals(). This ensures that two objects that are equal will be put into the same bucket in the hash table, and can be retrieved using the same key.

68. Difference between stream and collection?

In Java, a collection is a group of objects, such as a List, Set, or Map. Collections provide methods to add, remove, and manipulate elements within the group. A stream is a sequence of data elements that can be processed in a declarative way. In other words, it is a way of processing a collection of objects using functional programming techniques.

One key difference between streams and collections is that collections are in-memory data structures, while streams are not. Collections store objects in memory and allow for direct access to elements by index. Streams, on the other hand, process elements on-demand and do not store elements in memory.

Another key difference is the way in which elements are processed. Collections typically use iteration to access elements, while streams use a pipeline of operations to process elements. Streams provide a set of operations that can be combined to process elements in a declarative way. These operations can include filtering, sorting, mapping, and reducing.

Overall, the choice between using a stream or a collection will depend on the specific use case and requirements. If you need to store elements in memory and directly access them by index, a collection like ArrayList or LinkedList may be the better choice. If you need to process a large number of elements in a declarative way without storing them in memory, a stream may be the better choice.

69. What is a callable Interface?

The Callable interface is a part of the java.util.concurrent package that defines a single method call() which is similar to the run() method in the Runnable interface. However, the Callable interface is different from Runnable in that it returns a result and can throw an exception.

The call() method returns a generic value of type V which is the type of the result that the Callable implementation produces. The call() method can throw an exception of type Exception or a subclass of Exception.

A Callable task can be submitted to an ExecutorService and can be used to perform tasks in a separate thread. When the call() method is called, the thread waits until the task is completed and returns the result.

Here is an example of implementing the Callable interface:

```
import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {

    @Override
    public String call() throws Exception {
        // Perform some task
        return "Result of the task";
    }
}
```

This implementation of Callable performs some task and returns the result as a string. This Callable can then be submitted to an ExecutorService for execution.

70. What is a Runnable Interface?

The Runnable interface in Java is used to define a single method called run(), which is intended to contain the code that the thread will execute. It is a functional interface that represents a task to be run on a thread.

In other words, Runnable interface provides a way to define a task that can be executed by a thread. It is commonly used to create threads in Java by implementing the Runnable interface and passing an instance of the implementation to the Thread constructor.

Here is an example of implementing Runnable interface:

```
public class MyRunnable implements Runnable {

    @Override
    public void run() {
        // Code to be executed by thread
    }
}

// Creating thread using Runnable implementation
Thread thread = new Thread(new MyRunnable());
thread.start();
```

71. Difference between findFirst() and findAny()?

In Java, both findFirst and findAny are methods of the java.util.stream.Stream interface used for finding elements in a stream. However, there are some differences between the two methods:

Return value: findFirst returns the first element in the stream that matches the given predicate, while findAny returns any element in the stream that matches the given predicate. The exact element returned by findAny is not specified and may vary from run to run.

Performance: Since findFirst needs to find the first element in the stream that matches the given predicate, it may have to traverse the entire stream. On the other hand, findAny can return any element that matches the predicate, which means that it can take advantage of parallel processing and potentially be faster than findFirst in some cases.

Parallel processing: findAny is more suitable for parallel processing than findFirst. When used in a parallel stream, findFirst may have to wait for all the threads to finish before it can return the first element, whereas findAny can return any element that matches the predicate as soon as it finds one.

Use case: findFirst is generally used when you want to find a specific element in a stream, while findAny is used when you just need any element that matches a predicate, and the exact element doesn't matter.

Let's say we have a stream of integers and we want to find the first element that is greater than 10.

```
Stream<Integer> numbers = Stream.of(1, 5, 8, 12, 15, 18, 20);
```

Using findFirst, we can find the first element that matches the predicate as follows:

```
Optional<Integer> result = numbers.filter(n -> n > 10).findFirst();
```

This will return an Optional containing the value 12, since it's the first element in the stream that is greater than 10.

Using findAny, we can find any element that matches the predicate as follows:

```
Optional<Integer> result = numbers.filter(n -> n > 10).findAny();
```

This will also return an Optional containing a value greater than 10, but it's not guaranteed to be the first one in the stream. In fact, it could be any of the values 12, 15, 18, or 20, since they all match the predicate.

72. What is executor service?

ExecutorService is a framework provided by Java to handle and manage threads. It provides a layer of abstraction over low-level thread management tasks, such as creating threads and managing their lifecycle, allowing you to focus on the higher-level task of defining the work that needs to be executed.

With ExecutorService, you can create a pool of threads and submit tasks to be executed in the background. The framework handles the creation, allocation, and reuse of threads in the pool, and the execution of the tasks. This allows for efficient use of resources, as well as better performance and scalability, especially for applications that require the processing of many tasks in parallel.

The ExecutorService interface provides a number of methods for submitting and executing tasks, as well as for managing the thread pool. For example, you can use the submit() method to submit a Callable or Runnable task for execution, or the shutdown() method to shut down the ExecutorService and terminate all the threads in the pool.

Overall, ExecutorService simplifies the task of managing threads in Java, making it easier to build high-performance, concurrent applications.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ExecutorServiceExample {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(5); // create a pool of 5 threads

        // execute 10 tasks in the executor service
        for (int i = 0; i < 10; i++) {
            final int taskId = i;
            executorService.execute(() -> {
                System.out.println("Executing task " + taskId + " in thread " +
Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // simulate some work
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Task " + taskId + " completed in thread " +
Thread.currentThread().getName());
            });
        }

        // shutdown the executor service after all tasks are complete
        executorService.shutdown();
        try {
            if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            executorService.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
}
```

In this example, we create an ExecutorService with a fixed thread pool of 5 threads using the `Executors.newFixedThreadPool()` method. We then submit 10 tasks to the executor service using the `execute()` method. Each task is implemented as a lambda expression that simply prints a message, sleeps for 1 second, and then prints another message.

After all tasks are submitted, we call the `shutdown()` method on the executor service to tell it to stop accepting new tasks. We then use the `awaitTermination()` method to wait for all tasks to complete before shutting down the executor service completely. If the executor service does not complete within 60 seconds, we call the `shutdownNow()` method to force it to stop immediately.

73. What is the difference between Comparator and Comparable?

Both Comparator and Comparable are used for sorting objects in Java, but they differ in their implementation.

Comparable is an interface that is implemented by a class to provide a natural ordering of its objects. The `compareTo()` method is defined in the Comparable interface, and it is used to compare an object with another object of the same class. The compareTo() method returns an integer value, which is negative if the current object is less than the specified object, zero if they are equal, and positive if the current object is greater than the specified object. This natural ordering is used by the Arrays.sort() and Collections.sort() methods.

Comparator, on the other hand, is an interface that provides an external ordering of objects. It allows you to sort objects based on criteria other than their natural ordering. A Comparator is created separately from the class that is being sorted, and it is used to compare two objects of that class. The compare() method is defined in the Comparator interface, and it takes two arguments, the objects to be compared. The compare() method returns an integer value, which is negative if the first object is less than the second object, zero if they are equal, and positive if the first object is greater than the second object.

In summary, Comparable provides a natural ordering of objects and is used for classes where the natural ordering is well-defined, whereas Comparator provides an external ordering of objects and is used when we need to sort objects based on criteria other than their natural ordering.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class Person implements Comparable<Person> {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public int compareTo(Person o) {
        return Integer.compare(this.age, o.age);
    }
}

class PersonComparator implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getName().compareTo(o2.getName());
    }
}

public class ComparatorExample {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("John", 25));
        people.add(new Person("Jane", 30));
        people.add(new Person("Bob", 20));
        // Sort by age using Comparable
        Collections.sort(people);
        System.out.println("Sorted by age:");
        for (Person person : people) {
            System.out.println(person.getName() + " - " + person.getAge());
        }

        // Sort by name using Comparator
        Collections.sort(people, new PersonComparator());
        System.out.println("\nSorted by name:");
        for (Person person : people) {
            System.out.println(person.getName() + " - " + person.getAge());
        }
    }
}
```

In this example, we have a Person class that implements the Comparable interface to compare two Person objects based on their age. We also have a PersonComparator class that implements the Comparator interface to compare two Person objects based on their name.

In the main method, we create a list of Person objects and sort them first by age using the Collections.sort method (which uses the compareTo method defined in the Person class) and then by name using the Collections.sort method with a PersonComparator instance passed in.

74. What is the difference between final, finalize and finally?

final, finalize, and finally are three different concepts in Java, with different meanings and usage. Here's the difference between them:

- final: final is a keyword in Java that can be used to declare a variable, method, or class as constant or unchangeable. Once a variable is declared as final, its value cannot be changed, and once a method or class is declared as final, it cannot be overridden by subclasses.

Example:

```
final int x = 10; // x is a final variable
final class MyClass { } // MyClass is a final class
final void myMethod() { } // myMethod is a final method
```

- finalize: finalize is a method in Java that is called by the garbage collector before an object is garbage collected. The purpose of this method is to allow the object to release any resources it may have acquired during its lifetime.

Example:

```
class MyClass {
  private File file; // a resource that needs to be released

  public MyClass(String fileName) {
    this.file = new File(fileName);
  }

  @Override
  protected void finalize() throws Throwable {
    try {
      // release the file resource
      if (file != null) {
        file.close();
      }
    } finally {
      super.finalize();
    }
  }
}
```

- finally: finally is a block in Java that is used to define a block of code that will be executed regardless of whether an exception is thrown or not. This block is usually used to release resources that were acquired in a try block.

Example:

```
try {
  // some code that may throw an exception
} catch (Exception e) {
  // handle the exception
} finally {
  // release any resources acquired in the try block
}
```

In summary, final is used to declare constants or unchangeable elements, finalize is used to release resources before an object is garbage collected, and finally is used to define a block of code that will be executed regardless of whether an exception is thrown or not.

75. What happens if there is a return statement after throw exception line?

If a return statement is written after a throw statement in a method, the return statement will not be executed because the control flow of the method is transferred to the calling method as soon as the throw statement is executed.

In other words, if an exception is thrown, the method is terminated immediately and the exception is propagated up the call stack to the calling method or handled by an appropriate catch block if one is present. Any code that appears after the throw statement will not be executed.

Therefore, in most cases, it does not make sense to have a return statement after a throw statement in the same method, unless it is inside an appropriate finally block that must be executed regardless of whether an exception is thrown or not.

76. What is the difference between error and exception?

In Java, errors and exceptions are two different types of Throwable objects that represent different problems that can arise during program execution.

An error is a subclass of Throwable that represents problems that are typically outside of the control of the program and usually cannot be handled by the program. Examples of errors include OutOfMemoryError, StackOverflowError, and LinkageError. Errors usually indicate serious problems that prevent the program from continuing to run.

An exception is also a subclass of Throwable that represents problems that can be handled by the program. Examples of exceptions include IOException, SQLException, and NullPointerException. Exceptions can be caught and handled by the program, allowing the program to recover from the problem and continue running.

In summary, errors are typically caused by problems outside of the program's control, while exceptions are caused by problems that can be handled by the program.

77. Time complexity of Collections:

PriorityQueue (Java Doc)

- O(log n) time for the enqueing and dequeing methods (offer, poll, remove() and add)
- O(n) for the remove(Object) and contains(Object) methods
- O(1) for the retrieval methods (peek, element, and size)

ArrayList

- add() – takes O(1) time; however, worst-case scenario, when a new array has to be created and all the elements copied to it, it's O(n)
- add(index, element) – on average runs in O(n) time
- get() – is always a constant time O(1) operation
- remove() – runs in linear O(n) time. We have to iterate the entire array to find the element qualifying for removal.
- indexOf() – also runs in linear time. It iterates through the internal array and checks each element one by one, so the time complexity for this operation always requires O(n) time.
- contains() – implementation is based on indexOf(), so it'll also run in O(n) time.

CopyOnWriteArrayList

- add() – depends on the position we add value, so the complexity is O(n)
- get() – is O(1) constant time operation
- remove() – takes O(n) time
- contains() – likewise, the complexity is O(n)

LinkedList

- add() – appends an element to the end of the list. It only updates a tail, and therefore, it's O(1) constant-time complexity.
- add(index, element) – on average runs in O(n) time
- get() – searching for an element takes O(n) time.
- remove(element) – to remove an element, we first need to find it. This operation is O(n).
- remove(index) – to remove an element by index, we first need to follow the links from the beginning; therefore, the overall complexity is O(n).
- contains() – also has O(n) time complexity

O(1) for HashMap, LinkedHashMap, IdentityHashMap, WeakHashMap, EnumMap and ConcurrentHashMap.

For the tree structure TreeMap and ConcurrentSkipListMap: The put(), get(), remove(), and containsKey() operations time is O(log(n)).

For HashSet, LinkedHashSet, and EnumSet: The add(), remove() and contains() operations cost constant O(1) time thanks to the internal HashMap implementation.

Likewise, the TreeSet has O(log(n)) time complexity

The time complexity for ConcurrentSkipListSet is also O(log(n)) time, as it's based in skip list data structure.

For CopyOnWriteArraySet, the add(), remove() and contains() methods have O(n) average time complexity.

78. What is a Wrapper Class?

In Java, a wrapper class is a class that wraps or contains primitive data types and provides them with a set of methods, making it possible to treat them like objects. In other words, a wrapper class is a class that represents a primitive data type in an object-oriented way, providing additional methods that operate on the primitive data type. For example, Integer is a wrapper class that wraps the int primitive data type, providing a number of additional methods that operate on the int value. Wrapper classes are used to make it possible to use primitive data types in places where objects are required, such as in collections or as method parameters.

Wrapper classes in Java provide a set of methods that allow primitive data types to be used as objects. Some of the common methods provided by the wrapper classes are:

- ValueOf(): This method converts a string representation of a primitive data type to the corresponding wrapper class object.
- Parse: This method converts a string representation of a primitive data type to the corresponding primitive data type.
- CompareTo: This method compares two objects of the same wrapper class type.
- Equals: This method compares two objects of the same wrapper class type for equality.
- ToString: This method returns a string representation of the object.
- hashCode: This method returns a hash code value for the object.
- BooleanValue: This method returns the primitive boolean value of the Boolean object.
- intValue: This method returns the primitive int value of the Integer object.

and so on for each primitive data type.

79. What are Generics?

Generics in Java are a way to parameterize types in a class or method declaration. With generics, you can create classes and methods that work with a variety of data types, rather than just one specific type.

Using generics, you can define a class or method that can accept any data type. For example, the following Box class can hold an object of any type:

```
public class Box<T> {
    private T object;

    public void set(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }
}
```

The T in the angle brackets (<>) is the type parameter, which is a placeholder for a specific type. When you create an instance of the Box class, you can specify a specific type for T. For example:

```
Box<Integer> intBox = new Box<>();
intBox.set(42);
int value = intBox.get(); // value is 42
```

In this example, T is replaced with the Integer type when intBox is created. This means that intBox can only hold integers, and the get method returns an Integer.

Generics are useful for creating reusable code that works with multiple types. They can help to make code more type-safe and reduce the need for explicit type casting.

80. What is externalization?

Externalization is a mechanism in Java that allows an object to customize the way it is written to and read from an object stream. In other words, it provides a way to control the serialization and deserialization process of an object, which is the process of converting an object to a stream of bytes and back.

Unlike the default serialization provided by Java, which automatically serializes all the fields of an object, externalization allows an object to selectively choose which fields to serialize and how to serialize them. This can be useful in situations where the default serialization is not optimal, such as when the serialized form of an object is very large or contains sensitive data that should not be serialized.

To use externalization, an object must implement the java.io.Externalizable interface and provide implementations for the writeExternal and readExternal methods. The writeExternal method is called to write the object's state to an object output stream, and the readExternal method is called to read the object's state from an object input stream.

Here is an example of using externalization in Java:

```
import java.io.*;

class Person implements Externalizable {
    private String name;
    private int age;

    // Default constructor is needed for Externalizable
    public Person() {}

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Override readExternal method to read data from the input stream
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        name = (String) in.readObject();
        age = in.readInt();
    }

    // Override writeExternal method to write data to the output stream
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(name);
        out.writeInt(age);
    }

    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}

public class ExternalizationExample {
    public static void main(String[] args) throws Exception {
        Person person = new Person("John Doe", 30);

        // Write object to file using ObjectOutputStream and FileOutputStream
        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("person.obj"));
        out.writeObject(person);
        out.close();

        // Read object from file using ObjectInputStream and FileInputStream
        ObjectInputStream in = new ObjectInputStream(new FileInputStream("person.obj"));
        Person newPerson = (Person) in.readObject();
        in.close();

        System.out.println("Original object: " + person);
        System.out.println("Deserialized object: " + newPerson);
    }
}
```

In this example, we have created a Person class that implements Externalizable interface. The Person class has two instance variables, name and age. We have overridden the readExternal and writeExternal methods to customize the serialization process. In the main method, we have written a Person object to a file using ObjectOutputStream and read it back using ObjectInputStream. Finally, we have printed the original and deserialized Person objects to verify that the serialization and deserialization was successful.

Backend Interview Questions:

1. What is Redis and how does it work?

Redis is an in-memory data structure store that is used as a database, cache, and message broker. Redis provides a fast and flexible way to store and retrieve data.

Redis stores data in key-value pairs, where the key is a unique identifier for the data and the value is the data itself. Redis supports several data structures such as strings, hashes, lists, sets, and sorted sets, which makes it well-suited for a wide range of use cases.

Redis works by keeping all its data in memory, which provides fast access times for data retrieval and modification. When data is written to Redis, it is also persisted to disk, which ensures that data can be recovered in the event of a system failure. Redis provides a number of commands to perform various operations, such as setting and retrieving values, deleting values, and managing data structures.

Redis provides a number of features that make it a popular choice for use cases such as real-time analytics, leaderboards, session management, and full-text search. These features include high performance, scalability, data persistence, and support for multiple programming languages.

2. What is Kafka and how does it work?

Apache Kafka is a distributed streaming platform for building real-time data pipelines and applications. It works by allowing producers to publish messages to topics and consumers to subscribe to these topics. The messages are stored in a highly-available and highly-replicated cluster, providing durability and fault tolerance.

Here's an example of a simple Java application that uses Apache Kafka:

- The first step is to create a KafkaProducer instance, which is used to send messages to a topic.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "all");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

- Next, we can send messages to a topic using the send() method.

```
producer.send(new ProducerRecord<>("my-topic", "key", "value"));
producer.close();
```

- On the consumer side, we need to create a KafkaConsumer instance and subscribe to the topic.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "my-group");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("my-topic"));
```

- Finally, we can poll for new messages and process them.

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
        System.out.println("Received message: key = " + record.key() + ", value = " + record.value());
    }
}
```

This is a simple example that demonstrates the basic concepts of Apache Kafka. There is much more to learn about Apache Kafka, including more advanced topics such as partitioning, replication, and message durability.

3. What is RabbitMQ and how does it work? Give example with code.

RabbitMQ is an open-source message broker software that enables applications to communicate with each other by sending and receiving messages. It provides a reliable way of transferring messages between different systems, and supports multiple messaging protocols such as AMQP, MQTT, and STOMP.

In RabbitMQ, messages are sent to and received from queues. A producer sends messages to a queue, and a consumer reads messages from a queue. Producers and consumers communicate with RabbitMQ through an exchange, which routes messages to the appropriate queues based on message routing rules.

Here's an example of how to use RabbitMQ in a Spring Boot application:

Add RabbitMQ dependencies to your pom.xml file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Configure RabbitMQ connection in your application.properties file:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

Create a message producer to send messages to RabbitMQ:

```
@Component
public class MessageProducer {

    private final RabbitTemplate rabbitTemplate;
    private final Queue queue;

    @Autowired
    public MessageProducer(RabbitTemplate rabbitTemplate, Queue queue) {
        this.rabbitTemplate = rabbitTemplate;
        this.queue = queue;
    }

    public void send(String message) {
        rabbitTemplate.convertAndSend(queue.getName(), message);
    }
}
```

Create a message consumer to receive messages from RabbitMQ:

```
@Component
public class MessageConsumer {

    private final Queue queue;

    @Autowired
    public MessageConsumer(Queue queue) {
        this.queue = queue;
    }

    @RabbitListener(queues = "${queue.name}")
    public void receive(String message) {
        System.out.println("Received message: " + message);
    }
}
```
Create a Queue bean to hold the name of the RabbitMQ queue:

```
@Bean
public Queue queue() {
    return new Queue("myQueue", false);
}
```

Use the message producer to send a message to RabbitMQ:

```
@Autowired
private MessageProducer messageProducer;
messageProducer.send("Hello, world!");
```

When the message is sent, the message consumer will automatically receive it and print the message “Hello, world!” to the console.

This is just a basic example, but RabbitMQ can be used for more complex messaging scenarios as well.

4. What are replication factor message queues?

Replication factor is a feature in message queues, including RabbitMQ, that allows you to create replicas or copies of a message or a message queue.

In RabbitMQ, the replication factor specifies the number of replicas that should be created for a queue. This ensures that there are multiple copies of a queue on different nodes in a cluster, increasing the reliability and availability of the message queue. If one node fails or goes down, the other replicas can still serve the clients and messages.

For example, if a queue has a replication factor of 3, it means that there will be three replicas of that queue, with one primary and two backup replicas. Any message sent to the primary replica will be copied to the backup replicas, ensuring that the message is not lost in case the primary replica fails.

To set the replication factor in RabbitMQ, you can use the x-ha-policy parameter with the value of all followed by the x-ha-nodes parameter with the list of nodes that should replicate the queue. Here is an example:

```
Map<String, Object> args = new HashMap<>();
args.put("x-ha-policy", "all");
args.put("x-ha-nodes", Arrays.asList("rabbit@node1", "rabbit@node2", "rabbit@node3"));
```

channel.queueDeclare("my_queue", true, false, false, args);
In this example, the queue "my_queue" will have a replication factor of 3, with replicas on nodes "rabbit@node1", "rabbit@node2", and "rabbit@node3".

5. What are protocols AMQP, MQTT, and STOMP and when to use it?

AMQP, MQTT, and STOMP are messaging protocols used in distributed systems to enable communication between different applications and services. Here's a brief overview of each:

- AMQP (Advanced Message Queuing Protocol): It is an open standard application layer protocol for message-oriented middleware that provides messaging services for applications. It is designed to support messaging between applications and services in a reliable, secure, and interoperable way. AMQP provides features like message acknowledgments, message routing, and reliable queuing. It is widely used in enterprise messaging systems, and is supported by many popular message brokers such as RabbitMQ, Apache ActiveMQ, and Apache Qpid.
- MQTT (Message Queue Telemetry Transport): It is a lightweight, publish-subscribe-based messaging protocol that is designed for IoT (Internet of Things) devices with limited processing power and bandwidth. MQTT is used to exchange messages between IoT devices and applications, and is often used in applications like home automation, remote monitoring, and asset tracking.
- STOMP (Simple Text Oriented Messaging Protocol): It is a text-based protocol for messaging that provides an interoperable way for different message brokers to communicate with each other. It is designed to be simple and easy to implement, and provides features like message acknowledgement, message routing, and transaction support. STOMP is often used in web-based messaging applications and is supported by many message brokers like Apache ActiveMQ, RabbitMQ, and Apache Qpid.

When deciding which protocol to use, it is important to consider the specific requirements of your application. MQTT is a good choice for applications with low power devices and limited network bandwidth, while AMQP and STOMP are better suited for enterprise applications with more complex messaging requirements. Additionally, the choice of protocol may also depend on the available messaging broker, as some brokers support multiple protocols while others are optimized for a specific one.

6. What OAuth and how does it work?

OAuth is an open standard protocol that allows a client application to access a user's data or resources on a resource server without having to store or handle the user's credentials directly. It works by establishing a trusted relationship between the resource owner (the user), the resource server, and the client application. OAuth is commonly used in web and mobile applications to grant access to APIs, social media platforms, and other online services.

In Spring Boot, we can use the Spring Security OAuth2 module to implement OAuth2 authentication and authorization. Here's an example of how to configure and use OAuth2 in a Spring Boot application:

Add the necessary dependencies to your project's pom.xml file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
    <version>2.3.7.RELEASE</version>
</dependency>
```

Configure the OAuth2 client in your application.yml or application.properties file:

```
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            clientId: <your-client-id>
            clientSecret: <your-client-secret>
            scope:
              - email
              - profile
        provider:
          google:
            authorizationUri: https://accounts.google.com/o/oauth2/v2/auth
            tokenUri: https://www.googleapis.com/oauth2/v4/token
            userInfoUri: https://www.googleapis.com/oauth2/v3/userinfo
            userNameAttribute: sub
```

Implement a custom UserDetailsService to load the user details from your application's database:

```
@Service
public class CustomUserDetailsService implements UserDetailsService {
    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = userRepository.findByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("User not found with username: " +
username));

        return new org.springframework.security.core.userdetails.User(user.getUsername(), user.getPassword(),
                user.getAuthorities());
    }
}
```

Configure the OAuth2 authentication in your SecurityConfig class:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private CustomUserDetailsService userDetailsService;

    @Autowired
    private DataSource dataSource;

    @Autowired
    public void globalUserDetails(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService)
                .passwordEncoder(passwordEncoder());
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/login").permitAll()
                .anyRequest().authenticated()
                .and()
                .formLogin().permitAll()
                .and()
                .csrf().disable();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public TokenStore tokenStore() {
        return new JdbcTokenStore(dataSource);
    }
}
```

Add an OAuth2 login controller to handle the user's login and redirection to the OAuth2 provider:
typescript

```
@Controller
public class LoginController {
    @Autowired
    private OAuth2AuthorizedClientService authorizedClientService;

    @GetMapping("/login")
    public String login() {
        return "login";
    }

    @GetMapping("/")
    public String home(Authentication authentication, Model model) {
        OAuth2AuthenticationToken oauthToken = (OAuth2AuthenticationToken) authentication;
        OAuth2AuthorizedClient client = authorizedClientService.loadAuthorizedClient(
                oauthToken.getAuthorizedClientRegistrationId(), oauthToken.getName());

        model.addAttribute("user", oauthToken.get().getPrincipal().getAttributes());
        model.addAttribute("accessToken", client.getAccessToken().getTokenValue());

        return "home";
    }
}
```

This controller handles the `/login` endpoint, which displays the login form, and the root endpoint, which displays the user's details and access token after they've successfully logged in.

This is just a basic example of how to use OAuth2 in a Spring Boot application. There are many other configuration options and customization options available in Spring Security OAuth2, depending on your specific use case.

7. What is CSRF?

CSRF stands for Cross-Site Request Forgery. It is a type of web security vulnerability where an attacker tricks a user into performing an action on a website without their knowledge or consent. This can happen when a user is authenticated on a website and visits a malicious website or clicks on a malicious link that sends a request to the authenticated website on behalf of the user.

For example, suppose a user is authenticated on a banking website and clicks on a malicious link that sends a request to transfer money from the user's account to the attacker's account. If the user is not aware of this action, they may unknowingly perform the transfer, resulting in financial loss.

To prevent CSRF attacks, web applications can use techniques like CSRF tokens or SameSite cookies. CSRF tokens are random values generated by the server and included in the web form or URL parameters. When the user submits the form, the server checks the CSRF token to verify that the request is legitimate. SameSite cookies are cookies that restrict the use of cookies to first-party requests only, preventing them from being sent along with cross-site requests.

In addition, web developers can use frameworks and libraries that provide built-in protection against CSRF attacks, such as Spring Security in Java-based web applications.

8. What are anti patterns?

Anti-patterns are common solutions to recurring problems that initially seem like they may solve the problem, but ultimately result in more problems than they solve. They are often thought of as the opposite of design patterns, which are tried and tested solutions to commonly occurring problems.

Anti-patterns can occur at various levels, including software development, software architecture, project management, and organizational culture. They can be caused by various factors, such as lack of understanding of the problem, lack of communication, lack of experience, and pressure to deliver results quickly.

Examples of anti-patterns in software development include "spaghetti code" (code that is overly complex and tangled), "copy-paste programming" (reusing code without understanding it), and "god object" (a single object that handles too many responsibilities).

It's important to recognize and avoid anti-patterns because they can lead to wasted time, increased costs, and low-quality software or projects. By understanding common anti-patterns, developers and managers can make better decisions and avoid repeating mistakes.

9. What is the factory design pattern?

The Factory design pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. The Factory pattern decouples the code that uses the objects from the code that creates the objects.

Here's an example implementation of the Factory pattern in Java:

```java
// Shape.java
public interface Shape {
    void draw();
}

// Rectangle.java
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}

// Square.java
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}

// ShapeFactory.java
public class ShapeFactory {
    public Shape getShape(String shapeType){
        if(shapeType == null){
            return null;
        }
        if(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        } else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        }
        return null;
    }
}

// FactoryPatternDemo.java
public class FactoryPatternDemo {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();
        Shape shape1 = shapeFactory.getShape("RECTANGLE");
        shape1.draw();

        Shape shape2 = shapeFactory.getShape("SQUARE");
        shape2.draw();
    }
}
```

In this example, we have an interface called Shape that defines the draw() method. We have two concrete classes that implement the Shape interface: Rectangle and Square. We also have a ShapeFactory class that provides a method for creating Shape objects based on a string input. Finally, we have a FactoryPatternDemo class that uses the ShapeFactory to create and draw Rectangle and Square objects.

When we run the FactoryPatternDemo class, it will output:

```
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

This example demonstrates how the Factory pattern can be used to create objects without exposing the creation logic to the client code.

10. What is an observable design pattern?

The Observer design pattern is a behavioral design pattern that allows an object (called the subject) to maintain a list of its dependents (called observers) and notify them automatically when any state changes occur.

In the Observer pattern, the subject maintains a list of observers and provides an interface for attaching and detaching observers. When the subject's state changes, it notifies all of its observers, who can then take appropriate action.

This pattern is useful in situations where there is a one-to-many relationship between objects, and when the state of one object needs to be propagated to multiple other objects.

Here's an example of how the Observer pattern can be implemented in Java:

```
// Subject interface
public interface Subject {
    public void attach(Observer observer);
    public void detach(Observer observer);
    public void notifyObservers();
}

// Concrete subject class
public class WeatherStation implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private int temperature;

    public void setTemperature(int temperature) {
        this.temperature = temperature;
        notifyObservers();
    }

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature);
        }
    }
}

// Observer interface
public interface Observer {
    public void update(int temperature);
}

// Concrete observer class
public class PhoneDisplay implements Observer {
    @Override
    public void update(int temperature) {
        System.out.println("Phone display: Temperature is now " + temperature + " degrees.");
    }
}
```

In this example, WeatherStation is the subject that maintains a list of observers and notifies them when the temperature changes. PhoneDisplay is an observer that is interested in changes to the temperature and reacts accordingly. When the temperature changes, WeatherStation calls notifyObservers(), which iterates through its list of observers and calls update() on each observer, passing in the new temperature. In this case, PhoneDisplay receives the update and displays the new temperature on the phone.

Overall, the Observer pattern provides a way to decouple the subject from its observers, allowing for greater flexibility and modularity in your code.

11. What is a facade pattern?

The Facade design pattern is a structural pattern that provides a simplified interface to a complex system of classes, interfaces, and/or APIs. It is used to encapsulate a complex subsystem with a simpler interface, making it easier to use and understand.

The idea behind the Facade pattern is to provide a single entry point or interface to a complex subsystem, hiding its complexity from clients. This can be useful when you have a large and complex set of classes or APIs that need to be used together, but you don't want to expose that complexity to clients.

Here's an example of how the Facade pattern can be implemented in Java:

```
// Complex subsystem
public class SubsystemA {
    public void methodA() {
        System.out.println("SubsystemA: methodA");
    }
}

public class SubsystemB {
    public void methodB() {
        System.out.println("SubsystemB: methodB");
    }
}

public class SubsystemC {
    public void methodC() {
        System.out.println("SubsystemC: methodC");
    }
}

// Facade class
public class Facade {
    private SubsystemA subsystemA;
    private SubsystemB subsystemB;
    private SubsystemC subsystemC;

    public Facade() {
        subsystemA = new SubsystemA();
        subsystemB = new SubsystemB();
        subsystemC = new SubsystemC();
    }

    public void operation() {
        subsystemA.methodA();
        subsystemB.methodB();
        subsystemC.methodC();
    }
}

// Client code
public class Client {
    public static void main(String[] args) {
        Facade facade = new Facade();
        facade.operation();
    }
}
```

In this example, we have three subsystems (SubsystemA, SubsystemB, and SubsystemC) that have their own methods and functionality. We also have a Facade class that encapsulates these subsystems and provides a simplified interface to clients. The Facade class has a single method called operation() that internally calls the methods of the subsystems in a specific order. This way, the complexity of the subsystems is hidden from clients, and they only need to know about the Facade class and its simple interface.

Overall, the Facade pattern provides a way to simplify complex systems by providing a unified and simplified interface to clients. It promotes loose coupling between subsystems and clients, making it easier to maintain and modify the system over time.

12. What is a command pattern?

The Command design pattern is a behavioral pattern that encapsulates a request as an object, thereby allowing for the parameterization of clients with different requests, queueing or logging of requests, and the support of undoable operations.

The Command pattern is useful in situations where you need to decouple the object that invokes a request from the object that actually performs the request. This can be useful in situations where you want to provide a high level of abstraction and flexibility to clients, while still allowing them to interact with specific commands.

Here's an example of how the Command pattern can be implemented in Java:

```
// Command interface
public interface Command {
    public void execute();
}

// Concrete command classes
public class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }
}

public class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOff();
    }
}

// Receiver class
public class Light {
    public void turnOn() {
        System.out.println("Light is on");
    }

    public void turnOff() {
        System.out.println("Light is off");
    }
}

// Invoker class
public class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}

// Client code
public class Client {
    public static void main(String[] args) {
        Light light = new Light();
        Command lightOnCommand = new LightOnCommand(light);
        Command lightOffCommand = new LightOffCommand(light);

        RemoteControl remoteControl = new RemoteControl();
        remoteControl.setCommand(lightOnCommand);
        remoteControl.pressButton(); // Turns the light on

        remoteControl.setCommand(lightOffCommand);
        remoteControl.pressButton(); // Turns the light off
    }
}
```

In this example, we have two concrete command classes (LightOnCommand and LightOffCommand) that implement the Command interface. These command classes encapsulate the actions of turning a light on and off. We also have a Light class, which is the receiver of the commands. Finally, we have an Invoker class (RemoteControl) that sets the command and executes it when the pressButton() method is called.

Overall, the Command pattern provides a way to separate the requester of an action from the object that performs the action, providing a high degree of flexibility and extensibility. It allows for the encapsulation of commands into objects, allowing them to be stored, passed around, and even undone if needed.

13. What is an adapter pattern?

The Adapter pattern is a structural pattern that allows objects with incompatible interfaces to work together. It involves wrapping an existing class with a new interface that is more suitable for the client's needs.

In other words, the Adapter pattern acts as a bridge between two incompatible interfaces, allowing them to communicate with each other. This can be useful in situations where you need to integrate two existing systems that use different interfaces or when you want to reuse an existing class that doesn't quite match the requirements of a new system.

Here's an example of how the Adapter pattern can be implemented in Java:

```
// Adaptee class (the class we want to adapt)
public class LegacyRectangle {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("LegacyRectangle.draw() - x1: " + x1 + ", y1: " + y1 + ", x2: " + x2 + ", y2: " +
y2);
    }
}

// Target interface
public interface Shape {
    void draw(int x, int y, int width, int height);
}

// Adapter class
public class RectangleAdapter implements Shape {
    private LegacyRectangle adaptee;

    public RectangleAdapter(LegacyRectangle adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public void draw(int x, int y, int width, int height) {
        int x1 = x;
        int y1 = y;
        int x2 = x + width;
        int y2 = y + height;
        adaptee.draw(x1, y1, x2, y2);
    }
}

// Client code
public class Client {
    public static void main(String[] args) {
        // Create a LegacyRectangle object (the Adaptee)
        LegacyRectangle legacyRectangle = new LegacyRectangle();

        // Create a RectangleAdapter object (the Adapter)
        RectangleAdapter adapter = new RectangleAdapter(legacyRectangle);

        // Use the adapter to draw a Shape
        adapter.draw(10, 10, 100, 100);
    }
}
```

In this example, we have a LegacyRectangle class that represents an existing object that we want to adapt to a new interface. We define the new Shape interface that the client code expects to work with. We then create an RectangleAdapter class that adapts the LegacyRectangle object to the new Shape interface. Finally, we use the adapter object to draw a Shape in the Client code.

Overall, the Adapter pattern provides a way to make two incompatible interfaces work together. It can be useful when integrating existing systems or when reusing existing classes in a new system.

14. What is a bridge pattern?

The Bridge pattern is a structural design pattern that separates the abstraction from its implementation so that both can be modified independently. It allows the creation of complex objects by composing smaller objects together in a flexible manner.

The Bridge pattern is useful when you want to decouple an abstraction from its implementation, so that changes to one do not affect the other. For example, if you have a shape interface that needs to be rendered on multiple platforms, you could use the Bridge pattern to create a rendering abstraction that can be implemented on each platform independently.

Here's an example of how the Bridge pattern can be implemented in Java:

```
// Abstraction
public abstract class Shape {
    protected DrawingAPI drawingAPI;

    protected Shape(DrawingAPI drawingAPI) {
        this.drawingAPI = drawingAPI;
    }

    public abstract void draw();
}

// Refined Abstraction
public class CircleShape extends Shape {
    private double x, y, radius;

    public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI) {
        super(drawingAPI);
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    @Override
    public void draw() {
        drawingAPI.drawCircle(x, y, radius);
    }
}

// Implementor
public interface DrawingAPI {
    void drawCircle(double x, double y, double radius);
}

// Concrete Implementor 1
public class DrawingAPI1 implements DrawingAPI {
    @Override
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
    }
}

// Concrete Implementor 2
public class DrawingAPI2 implements DrawingAPI {
    @Override
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);
    }
}

// Client
public class Client {
    public static void main(String[] args) {
        Shape[] shapes = {
                new CircleShape(1, 2, 3, new DrawingAPI1()),
                new CircleShape(5, 7, 11, new DrawingAPI2())
        };

        for (Shape shape : shapes) {
            shape.draw();
        }
    }
}
```

In this example, we have an abstraction Shape that is composed of a DrawingAPI implementation. We define two concrete DrawingAPI implementations, DrawingAPI1 and DrawingAPI2. We then create a CircleShape class that extends Shape and implements the draw() method. Finally, we use the CircleShape object to draw a circle using both DrawingAPI implementations.

Overall, the Bridge pattern provides a way to decouple an abstraction from its implementation, allowing for greater flexibility and easier maintenance. It is especially useful when working with complex objects that are composed of smaller objects that may need to be changed independently.

15. What is a builder pattern?

The Builder pattern is a creational design pattern that allows you to create complex objects step by step, using a clear and readable syntax. It is useful when you need to create objects that have a lot of optional properties or when the object creation process is complex.

The Builder pattern separates the construction of an object from its representation, allowing you to create different representations of the same object using the same construction process.

Here's an example of how the Builder pattern can be implemented in Java:

```
// Product
public class Car {
    private String make;
    private String model;
    private int year;
    private int numDoors;
    private boolean hasAC;
    private boolean hasGPS;

    public Car(String make, String model, int year, int numDoors, boolean hasAC, boolean hasGPS) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.numDoors = numDoors;
        this.hasAC = hasAC;
        this.hasGPS = hasGPS;
    }

    // getters and setters...
}

// Builder
public class CarBuilder {
    private String make;
    private String model;
    private int year;
    private int numDoors;
    private boolean hasAC;
    private boolean hasGPS;

    public CarBuilder(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    public CarBuilder withNumDoors(int numDoors) {
        this.numDoors = numDoors;
        return this;
    }

    public CarBuilder withAC() {
        this.hasAC = true;
        return this;
    }

    public CarBuilder withGPS() {
        this.hasGPS = true;
        return this;
    }

    public Car build() {
        return new Car(make, model, year, numDoors, hasAC, hasGPS);
    }
}

// Client
public class Client {
    public static void main(String[] args) {
        Car car = new CarBuilder("Toyota", "Camry", 2022)
                        .withNumDoors(4)
                        .withAC()
                        .withGPS()
                        .build();

        System.out.println(car);
    }
}
```

In this example, we have a Car class that represents a car object with various properties. We also have a CarBuilder class that allows us to create a car object step by step, by setting optional properties using fluent methods. Finally, we create a Client class that uses the CarBuilder to create a Car object.

Overall, the Builder pattern provides a way to create complex objects step by step, using a clear and readable syntax. It is especially useful when you need to create objects with a lot of optional properties or when the object creation process is complex.

16. What is a decorator pattern?

The Decorator pattern is a structural design pattern that allows you to add behavior to an object dynamically, without changing its class. This pattern is useful when you want to add functionality to an object at runtime, and you don't want to modify the existing code.

In the Decorator pattern, you create a set of decorator classes that wrap the original object. Each decorator implements the same interface as the original object and adds additional functionality. Decorators can be stacked on top of each other to add multiple layers of behavior.

Here's an example of how the Decorator pattern can be implemented in Java:

```
// Component interface
public interface Pizza {
    String getDescription();
    double getCost();
}

// Concrete component
public class MargheritaPizza implements Pizza {
    @Override
    public String getDescription() {
        return "Margherita Pizza";
    }

    @Override
    public double getCost() {
        return 6.99;
    }
}

// Decorator
public abstract class PizzaDecorator implements Pizza {
    protected Pizza pizza;

    public PizzaDecorator(Pizza pizza) {
        this.pizza = pizza;
    }

    @Override
    public String getDescription() {
        return pizza.getDescription();
    }

    @Override
    public double getCost() {
        return pizza.getCost();
    }
}
```

```
// Concrete decorator
public class CheeseDecorator extends PizzaDecorator {
    public CheeseDecorator(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + ", extra cheese";
    }

    @Override
    public double getCost() {
        return pizza.getCost() + 1.50;
    }
}

// Client
public class PizzaShop {
    public static void main(String[] args) {
        Pizza margheritaPizza = new MargheritaPizza();
        System.out.println(margheritaPizza.getDescription() + ": $" + margheritaPizza.getCost());

        Pizza cheesePizza = new CheeseDecorator(new MargheritaPizza());
        System.out.println(cheesePizza.getDescription() + ": $" + cheesePizza.getCost());
    }
}
```

In this example, we have a Pizza interface that defines the methods for a pizza object. We also have a MargheritaPizza class that implements the Pizza interface. We then create a PizzaDecorator abstract class that also implements the Pizza interface and contains a reference to the original Pizza object. Finally, we create a CheeseDecorator concrete class that extends the PizzaDecorator and adds extra behavior to the original Pizza object.

In the client code, we create an instance of the original MargheritaPizza and print out its description and cost. We then create a new CheeseDecorator object and pass in the original MargheritaPizza as a parameter. We print out the description and cost of the new pizza, which now includes the extra behavior added by the CheeseDecorator.

Overall, the Decorator pattern provides a way to add behavior to an object dynamically, without changing its class. It allows for flexible and reusable code, as decorators can be stacked on top of each other to add multiple layers of behavior.

17. What is a proxy pattern?

The Proxy pattern is a structural design pattern that provides a surrogate or placeholder for another object to control access to it. In other words, a proxy acts as a wrapper or intermediary around an object, allowing you to perform additional actions before or after accessing the object.

The Proxy pattern is useful in situations where you want to add extra functionality to an existing object without modifying its code. This can include things like caching, logging, or security checks.

Here's an example of how the Proxy pattern can be implemented in Java:

```
// Subject interface
public interface Image {
    void display();
}

// Real subject
public class RealImage implements Image {
    private String fileName;

    public RealImage(String fileName) {
        this.fileName = fileName;
        loadFromDisk(fileName);
    }

    @Override
    public void display() {
        System.out.println("Displaying " + fileName);
    }

    private void loadFromDisk(String fileName) {
        System.out.println("Loading " + fileName);
    }
}
```

```
// Proxy
public class ImageProxy implements Image {
    private String fileName;
    private RealImage realImage;

    public ImageProxy(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(fileName);
        }
        realImage.display();
    }
}

// Client
public class ProxyDemo {
    public static void main(String[] args) {
        Image image = new ImageProxy("test_image.jpg");

        // Image will be loaded from disk
        image.display();

        // Image will not be loaded from disk
        image.display();
    }
}
```

In this example, we have an Image interface that defines the methods for an image object. We also have a RealImage class that implements the Image interface and loads an image from disk. We then create an ImageProxy class that also implements the Image interface and acts as a wrapper around the RealImage object. The ImageProxy class checks whether the RealImage object has already been loaded, and if not, it creates a new RealImage object and loads the image from disk.

In the client code, we create an instance of the ImageProxy and call the display() method. The first time this method is called, the ImageProxy creates a new RealImage object and loads the image from disk. The second time this method is called, the ImageProxy reuses the existing RealImage object and does not need to load the image from disk again.

Overall, the Proxy pattern provides a way to add extra functionality to an object without modifying its code. It can be used to control access to an object, perform additional actions before or after accessing the object, or provide a placeholder for an object until it is actually needed.

18. What is a chain of responsibility pattern?

The Chain of Responsibility pattern is a behavioral design pattern that allows you to pass requests along a chain of handlers until one of them handles the request or the request is not handled at all.

In this pattern, each handler in the chain has a chance to handle the request or pass it on to the next handler in the chain. This provides a way to decouple the sender of a request from its receiver, and allows multiple objects to have a chance to handle the request.

Here's an example of how the Chain of Responsibility pattern can be implemented in Java:

```
// Handler interface
public interface Handler {
    void handleRequest(Request request);
}

// Concrete handler 1
public class ConcreteHandler1 implements Handler {
    private Handler nextHandler;

    public void setNextHandler(Handler nextHandler) {
        this.nextHandler = nextHandler;
    }

    @Override
    public void handleRequest(Request request) {
        if (request.getType() == RequestType.TYPE1) {
            System.out.println("ConcreteHandler1 handles request " + request);
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        } else {
            System.out.println("No handler found for request " + request);
        }
    }
}

// Concrete handler 2
public class ConcreteHandler2 implements Handler {
    private Handler nextHandler;

    public void setNextHandler(Handler nextHandler) {
        this.nextHandler = nextHandler;
    }

    @Override
    public void handleRequest(Request request) {
        if (request.getType() == RequestType.TYPE2) {
            System.out.println("ConcreteHandler2 handles request " + request);
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        } else {
            System.out.println("No handler found for request " + request);
        }
    }
}

// Request class
public class Request {
    private RequestType type;

    public Request(RequestType type) {
        this.type = type;
    }

    public RequestType getType() {
        return type;
    }

    public void setType(RequestType type) {
        this.type = type;
    }

    @Override
    public String toString() {
        return "Request{" +
                "type=" + type +
                '}';
    }
}

// Request type enum
public enum RequestType {
    TYPE1, TYPE2
}

// Client
public class ChainOfResponsibilityDemo {
    public static void main(String[] args) {
        Handler handler1 = new ConcreteHandler1();
        Handler handler2 = new ConcreteHandler2();

        handler1.setNextHandler(handler2);

        handler1.handleRequest(new Request(RequestType.TYPE1));
        handler1.handleRequest(new Request(RequestType.TYPE2));
        handler1.handleRequest(new Request(RequestType.TYPE3));
    }
}
```

In this example, we have a Handler interface that defines the method for handling requests. We then have two concrete handlers, ConcreteHandler1 and ConcreteHandler2, that implement the Handler interface and handle requests of different types. Each handler also has a reference to the next handler in the chain.

We then create a Request class that represents a request to be handled, and a RequestType enum that defines the different types of requests.

In the client code, we create instances of ConcreteHandler1 and ConcreteHandler2, and set up the chain of handlers by setting the next handler for each one. We then create several requests and pass them to the first handler in the chain, handler1. The handleRequest() method for handler1 checks the type of the request and either handles it or passes it on to the next handler in the chain.

Overall, the Chain of Responsibility pattern provides a way to achieve loose coupling between the sender of a request and its receiver, and allows for flexible handling of requests by different objects. It can be useful in scenarios where you have multiple objects that may handle a request, but you don't know which one will handle it until runtime. It also allows you to easily add or remove handlers from the chain without affecting the other objects in the chain.

However, it's important to be careful when implementing the Chain of Responsibility pattern, as it can be easy to create chains that are too long or too complex. This can make the code difficult to understand and maintain, and may result in performance issues if there are too many objects in the chain. It's also important to ensure that every request is handled by at least one object in the chain, to avoid the risk of the request being dropped or ignored.

19. What is a load balancer?

A load balancer is a network device or software application that distributes incoming network traffic across multiple servers or network resources. The purpose of a load balancer is to optimize resource utilization, maximize throughput, minimize response time, and avoid overload on any individual resource.

Load balancers are commonly used in web applications, where they distribute incoming HTTP requests across multiple web servers, each of which can handle a portion of the traffic. The load balancer may use various algorithms to determine how to distribute the traffic, such as round-robin, least connections, IP hash, or weighted distribution.

Load balancers can also provide additional features like SSL offloading, session persistence, health checks, and failover. SSL offloading offloads the SSL encryption and decryption process from the web servers to the load balancer, which can improve performance and reduce server workload. Session persistence ensures that subsequent requests from the same client are sent to the same web server, preserving session state. Health checks monitor the status of the web servers and remove any servers that are not responding or are experiencing high load. Failover ensures that if a web server fails, its traffic is automatically redirected to another healthy web server.

There are different types of load balancers, such as hardware load balancers and software load balancers. Hardware load balancers are physical devices that are dedicated to load balancing, while software load balancers are software applications that can run on commodity hardware or in the cloud.

Overall, load balancers play a critical role in scaling and optimizing web applications by distributing traffic across multiple resources, providing high availability and improved performance, and reducing the risk of overload or downtime.

20. What are the different implementations of Singleton Pattern?

- Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the Java Virtual Machine.
- The singleton class must provide a global access point to get the instance of the class.
- Singleton pattern is used for logging, drivers objects, caching, and thread pool.
- Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade, etc.
- Singleton design pattern is used in core Java classes also (for example, java.lang.Runtime, java.awt.Desktop).

To implement a singleton pattern, we have different approaches, but all of them have the following common concepts.

- Private constructor to restrict instantiation of the class from other classes.
- Private static variable of the same class that is the only instance of the class.
- Public static method that returns the instance of the class, this is the global access point for the outer world to get the instance of the singleton class.

Eager initialization:

```
public class EagerSingleton {
    private static final EagerSingleton instance = new EagerSingleton();
    private EagerSingleton() {}
    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

Lazy initialization:

```
public class LazySingleton {
    private static LazySingleton instance;
    private LazySingleton() {}
    public static synchronized LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}
```

Thread-safe lazy initialization:

```
public class ThreadSafeSingleton {
    private static volatile ThreadSafeSingleton instance;
    private ThreadSafeSingleton() {}
    public static ThreadSafeSingleton getInstance() {
        if (instance == null) {
            synchronized (ThreadSafeSingleton.class) {
                if (instance == null) {
                    instance = new ThreadSafeSingleton();
                }
            }
        }
        return instance;
    }
}
```

Static block initialization:

```
public class StaticBlockSingleton {
    private static StaticBlockSingleton instance;
    private StaticBlockSingleton() {}
    static {
        try {
            instance = new StaticBlockSingleton();
        } catch (Exception e) {
            throw new RuntimeException("Error initializing singleton instance");
        }
    }
    public static StaticBlockSingleton getInstance() {
        return instance;
    }
}
```

Bill Pugh Implementation:

```
public class BillPughSingleton {
    private BillPughSingleton(){}
    private static class SingletonHelper {
        private static final BillPughSingleton INSTANCE = new BillPughSingleton();
    }

    public static BillPughSingleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

Enum Implementation:

```
public enum EnumSingleton {

    INSTANCE;

    public static void doSomething() {
        // do something
    }
}
```

Serialization Implementation:

```
import java.io.Serializable;

public class SerializedSingleton implements Serializable {

    private static final long serialVersionUID = -7604766932017737115L;
    private SerializedSingleton(){}
    private static class SingletonHelper {
        private static final SerializedSingleton instance = new SerializedSingleton();
    }
    public static SerializedSingleton getInstance() {
        return SingletonHelper.instance;
    }
}
```

21. What are different ways we can break Singleton?

Singleton is a creational design pattern that allows ensuring that a class has only one instance, while providing a global point of access to that instance.

However, there are a few ways that a singleton instance can be broken, including:

- Reflection: With reflection, it's possible to access a class's private constructor and create a new instance of the singleton class.

To prevent a singleton from being created via reflection, we can add a check in the Singleton constructor to throw an exception if the instance already exists:

```
private Singleton() {
    if (INSTANCE != null) {
        throw new IllegalStateException("Singleton already created");
    }
    // initialize instance
}
```

- Serialization: If a Singleton class is made Serializable and then deserialized more than once, it will create a new instance of the Singleton class.

To prevent a singleton from being created via deserialization, we can implement the readResolve() method, which will return the existing instance:

```
private Object readResolve() {
    return INSTANCE;
}
```

- Multithreading: If multiple threads access the Singleton instance concurrently, it can create multiple instances.

To prevent a singleton from being created via concurrent access, we can make the instance creation synchronized:

```
private static synchronized Singleton getInstance() {
    if (INSTANCE == null) {
        INSTANCE = new Singleton();
    }
    return INSTANCE;
}
```

- Classloaders: If the Singleton class is loaded by multiple classloaders, it can create multiple instances.

To prevent a singleton from being created via different classloaders, we can make the singleton class final and make sure it is only loaded by the system classloader:

```
public final class Singleton {
    private static Singleton INSTANCE = null;
    static {
        ClassLoader loader = Singleton.class.getClassLoader();
        if (loader != null) {
            throw new IllegalStateException("Singleton loaded by a non-system classloader");
        }
    }
    // ...
}
```

To prevent these problems, it's recommended to use an enum to implement the Singleton pattern, as enums are inherently serializable, reflectively immutable, and thread-safe by default. Alternatively, lazy initialization with double-checked locking can also be used, but it requires careful synchronization to be thread-safe.

22. Design a Logger Class using a singleton pattern.

```
public class Logger {
    private static Logger instance;
    private Logger() {}

    public static synchronized Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    public void log(String message) {
        // Logging implementation
        System.out.println(message);
    }
}
```
In this example, the Logger class is designed as a Singleton, meaning that only one instance of the class can be created. The class has a private constructor, which prevents it from being instantiated from outside the class. Instead, the class provides a static method, getInstance(), which returns the one and only instance of the Logger class.

The log() method is a simple example of a logging function that accepts a message as a parameter and logs it to the console.

This design pattern ensures that only one instance of the Logger class exists at any given time, which is useful for managing resources and ensuring that the logging system is consistent throughout the application.

23. What is a strangler pattern?

The Strangler Pattern is a technique used in software engineering to manage the migration of an existing system to a new architecture. It is often used when transitioning from a monolithic architecture to a microservices architecture.

The idea behind the pattern is to gradually replace the functionality of the old system with new microservices, by gradually "strangling" the old system as the new system is built up around it. This can help to minimize the risks associated with a complete system rewrite, and can allow for a more gradual and incremental migration to the new architecture.

As an example, consider a monolithic e-commerce system that is being migrated to a microservices architecture. Rather than attempting to rewrite the entire system from scratch, the Strangler Pattern can be used to gradually build up new microservices around the existing system, while slowly taking over more and more of its functionality. For example, a new product catalog service could be built and deployed alongside the existing system, while still using the same database. As the new service matures and takes on more functionality, it can slowly replace the old system, until the old system is eventually phased out entirely.

In terms of implementation, the Strangler Pattern can be facilitated by a number of different approaches, such as using a service gateway to route requests between the old and new systems, or using event-driven architecture to gradually introduce new functionality and services.

24. How to break Monolith into multiple Microservices?

Breaking a monolith into multiple microservices is not a trivial task and requires a well-thought-out plan and strategy. Here are some high-level steps that can be taken to break a monolith into multiple microservices:

- Identify the domain model: Understand the domain model of the monolith and identify its various components.
- Analyze the monolith: Analyze the monolith to identify the different modules or functionalities that can be separated into standalone services.
- Define the service boundaries: Identify the clear boundaries between the different services and ensure that the data is not shared between them.
- Refactor the code: Refactor the code to break the monolith into smaller, more manageable services.
- Choose the right technology: Choose the right technology stack for each microservice, keeping in mind the unique needs and requirements of each service.
- Implement communication protocols: Implement communication protocols between the different services to ensure that they can interact with each other seamlessly.
- Monitor and maintain the microservices: Monitor and maintain the microservices to ensure that they are functioning optimally and can be scaled up or down as required.
- Continuously improve the architecture: Continuously improve the microservices architecture based on the feedback received from the users.

It is also important to keep in mind that breaking a monolith into microservices is not a one-time activity, and it requires continuous effort to ensure that the services are functioning optimally and meeting the business requirements.

25. What are the advantages and disadvantages of Monolithic architecture?

Monolithic architecture is a traditional architectural style for building applications where all the components of an application are bundled into a single executable file. The advantages and disadvantages of Monolithic architecture are:

Advantages:

- Simplicity: Monolithic architecture is simple to design, develop, test, and deploy since everything is contained in a single application.
- Easier debugging: Since all components are in a single application, it is easier to debug and fix any errors in the code.
- Better performance: Monolithic architecture can provide better performance since all components are running in a single process, there is no need for inter-process communication, and it is easier to optimize performance.

Disadvantages:

- Limited scalability: Scaling a monolithic application can be difficult as the entire application needs to be scaled up, even if only a single component needs more resources.
- Limited technology choices: A monolithic application may be limited to a specific technology stack since all components have to be developed using the same technology.
- Longer development time: Since all components are developed at the same time, it can lead to longer development time and slower time-to-market.
- High risk of failure: A monolithic application has a single point of failure, which means that if one component fails, it can bring down the entire application.
- Difficult to maintain: As the application grows, it can become difficult to maintain and make changes to the code.

These limitations led to the rise of microservices architecture, where applications are broken down into smaller, independent services that can be developed, deployed, and scaled independently.

26. What are the advantages and disadvantages of Microservices architecture?

Microservices architecture has the following advantages:

- Scalability: Microservices can be scaled independently, allowing for better resource utilization and efficiency.
- Flexibility: Each microservice can be developed, tested, and deployed independently, allowing for faster time-to-market and the ability to make changes without affecting the entire system.
- Resilience: If a microservice fails, the rest of the system can continue to function.
- Better fault isolation: Faults can be isolated to a single microservice, reducing the impact on the entire system.
- Easier to maintain: Smaller, more focused microservices are easier to maintain and understand.

However, microservices architecture also has the following disadvantages:

- Increased complexity: Managing multiple microservices can be complex, especially as the system grows in size.
- Higher cost: Building and managing microservices requires more resources, which can be expensive.
- Operational overhead: Each microservice requires its own infrastructure and operational support, which can add to the operational overhead.
- Distributed systems challenges: Microservices require communication across network boundaries, which can lead to latency and other network-related issues.
- Testing challenges: Testing a system with multiple microservices can be more challenging than testing a monolithic system.

27. What is Microservice Orchestration and Choreography?

Microservice Orchestration and Choreography are two approaches for managing the communication and coordination between microservices in a distributed system.

Orchestration involves having a central service, often referred to as an orchestrator or workflow engine, that is responsible for coordinating the flow of messages between microservices. The orchestrator maintains the state of the overall workflow and coordinates the invocation of individual microservices. This approach can provide centralized control and visibility into the overall system, but can also introduce a single point of failure.

Choreography, on the other hand, involves each microservice communicating directly with one or more other microservices to coordinate their actions. Each service is responsible for managing its own state and knowing how to interact with other services. This approach can be more distributed and scalable than orchestration, but can also be more difficult to manage and monitor.

In summary, both orchestration and choreography have their own advantages and disadvantages, and the choice between the two depends on the specific needs and requirements of the system being built.

28. What are different ways in which two microservices communicate?

In a microservices architecture, there are different ways in which two microservices within the same application can communicate with each other. Some of the common ways are:

- Synchronous communication: In this type of communication, the client sends a request to the server, and the server sends a response back. This can be achieved using REST APIs or RPC (Remote Procedure Call) protocols like gRPC.
- Asynchronous communication: In this type of communication, the sender does not wait for a response from the receiver. Instead, it sends a message to a message broker like RabbitMQ or Apache Kafka, which is responsible for delivering the message to the receiver. The receiver can then process the message at its own pace.
- Event-driven communication: In this type of communication, microservices communicate with each other through events. An event is a signal that something has happened, such as a new user being created or an order being placed. The event is broadcasted to all interested parties, and each microservice can choose to act on the event if it needs to.
- Shared database: In some cases, microservices may share a common database. In this case, one microservice can write data to the database, and another microservice can read the data from the same database. However, this approach can create coupling between the microservices and can make it difficult to scale the application.
- In-process communication: In this type of communication, multiple microservices run within the same process or JVM. They can communicate with each other using in-memory data structures like ConcurrentHashMap or message queues. However, this approach can create tight coupling between the microservices and can make it difficult to deploy them independently.

The choice of communication mechanism depends on the specific requirements of the application and the constraints of the underlying infrastructure.

29. How to check the health of microservices using an actuator?

Spring Boot Actuator provides a set of endpoints to monitor and manage the health of a microservice. To check the health of a microservice using an actuator, follow these steps:

Add the Spring Boot Actuator dependency to the pom.xml file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Configure the Actuator endpoints in the application.properties file:

```
management.endpoints.web.exposure.include=health,info
```

This will enable the /actuator/health and /actuator/info endpoints.

Start the microservice and access the /actuator/health endpoint in a web browser or using a command-line tool like curl:

```
$ curl http://localhost:8080/actuator/health
```

The response will indicate the health status of the microservice.

You can also customize the health endpoint to include additional information, such as database connectivity, by implementing a HealthIndicator and registering it with the Actuator.

30. What is push and pull based CDN?

Content Delivery Network (CDN) is a distributed network of servers that can efficiently deliver web content to users by caching and distributing it from servers that are closest to the user's location. There are two main types of CDN architectures: push-based CDN and pull-based CDN.

In a push-based CDN, the content is pushed to all the edge servers in the network from a central location, typically the origin server. This approach ensures that the content is available at all the edge servers before the user requests it. Push-based CDNs are often used for large-scale delivery of static content such as images, videos, and other multimedia files.

In a pull-based CDN, the content is only delivered to the edge server when the user requests it. When the user requests a piece of content, the edge server closest to the user retrieves it from the origin server and caches it for subsequent requests. This approach is more efficient for delivering dynamic content that changes frequently, such as news articles, weather updates, or e-commerce product information. The pull-based CDN can also scale more easily, as the number of edge servers can be increased or decreased on demand to handle changes in traffic.

31. What is a video aggregator?

A video aggregator is a website or application that collects and curates video content from multiple sources and displays it in a single location. It allows users to easily browse and watch videos from various sources without having to navigate to each source individually. Video aggregators can aggregate videos from a variety of sources including YouTube, Vimeo, and other video hosting platforms, as well as content from social media platforms and other websites. Some popular video aggregators include YouTube, Vimeo, and Dailymotion.

32. What is a marker interface?

A marker interface is a type of interface in Java that doesn't contain any method declarations. Its purpose is to mark a class as having a particular property or behavior, which can be detected at runtime by other code.

For example, the java.io.Serializable interface is a marker interface that indicates that a class can be serialized and deserialized. When a class implements the Serializable interface, it can be written to a stream and then read back from the stream, allowing it to be transmitted across a network or stored to disk.

Marker interfaces are also used by frameworks to enable certain behaviors or optimizations. For example, the Spring Framework uses marker interfaces such as @Repository or @Component to enable certain behavior, such as automatic bean detection and configuration.

Since a marker interface has no methods, it doesn't impose any implementation requirements on the class that implements it. It simply serves as a marker for the JVM to detect certain behaviors or properties of the class.

Here's an example of a marker interface in Java:

```
public interface Serializable {
    // This is a marker interface that indicates a class can be serialized
}
```

In this example, Serializable is a built-in marker interface in Java that indicates that a class can be serialized, which means it can be converted into a stream of bytes and stored or transmitted across a network. Any class that implements the Serializable interface can be serialized by Java's built-in serialization mechanism.

Here's an example of a class that implements the Serializable marker interface:

```
import java.io.Serializable;

public class Person implements Serializable {
    private String name;
    private int age;
    private String address;

    // constructor, getters and setters
}
```

By implementing the Serializable interface, the Person class can now be serialized and deserialized using Java's built-in serialization mechanism, without the need for any additional code. This is because the Serializable interface serves as a marker to indicate that the class can be serialized.

33. What is a local artifactory?

A local Artifactory is a repository manager that helps to manage software artifacts. It acts as an intermediary between developers and the outside world, allowing them to store, organize, and retrieve artifacts such as JAR files, Docker images, and more. It is a tool that helps in storing and sharing binary artifacts, such as libraries, build outputs, and other dependencies that are needed to build and run applications.

A local Artifactory is installed on-premises or in a private cloud, providing the user with complete control over their artifacts. It is used to improve the build process by automating the process of resolving dependencies and managing builds. It also reduces the time it takes to build an application by providing quick access to the required artifacts. In addition, it provides a central location for all artifacts, ensuring that developers are working with the same versions of libraries and components.

Some examples of local Artifactory tools include JFrog Artifactory, Sonatype Nexus, and Apache Archiva.

34. What is sharding?

Sharding is a database architecture technique used to horizontally partition data across multiple servers or nodes, with each node containing a subset of the data. It is a technique used to improve the scalability, performance, and availability of a database system by distributing the load across multiple servers.

In a sharded database system, data is divided into multiple partitions, known as shards, and each shard is stored on a separate server. Each shard contains a subset of the data and is independent of the other shards. Sharding allows data to be distributed across multiple servers, which allows for better scalability and performance.

Sharding can be done in different ways, including by key range, hash-based sharding, or even a combination of both. In key range sharding, data is partitioned based on a predefined range of values of a particular key. In hash-based sharding, the data is partitioned based on the hash value of a particular key.

Sharding is commonly used in large-scale distributed systems and is particularly popular in web applications, where scalability is a critical requirement. However, it comes with some trade-offs and complexities, such as data consistency, partitioning strategy, and maintenance.

35. How is sharding different from partitioning?

Sharding and partitioning are related concepts, but they are not exactly the same thing.

Sharding involves breaking a large data set into smaller, more manageable parts called shards, which can be distributed across multiple servers or nodes in a distributed system. Each shard contains a subset of the data, and the data is distributed based on some criteria, such as a shard key or hash function. Sharding is often used to improve performance and scalability in systems that deal with large amounts of data, by allowing the system to distribute the data processing and storage load across multiple nodes.

Partitioning, on the other hand, involves dividing a data set into smaller, more manageable parts called partitions, which can be processed in parallel. Partitioning is a common technique used in parallel computing, distributed systems, and databases to improve performance and scalability by allowing multiple processors or nodes to work on different parts of the data set simultaneously. Unlike sharding, partitioning does not necessarily involve distributing the data across multiple nodes.

In summary, sharding is a specific form of partitioning that involves distributing data across multiple nodes or servers, while partitioning is a more general technique for dividing a data set into smaller parts that can be processed in parallel.

36. What is latency, throughput, and availability of a system?

Latency, throughput, and availability are important metrics used to measure the performance and reliability of a system:

- Latency: It refers to the time taken for a system to respond to a request or perform an operation. In simple terms, it is the delay between the initiation of an action and the response. Low latency is desirable as it ensures that the system is responsive and provides a good user experience.
- Throughput: It refers to the number of requests that a system can handle within a given period of time. It is typically measured in requests per second (RPS) or transactions per second (TPS). High throughput is desirable as it ensures that the system can handle a large number of requests without getting overloaded.
- Availability: It refers to the percentage of time that a system is operational and available for use. It is typically measured as a percentage of the total time in a given period. High availability is desirable as it ensures that the system is reliable and can be accessed by users at all times.

In summary, latency is the delay between a request and response, throughput is the number of requests the system can handle, and availability is the percentage of time the system is operational. All three metrics are important in evaluating the performance and reliability of a system.

37. How is Horizontal scaling different from Vertical scaling?

Horizontal scaling and vertical scaling are two different approaches to increasing the capacity of a system.

Horizontal scaling involves adding more machines to a system to increase its capacity. This is also known as scaling out. With horizontal scaling, the workload is distributed across multiple machines, which can help improve the overall performance and availability of the system. Horizontal scaling can be achieved by adding more nodes to a cluster or by using load balancers to distribute traffic across multiple machines.

Vertical scaling, on the other hand, involves increasing the resources (such as CPU, memory, and storage) of a single machine to increase its capacity. This is also known as scaling up. With vertical scaling, a single machine can handle more workload and process more requests. Vertical scaling can be achieved by upgrading the hardware of a machine, such as adding more RAM, CPUs, or hard disks.

The main difference between horizontal scaling and vertical scaling is that horizontal scaling involves adding more machines to a system, while vertical scaling involves increasing the resources of a single machine. Horizontal scaling is typically used in distributed systems where workloads are distributed across multiple machines, while vertical scaling is used in monolithic systems where a single machine needs to handle a large workload.

38. What is the definition of marshaling?

Marshalling (also known as serialization) is the process of converting an object into a format that can be easily transmitted or stored, such as a byte array, a string, or an XML document. The marshalling process typically involves converting an object's data into a byte stream that can be transmitted over a network or written to disk, along with information about the object's type and structure. The resulting marshalled data can then be unmarshalled (or deserialized) at a later time, and used to recreate the original object. Marshalling is commonly used in distributed computing environments, such as client-server systems, where objects may need to be transmitted across a network from one process to another.

39. What are SOLID principles?

SOLID principles are a set of design principles for object-oriented programming that were introduced by Robert C. Martin in the early 2000s. The SOLID acronym stands for:

Each of these principles focuses on a specific aspect of software design, with the goal of making the code more maintainable, flexible, and easy to understand.

- Single Responsibility Principle (SRP): A class should have only one reason to change. It means a class should have only one responsibility, and all the functions of that class should be directly related to that responsibility. It helps in maintaining the code, making it more readable, and easier to test.
- Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. It means that new features can be added to the software without changing the existing code. It helps in keeping the code stable and preventing regressions.
- Liskov Substitution Principle (LSP): Subtypes must be substitutable for their base types. It means that a derived class should be able to replace its base class without affecting the correctness of the program. It helps in writing correct and maintainable code.
- Interface Segregation Principle (ISP): A client should not be forced to depend on methods it does not use. It means that the interface of a class should be tailored to the specific needs of its clients. It helps in reducing the coupling between classes.
- Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions. It means that the high-level modules should depend on abstractions instead of concrete implementations. It helps in making the code more flexible and easier to test.

By following these principles, developers can write code that is more modular, easier to maintain, and less prone to errors or regressions.

40. What is the CAP theorem?

The CAP theorem, also known as Brewer's theorem, is a fundamental concept in distributed computing that states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees:

Consistency (C): Every read receives the most recent write or an error.

Availability (A): Every request receives a response, without guarantee that it contains the most recent version of the information.

Partition tolerance (P): The system continues to operate despite arbitrary partitioning due to network failures.

According to the CAP theorem, a distributed system can provide at most two of these guarantees, but not all three at the same time. Therefore, when designing a distributed system, the system architect must decide which two guarantees are most important and design the system accordingly.

41. What is Http Codes 204, 304, 404, 502, 403, 401 and 503??

HTTP status codes are three-digit numbers returned by servers to indicate the status of a response to a client's request. Here are the meanings of the specific status codes you mentioned:

- 204 No Content: The server has successfully processed the request and is not returning any content. This is often used for DELETE or PUT requests.
- 304 Not Modified: The resource requested by the client has not been modified since the last time it was accessed, and the server is not sending a new copy of the resource in the response. This is typically used for caching purposes.
- 404 Not Found: The server was unable to find the requested resource. This can happen if the resource has been moved or deleted, or if the URL provided by the client is incorrect.
- 502 Bad Gateway: This error is typically returned by a proxy server when it is unable to communicate with an upstream server to fulfill the client's request.
- 403 Forbidden: This code indicates that the server understood the request but refuses to authorize it. This status is similar to a 401 (Unauthorized) status, but indicates that the server understands the request and is refusing to fulfill it. This can happen when the user doesn't have sufficient permissions to access the requested resource, or when the server is configured to block certain requests.
- 401 Unauthorized: This code indicates that the client needs to authenticate itself to get the requested response. This can happen when the user attempts to access a protected resource without providing valid credentials or when the credentials provided are invalid.
- 503 Service Unavailable: This code indicates that the server is currently unable to handle the request due to a temporary overload or maintenance. The server may include a Retry-After header indicating how long the client should wait before retrying the request.

42. What is the different between POST and PUT request?

In the context of HTTP methods, both POST and PUT are used to send data to a server. However, they have different meanings and intended uses.

POST is typically used to submit a new entity to the server. In other words, if you want to create a new resource on the server, you would use a POST request. The data is sent in the request body, and the server will create a new resource and return a response indicating the status of the operation.

PUT, on the other hand, is typically used to update an existing entity on the server. In other words, if you want to modify an existing resource, you would use a PUT request. The data is also sent in the request body, but the server will update an existing resource rather than creating a new one.

In summary, POST is used to create a new resource on the server, while PUT is used to update an existing resource.

43. Can we use PUT to create data on server and what is the difference between PUT and PATCH?

Yes, it is possible to use the PUT method to create data on the server. The PUT method is generally used to update or replace an existing resource on the server with the content of the request. However, some APIs and servers may also allow creating new resources using the PUT method by specifying the URI of the new resource in the request.

In general, the decision to use PUT or POST to create new resources on a server depends on the specific requirements and design of the API or system being used. Both PUT and PATCH are HTTP methods used to update a resource on a server, but they have some differences in how they are used.

PUT is a method used to update a resource on a server. When using PUT, the client sends the entire updated resource to the server, including any fields that were not changed. The server then replaces the existing resource with the new one. If the resource does not exist, the server creates it.

PATCH, on the other hand, is used to update a specific field of a resource, rather than the entire resource. The client sends the specific field that needs to be updated to the server, and the server replaces that field with the new value. PATCH is useful when you only need to update a small portion of a resource, rather than the entire resource.

In summary, PUT is used to replace the entire resource, while PATCH is used to update a specific field of the resource.

44. What is consistent hashing?

Consistent hashing is a technique used in distributed computing and database systems to balance the load across multiple servers while maintaining high availability and minimizing the amount of data that needs to be moved when nodes are added or removed from the system.

In traditional hashing algorithms, a key is hashed to a fixed set of nodes or buckets, which are responsible for storing and retrieving the data associated with that key. However, in a distributed system, the number of nodes or buckets can vary dynamically, which can lead to significant data movement when nodes are added or removed, and can affect the availability and performance of the system.

Consistent hashing addresses this issue by mapping each key to a node in a way that is deterministic, but also minimizes the amount of data that needs to be moved when nodes are added or removed. This is achieved by using a hash function that maps keys to a continuous range of values, such as a circle or a line. Each node is assigned a position on this range based on its own hash value, and keys are then mapped to the node whose position is closest to the key's own hash value.

Because each node is responsible for a continuous range of hash values, adding or removing nodes only affects a small portion of the keys that are mapped to each node, resulting in minimal data movement and better overall system performance and availability.

Here's an example of consistent hashing in Java using the Apache Cassandra library:

```
import com.google.common.hash.Hashing;
import org.apache.cassandra.dht.Murmur3Partitioner;
import org.apache.cassandra.dht.Token;
import java.nio.charset.StandardCharsets;
import java.util.SortedMap;
import java.util.TreeMap;

public class ConsistentHashingExample {
    private SortedMap<Token, String> circle = new TreeMap<Token, String>(new Murmur3Partitioner());

    // Add a node to the hash ring
    public void addNode(String node) {
        circle.put(new Murmur3Partitioner().getToken(Hashing.sha256().hashString(node,
StandardCharsets.UTF_8)), node);
    }

    // Remove a node from the hash ring
    public void removeNode(String node) {
        Token token = new Murmur3Partitioner().getToken(Hashing.sha256().hashString(node,
StandardCharsets.UTF_8));
        circle.remove(token);
    }

    // Get the node responsible for a given key
    public String getNodeForKey(String key) {
        if (circle.isEmpty()) {
            return null;
        }
        Token token = new Murmur3Partitioner().getToken(Hashing.sha256().hashString(key,
StandardCharsets.UTF_8));
        if (!circle.containsKey(token)) {
            SortedMap<Token, String> tailMap = circle.tailMap(token);
            token = tailMap.isEmpty() ? circle.firstKey() : tailMap.firstKey();
        }
        return circle.get(token);
    }

    public static void main(String[] args) {
        ConsistentHashingExample ch = new ConsistentHashingExample();
        ch.addNode("node1");
        ch.addNode("node2");
        ch.addNode("node3");
        System.out.println(ch.getNodeForKey("key1")); // node2
        System.out.println(ch.getNodeForKey("key2")); // node1
        ch.removeNode("node2");
        System.out.println(ch.getNodeForKey("key1")); // node3
        System.out.println(ch.getNodeForKey("key2")); // node3
    }
}
```

In this example, we implement a simple consistent hashing scheme using the Murmur3 hash function provided by the Apache Cassandra library. We use a sorted map to represent the hash ring, with each node assigned a token based on its hash value. To add or remove a node from the ring, we simply add or remove its token from the map. To find the node responsible for a given key, we hash the key and look for the first token in the map that is greater than or equal to the key's hash value. If there is no such token, we wrap around to the beginning of the map.

We test our consistent hashing scheme by adding three nodes ("node1", "node2", and "node3") and assigning two keys ("key1" and "key2") to nodes using the getNodeForKey method. We then remove "node2" from the ring and reassign the keys using the same method. The output of the program should be "node2" for "key1" and "node1" for "key2" before removing "node2", and "node3" for both keys after removing "node2".

45. What are bloom filters?

Bloom filters are a probabilistic data structure used to test whether an element is a member of a set. They were invented by Burton Howard Bloom in 1970.

A bloom filter consists of a bit array and a set of hash functions. To add an element to the filter, the element is hashed by each of the hash functions, and the corresponding bits in the bit array are set to 1. To test whether an element is a member of the set, the element is hashed by each of the hash functions, and the corresponding bits in the bit array are checked. If any of the bits are 0, then the element is definitely not in the set. If all of the bits are 1, then the element is probably in the set, but there is a small probability of a false positive (i.e., the filter reports that the element is in the set even though it is not).

The probability of a false positive can be controlled by adjusting the size of the bit array and the number of hash functions used. A larger bit array and more hash functions will reduce the probability of false positives, but will also increase the space requirements and the computational cost of adding elements and testing for membership.

Bloom filters are useful in situations where the set of elements to be tested for membership is large and the cost of false positives is low. They are commonly used in network protocols, databases, and search engines to quickly filter out elements that are definitely not in the set before performing more expensive tests.

Here's an example of how to use a Bloom filter in Java using the Guava library:

```
import com.google.common.hash.BloomFilter;
import com.google.common.hash.Funnels;

public class BloomFilterExample {
    public static void main(String[] args) {
        // Create a Bloom filter with a maximum of 1000 elements and a false positive rate of 0.01
        BloomFilter<String> filter = BloomFilter.create(Funnels.stringFunnel(), 1000, 0.01);

        // Add some elements to the filter
        filter.put("apple");
        filter.put("banana");
        filter.put("orange");

        // Test for membership
        System.out.println(filter.mightContain("apple")); // true
        System.out.println(filter.mightContain("grape")); // false
    }
}
```

In this example, we create a Bloom filter with a maximum of 1000 elements and a false positive rate of 0.01. We then add some elements to the filter ("apple", "banana", and "orange") and test for membership using the mightContain method. The output of the program should be true for "apple" and false for "grape".

46.
47. Explain in detail SkipList, HashIndex, SSTable, LSM tree, B-tree, Inverted Index, Suffix tree and R-tree.

Database Interview Questions:

1. How is Indexing done in Databases?

Indexing is a technique used in databases to optimize the performance of queries by providing faster access to data. An index is a data structure that stores a copy of a portion of a table's data in a way that allows for efficient retrieval based on the values of one or more columns.

In a database, indexing works by creating a separate data structure that contains a sorted list of values for one or more columns in a table. The index is usually stored in a separate file or data structure, and it points to the locations of the actual data in the table. When a query is executed against the table, the database engine can use the index to locate the data more quickly, instead of scanning the entire table.

There are several types of indexes that are commonly used in databases, including:

- B-tree index: This is the most common type of index, which stores the values in a balanced tree structure. B-tree indexes are efficient for range queries and can be used with columns that have a wide range of values.

Suppose we have a database table with the following columns:

```
id (integer)
name (string)
age (integer)
email (string)
```

To optimize queries that involve searching by the id column, we can create a B-tree index on that column. This will allow us to quickly find rows with a specific id value, without having to scan the entire table.

Here is an example of how to create a B-tree index on the id column using SQL syntax:

```
CREATE INDEX id_index ON table_name (id);
```

This will create a new index called id_index on the id column of the table_name table.

Now, when we execute a query that involves searching by id, the database system will use the B-tree index to quickly find the relevant rows. For example:

```
SELECT * FROM table_name WHERE id = 123;
```

This query will use the B-tree index to locate the row with id = 123 and return its values for all columns.

- Bitmap index: This type of index is used for columns with a small number of distinct values, such as gender or boolean columns. Bitmap indexes use a bitmap to represent the presence or absence of a value in the index.

In a bitmap index, each bit in the bitmap corresponds to a single value in the indexed column. If the value is present in a row, the corresponding bit is set to 1. If the value is not present in a row, the bit is set to 0. This allows for fast, efficient querying of the data, as bitwise operations can be used to combine bitmaps and find the intersection of the rows that match a particular query.

One advantage of bitmap indexing over B-tree indexing is that it can be faster for certain types of queries, particularly those that involve boolean expressions or set operations. For example, a query that asks for all rows where a particular value is present in one column and another value is present in a different column can be answered quickly using bitmap indexes.

However, bitmap indexing is generally less space-efficient than B-tree indexing, as it requires one bit per row per indexed value. This means that for columns with many distinct values, the bitmaps can become quite large and may not fit in memory, which can affect performance.

Here's an example of how bitmap indexing might be used in a database:

Suppose we have a table of sales data with columns for the date of the sale, the product sold, and the region where the sale took place. We might create a bitmap index on the "product" column to allow for efficient querying of the data by product. For each distinct product, we would create a bitmap that indicates which rows in the table contain that product. To answer a query for all sales of product X in region Y, we could combine the bitmaps for product X and region Y using a bitwise AND operation, which would give us a bitmap indicating the rows where both conditions are true. We could then scan the table to retrieve the actual sales data for those rows.

- Hash index: This type of index is used for columns with a large number of distinct values, such as IDs or phone numbers. Hash indexes use a hash function to map the values to a unique index value.

One of the advantages of hash indexing is that it can be very efficient for equality searches (e.g., WHERE clause queries that check for exact matches). However, it is not well-suited for range queries, since the hash function does not preserve order.

Here is an example of how hash indexing might work in a database:

Suppose we have a table called "employees" with the following columns:

```
employee_id
```

```
first_name
last_name
salary
```

To create a hash index on the "employee_id" column, we would first define a hash function that maps each employee_id to a specific location in the index structure. For example, we might use the following hash function:

```
hash(employee_id) = employee_id % 1000
```

This hash function would map each employee_id to a value between 0 and 999, which corresponds to a specific index block.

We would then create an index structure that consists of 1000 blocks, each of which corresponds to a specific range of employee_id values. Each index block would contain pointers to the corresponding data blocks in the table.

When a query is executed that includes a WHERE clause on the employee_id column, the hash function is used to locate the corresponding index block(s), which contain pointers to the relevant data blocks. The data blocks are then read to retrieve the required rows.

The process of creating an index involves selecting the columns to be indexed, defining the type of index to be created, and then building the index using the data from the table. It is important to choose the right columns to index, as indexing too many columns can lead to decreased performance due to increased overhead and maintenance costs.

In summary, indexing is a technique used in databases to optimize the performance of queries by providing faster access to data. It works by creating a separate data structure that contains a sorted list of values for one or more columns in a table, and is used by the database engine to locate the data more quickly when executing queries. There are several types of indexes that are commonly used in databases, each with its own advantages and limitations.

2. What is partition in mongodb?

In MongoDB, partition refers to the process of dividing a large database or collection into smaller, more manageable pieces called shards. Sharding helps distribute data across multiple servers, allowing for better scalability and performance. Each shard consists of a subset of the data, and the data is partitioned based on a shard key. The shard key determines how the data is distributed across the shards.

Partitioning can be done at both the database and collection level in MongoDB. When sharding a database, the data is divided into multiple shards, each with its own replica set. The shards are then distributed across multiple physical servers, forming a sharded cluster. When sharding a collection, the data is divided into chunks, which are distributed across the shards based on the shard key. Each chunk is replicated across multiple servers in a shard to ensure high availability and data durability.

3. What is primary and secondary indexing in mongodb?

In MongoDB, indexes are used to improve the performance of queries. There are two types of indexes: primary and secondary indexes.

Primary index:
- Primary index is an index on the "_id" field of a collection, which is created by default when a collection is created.
- It is unique and ensures that no two documents in a collection have the same "_id" value.
- Primary index is used for efficient retrieval of a document by "_id" field, which is frequently used as a query predicate.

Secondary index:
- Secondary index is an index on a field other than "_id" in a collection.
- It allows for efficient retrieval of documents by the indexed field, which can improve query performance.
- Unlike the primary index, a secondary index can have duplicate values.
- MongoDB supports several types of secondary indexes, including single-field indexes, compound indexes, multikey indexes, text indexes, and geospatial indexes.

Here's an example of creating a secondary index in MongoDB using the "createIndex" method:

```
db.users.createIndex({ "email": 1 })
```

This creates a single-field index on the "email" field of the "users" collection.

4. How to track relation in nosql db?

In NoSQL databases, there are different ways to track relationships between data. Here are a few approaches:

- Embedded documents: In this approach, data from multiple entities are stored together in a single document. For example, if you have a user and their order history, you could store the user's information and their order history as embedded documents within a single user document. This approach is commonly used in document-oriented databases like MongoDB.
- Reference-based relationships: In this approach, each entity is stored in a separate document, but the documents contain references to each other. For example, a user document could contain a reference to an order document by including the order's ID. This approach is commonly used in graph databases like Neo4j.
- Denormalization: In this approach, data from related entities are duplicated across multiple documents to avoid expensive joins. For example, if you have a blog post and comments, you could store the comments' information alongside the blog post information to avoid having to join the two tables when fetching data. This approach is commonly used in column-family stores like Cassandra.

There are also hybrid approaches that combine the above methods to suit specific use cases. Choosing the right approach depends on your specific application's requirements and performance needs.

5. What is cursor in database?

In the context of databases, a cursor is a pointer or a mechanism that allows applications to iterate over a set of results returned by a database query. When a query is executed, the result set is created, and a cursor is automatically associated with it. The cursor contains information about the current position in the result set, such as the current row, and allows the application to move forward or backward through the result set.

A cursor can be used to fetch one or more rows at a time from the result set, which can be useful when processing large amounts of data. Cursors can be forward-only, which means that they can only be moved forward through the result set, or they can be scrollable, which allows them to be moved both forward and backward.

However, the use of cursors can have performance implications, particularly when dealing with large result sets. Cursors typically require additional resources to be allocated on the database server and can result in additional network traffic between the database server and the application. Therefore, it is generally recommended to avoid using cursors unless necessary, and instead rely on set-based operations to process data whenever possible.

Here is an example of using a cursor in SQL Server to retrieve and display data from a table:

```
DECLARE @employee_id INT, @employee_name VARCHAR(50)

DECLARE employee_cursor CURSOR FOR
SELECT employee_id, employee_name
FROM employees

OPEN employee_cursor

FETCH NEXT FROM employee_cursor INTO @employee_id, @employee_name

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Employee ID: ' + CONVERT(VARCHAR(10), @employee_id) + ', Employee Name: ' + @employee_name

    FETCH NEXT FROM employee_cursor INTO @employee_id, @employee_name
END

CLOSE employee_cursor
DEALLOCATE employee_cursor
```

In this example, we declare a cursor employee_cursor and use it to select all employee IDs and names from a table called employees. We then open the cursor, fetch the first row of data into variables @employee_id and @employee_name, and use a WHILE loop to iterate through all rows in the cursor. Within the loop, we print out the employee ID and name, and fetch the next row of data until there is no more data to fetch. Finally, we close and deallocate the cursor.

6. What is the difference between truncate and delete?

Both TRUNCATE and DELETE are SQL commands used to remove data from a database table. However, there are some key differences between the two:

- TRUNCATE is a DDL (Data Definition Language) command, while DELETE is a DML (Data Manipulation Language) command. TRUNCATE is used to remove all the data from a table, while DELETE is used to remove selected rows from a table.

- TRUNCATE is faster than DELETE. When you use TRUNCATE, the database system deallocates the data pages and frees the space back to the system immediately, while DELETE only marks the data as deleted and keeps the space occupied until a later time.
- TRUNCATE cannot be rolled back, while DELETE can be rolled back.
- TRUNCATE resets the identity of the table, while DELETE does not.
- TRUNCATE cannot be used with a WHERE clause, while DELETE can be used with a WHERE clause to select specific rows to delete.

In summary, if you want to remove all the data from a table quickly, use TRUNCATE. If you want to remove selected rows or only a portion of the data, use DELETE.

7. What is ReadWrite lock?

A ReadWrite lock is a synchronization mechanism used in multithreaded programming to allow multiple threads to access a shared resource at the same time. It consists of two types of locks: a read lock and a write lock.

The read lock allows multiple threads to acquire it simultaneously, and it only blocks when a thread attempts to acquire a write lock. This means that multiple threads can read the shared resource concurrently, but only one thread can write to it at a time.

The write lock, on the other hand, only allows one thread to acquire it at a time, and it blocks all other threads (both read and write) until it is released. This means that only one thread can modify the shared resource at any given time, and all other threads must wait until the write lock is released.

The ReadWrite lock provides better performance and scalability than a simple synchronized block because it allows multiple threads to read the shared resource at the same time. This is particularly useful in situations where the shared resource is read more often than it is written to.

Example usage of ReadWrite lock in Java:

```java
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class SharedResource {
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private int value;

    public void setValue(int newValue) {
        lock.writeLock().lock();
        try {
            this.value = newValue;
        } finally {
            lock.writeLock().unlock();
        }
    }

    public int getValue() {
        lock.readLock().lock();
        try {
            return this.value;
        } finally {
            lock.readLock().unlock();
        }
    }
}
```

In this example, the SharedResource class has a private field value that can be read and written by multiple threads. The setValue() method acquires the write lock before modifying the value, while the getValue() method acquires the read lock before returning the current value. This ensures that multiple threads can read the value concurrently, while only one thread can modify it at any given time.

8. What are disadvantages of having a separate database per service?

While there are some benefits to having a separate database per service in a microservices architecture, there are also some potential disadvantages, including:

- Increased complexity: Managing multiple databases can be more complex and time-consuming, particularly when it comes to ensuring data consistency across different services.
- Higher resource usage: Running multiple databases can consume more resources, particularly if each database needs to be replicated or backed up separately.

- Data duplication: Different services may need to store similar data, which can result in data duplication across multiple databases. This can make it harder to manage data consistency and increase storage requirements.
- Data consistency: Since each service has its own database, ensuring data consistency across different services can be challenging. It may require additional effort to synchronize data between services, particularly if services need to work with data from other services.
- Limited scalability: Having a separate database for each service can limit scalability, particularly if each database needs to be hosted on a separate server. This can make it harder to scale individual services independently, which is one of the key benefits of microservices architecture.

Overall, while a separate database per service can be a useful approach in some cases, it's important to carefully consider the tradeoffs and potential challenges before implementing this approach.

9. What is normalization?

Normalization is a process of organizing data in a database so that data is stored in a structured and efficient manner. It involves breaking down a larger table into smaller, more manageable tables and establishing relationships between them.

Normalization is essential for maintaining data integrity and consistency, and helps to prevent data redundancy and inconsistencies. It also ensures that data is stored in the most efficient way possible, making it easier to access and manipulate.

Normalization is typically achieved through a series of normalization steps, each of which involves analyzing the data and breaking it down into smaller tables based on common attributes or relationships between data elements. The end result is a database that is well-organized, efficient, and easy to work with.

Normalization is a process of organizing data in a database to minimize redundancy and dependency. It is divided into multiple levels, called normal forms, each of which has specific requirements for the structure of the database. The following are the different types of normalization:

- First Normal Form (1NF): It requires the database table to have a primary key and a single value in each column.

Example: Suppose we have a table called "employee" that has two columns - "employee_id" and "employee_skills". In the first normal form, we would split the second column into separate columns for each skill, such as "skill1", "skill2", and so on.

- Second Normal Form (2NF): It requires the table to be in the first normal form and also have no partial dependencies, meaning that all non-key columns must be fully dependent on the primary key.

Example: Suppose we have a table called "orders" that has columns "order_id", "product_id", "product_name", and "product_price". To convert it into second normal form, we would create a new table for products with columns "product_id", "product_name", and "product_price", and reference it from the "orders" table using only "product_id".

- Third Normal Form (3NF): It requires the table to be in the second normal form and also have no transitive dependencies, meaning that non-key columns cannot depend on other non-key columns.

Example: Suppose we have a table called "customers" that has columns "customer_id", "customer_name", "customer_address", and "customer_city". To convert it into third normal form, we would create a new table for cities with columns "city_id" and "city_name", and reference it from the "customers" table using only "customer_id" and "city_id".

- Boyce-Codd Normal Form (BCNF): It requires the table to be in the third normal form and also have no non-trivial dependencies, meaning that every determinant must be a candidate key.

Example: Suppose we have a table called "sales" that has columns "sales_id", "product_id", "product_name", "customer_id", and "customer_name". To convert it into Boyce-Codd normal form, we would split it into three tables: "sales" with columns "sales_id", "product_id", and "customer_id", "products" with columns "product_id" and "product_name", and "customers" with columns "customer_id" and "customer_name".

- Fourth Normal Form (4NF): It requires the table to be in the Boyce-Codd normal form and also have no multi-valued dependencies, meaning that a non-key column cannot have more than one value for each combination of values in the key columns.

Example: Suppose we have a table called "students" that has columns "student_id", "student_name", and "course_ids". To convert it into fourth normal form, we would create a new table for courses with columns "course_id" and "course_name", and reference it from the "students" table using only "student_id" and "course_id".

10. What are ACID properties?

ACID (Atomicity, Consistency, Isolation, Durability) properties are a set of fundamental properties that ensure the reliability of transactions in a database management system (DBMS).

- Atomicity: Atomicity guarantees that a transaction is treated as a single, indivisible operation. This means that either all the operations in a transaction must complete successfully, or none of them should execute at all.
- Consistency: Consistency ensures that a transaction brings the database from one valid state to another. It ensures that the data is in a consistent state before and after the transaction execution.
- Isolation: Isolation ensures that a transaction executes in a self-contained manner and is not affected by any other transaction executing concurrently. This means that a transaction can execute as if it is the only transaction running in the database system, even if there are other transactions executing simultaneously.
- Durability: Durability ensures that once a transaction has been committed, it will remain in the system even in the event of a system failure. The changes made by the transaction are permanently recorded in the database, and subsequent system failures do not impact these changes.

Together, these properties help to ensure that transactions are reliable, consistent, and recoverable in the face of various failures.

11. How to handle exceptions from the database if the actual data is not equal to expected?

When working with databases, it is important to handle exceptions appropriately to ensure that data is processed correctly. One way to handle exceptions from the database if the actual data is not equal to expected is to use the try-catch block to catch any SQL exceptions that may occur.

Here is an example of how to handle exceptions in Java using a try-catch block:

```
try {
    // execute database query and retrieve data
} catch (SQLException e) {
    // handle database exception
    System.err.println("SQL Exception: " + e.getMessage());
}
```

In this example, the try block contains the database query and data retrieval code. If an exception occurs during the execution of this code, the catch block will catch the SQL exception and handle it appropriately. The SQLException object provides information about the error that occurred, such as the error message, error code, and SQL state.

Once you have caught the exception, you can handle it in a way that is appropriate for your application. This may involve logging the error, retrying the database operation, or rolling back a transaction.

It is also a good practice to handle exceptions at the appropriate level of abstraction in your application. For example, if you are working with a service layer, you should catch exceptions at that level and translate them into meaningful exceptions for the calling client. This helps to decouple the database implementation from the rest of the application and promotes modularity and maintainability.

12. What are the datatypes in SQL?

SQL has several datatypes, some of the commonly used ones are:

- Numeric Datatypes: Used for representing numeric values. Examples include INT, BIGINT, FLOAT, DOUBLE, DECIMAL, etc.
- Character String Datatypes: Used for representing character strings. Examples include CHAR, VARCHAR, TEXT, etc.
- Date/Time Datatypes: Used for representing date and time values. Examples include DATE, TIME, DATETIME, TIMESTAMP, etc.
- Boolean Datatype: Used for representing boolean values. Example includes BOOLEAN.
- Binary Datatypes: Used for representing binary data. Examples include BLOB, BYTEA, etc.
- Other Datatypes: There are also other datatypes like ENUM, SET, JSON, XML, etc., which are used for representing specific types of data.

The specific datatypes available may vary depending on the database system being used.

13. What is Named Query?

A named query in a database is a pre-defined SQL statement that has been assigned a name and saved for later use. Instead of writing the same SQL query over and over again, you can create a named query and refer to it by name whenever you need to execute that particular query.

Named queries can help simplify database development by making frequently used queries easier to manage and less error-prone. For example, you might use a named query to retrieve all customers who have made a purchase in the last 30 days, or to retrieve a list of products with a certain price range.

In addition to simplifying the process of writing SQL queries, named queries can also help improve performance by allowing the database to cache frequently used queries. This can reduce the amount of time it takes to execute a query, since the database doesn't have to compile and optimize the query every time it's executed.

Named queries are often used in object-relational mapping (ORM) frameworks like Hibernate, where they can be defined in the mapping configuration file or annotated directly in the Java code.

Suppose we have an Employee entity and we want to create a named query to retrieve all employees with a salary greater than a certain amount. We can define the named query in the Employee entity class as follows:

```
@Entity
@Table(name = "employees")
@NamedQuery(name = "Employee.findHighlyPaid",
            query = "SELECT e FROM Employee e WHERE e.salary > :salary")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "salary")
    private Double salary;

    // getters and setters
}
```

In this example, we have defined a named query called "`Employee.findHighlyPaid`" that selects all employees from the "`employees`" table where the salary is greater than a parameter named ":`salary`". The "`:salary`" parameter is a placeholder for the actual salary value that we will provide when we execute the query.

To execute the named query in our Java code, we can use the EntityManager object provided by the Hibernate JPA implementation:

```
TypedQuery<Employee> query = entityManager.createNamedQuery("Employee.findHighlyPaid", Employee.class);
query.setParameter("salary", 50000.0);
List<Employee> employees = query.getResultList();
```

In this example, we are creating a `TypedQuery` object from the named query "`Employee.findHighlyPaid`" and setting the value of the ":`salary`" parameter to 50000.0. We then execute the query using `getResultList()` and retrieve a List of Employee objects that match the query criteria.

14. Why do we need NoSQL DB?

NoSQL (Not Only SQL) databases have become increasingly popular in recent years due to their ability to handle large volumes of unstructured and semi-structured data more efficiently than traditional relational databases. Here are some reasons why we need NoSQL databases:

- Scalability: NoSQL databases are designed to scale horizontally, which means they can handle large volumes of data and traffic by adding more nodes to the cluster. This makes them suitable for modern web applications that need to handle millions of users and a high volume of data.
- Performance: NoSQL databases can offer better performance than traditional relational databases in certain use cases. They are designed to optimize read and write operations for large volumes of data, which can result in faster query response times.
- Flexibility: NoSQL databases are more flexible than relational databases when it comes to data modeling. They allow developers to store data in a variety of formats, including key-value, document, graph, and column-family. This makes them ideal for handling unstructured and semi-structured data that may not fit well into a rigid schema.
- Cost-effective: NoSQL databases can be more cost-effective than traditional relational databases because they are often open-source and can be run on commodity hardware. They also require less maintenance and administration than relational databases.

- Availability: NoSQL databases are designed to be highly available, meaning they can provide uninterrupted service even if some nodes in the cluster fail. This makes them suitable for applications that require high availability and fault tolerance.

Overall, NoSQL databases provide a more flexible, scalable, and cost-effective approach to managing large volumes of unstructured and semi-structured data, making them ideal for modern web applications and other use cases that require handling large volumes of data.

15. What are several SQL Exceptions in Spring Boot?

Spring Boot provides a number of built-in SQL exceptions that can occur when working with databases. Here are some common SQL exceptions and their possible causes:

- DataAccessException: This is a generic exception that can be thrown when there is a problem accessing data from a database. It can be caused by a variety of issues such as connection problems, database server issues, or syntax errors in SQL queries.
- DataIntegrityViolationException: This exception is thrown when there is a problem with data integrity in the database. For example, if you try to insert a record with a primary key that already exists, this exception will be thrown.
- DuplicateKeyException: This exception is thrown when you try to insert a record with a primary key that already exists in the database.
- IncorrectResultSizeDataAccessException: This exception is thrown when a query returns an incorrect number of rows. For example, if you expect a query to return a single row but it returns multiple rows or no rows, this exception will be thrown.
- CannotAcquireLockException: This exception is thrown when a transaction fails to acquire a lock on a database resource. This can happen when multiple transactions are trying to access the same resource simultaneously.
- DeadlockLoserDataAccessException: This exception is thrown when a transaction fails due to a deadlock. A deadlock occurs when two or more transactions are waiting for each other to release a lock on a database resource.

These are just a few examples of the SQL exceptions that can occur in Spring Boot applications. The specific causes of these exceptions can vary depending on the database being used, the configuration of the database and the queries being executed. It's important to understand the possible causes of these exceptions so that you can identify and fix issues when they occur.

Angular Interview Questions:

1. What is Angular?

Angular is a component-based framework for building structured, scalable, single-page web applications for client-side.

2. What are the advantages of Angular?
    a. With Angular, it is relatively simple to build single-page applications
    b. To make use of flexible and structured applications(OOPs friendly)
    c. It is cross-platform and open source
    d. It provides services through which we can reuse code
    e. It provides testing

3. What is the difference between Angular and AngularJS?

Angular	AngularJS
a. It supports both JavaScript and TypeScript b. It has a component-based Architecture c. It has a CLI tool d. It used dependency injection e. It supports mobile browsers f. It is very fast	a. It only supports JavaScript b. It has a Model View Controller(MVC) Architecture c. It does not have a CLI tool d. It does not use dependency injection e. It does not support mobile browsers f. It is not so fast

4. What is NPM?

NPM(Node Package Manager) is an online repository from where we can get thousands of free libraries which we can use in our Angular project.
The node_modules folder is used to save all downloaded packages from npm in our computer.

5. What is a CLI tool?

CLI is a Command Line Interface tool which can be used to initialize and develop Angular applications.

6. What are components in Angular?

Components are the most basic building block of an Angular app. The CSS, HTML, Typescript and Spec files together make one component.

7. What are Selector and Template?

- A Selector is used to identify each component uniquely in the component tree.
- A Template is an HTML view of an Angular component.

8. What is a Module in Angular? What is a module.ts file?

A module is a place where you can group the components, directives, pipes, and services, which are related to the application.

9. How does an Angular App gets Loaded and Started? What are index.html, app-root, selector and main.ts?

- Angular is used to create Single Page Applications, the index.html file is that single page. Index.html will invoke the main.js file which is the javascript version of the main.ts file.
- main.ts file is like the entry point of the web app. It compiles the web app and bootstraps the AppModule to run in the browser.
- AppModule file will then bootstrap the AppComponent.
- AppComponent or AppRootComponent has the HTML file reference which we will see finally in the browser.

10. What are a Bootstrapped Module and Bootstrapped Component?

When the Angular web application will start then the first module launched is the bootstrapped module and the same is true for the bootstrapped component also. A bootstrapped component is an entry component that Angular loads into the DOM during the bootstrap process or application launch time.

11. What is Data Binding in Angular?

Data binding is the way to communicate between the typescript code of the component and its HTML view.

12. What is String Interpolation in Angular?

String Interpolation is a one-way data-binding technique that is used to transfer the data from a TypeScript code(component) to an HTML template (view).

a. String interpolation can work on string type only not numbers or any other type.
b. It is represented inside {{data}} double curly braces.

```
<!-- String Interpolation -->
<div>{{title}}</div>
```

13. What is Property Binding in Angular?

Property binding is a superset of interpolation. It can do whatever interpolation can do. In addition, it can set an element property to a non-string data value like Boolean.

```
<!-- Property Binding -->
<div [innerText]='title'></div>
<input type="text" [disabled]='isDisabled'>
```

14. What is Event Binding in Angular?

Event binding is used to handle the events raised by the user
actions like button click.

```
<!-- Event Binding -->
```

```
<button (click)="clickFunction()">Click Interview</button>
```

15. What is Two Way Data Binding in Angular?
16. What is a JWT token and what are the advantages of using it?
17. How to handle JWT tokens in Angular?
18. How to implement autocomplete in the frontend?
19. What are the different methods to send data between parent and child components in Angular?
20. What are methods to send data between components in Angular?
21. What are the differences between BehaviorSubject and Observable?
22. What is ViewChild?
23. What are Angular Lifecycles?
24. What are differences between data binding and class binding?
25. What is interpolation in Angular?
26. What are the different types of services in Angular?
27. How will you compare two javascript objects?
28. What are the differences between window and document in javascript?
29. What are the differences between a Class component and Functional component?
30. What is callback hell and what is the main cause of it?
31. What are different types of Lifecycle hooks in Angular?
32. Difference between var and let?
33. How to use async pipe?
34. How to transfer data between components?
35. How to transfer data between modules?
36. Difference between promise and observable?
37. What are Angular fundamental and advance concepts?
38. How is error and exception handling is done in Angular? Give example with code.
39. How is login management and session management done in Angular? Give example with code.
40. What are Angular's feature? What is the advantages and disadvantages of using Angular?
41. Describe Angular's architecture.
42. What are Typescript fundamental and advance concepts?
43. What are functions and features of Typescript?
44. What are Javascript fundamental, object oriented and advance concepts?
45. What is responsiveness?
46. What is LESS and SASS?
47. What is positioning and display in CSS?
48. What is grid system in CSS?
49. What are pseudo class and pseudo element in CSS?
50. What are HTML5 API, tags, elements?
51. What is progressive web?
52. What is prototyping in JavaScript?
53. What is event loop?
54. What is callback?

- **General Interview questions:**

1. PAYTM: There is a huge file which needs to be processed. Around 20 MB. How to process?
   I told that let's say 10 lakh lines are there in the file. Read 50k / 1 lakh lines at a time and push those lines to a blocking queue. Then configure some fixed number of threads who will consume from the blocking queue and will procss them parallelly. The no of threads will depend on whether it's a CPU Intensive task(Go for no of available processors) or I/O intensive task(No of threads can be greater).

2. PAYTM: Maker-Checker validation: Frontend is the maker. Backend is the checker. Backend must validate data before storing into DB.
   a. First I told, doing RequestBody validation in the POST call using @Valid annotation and @Pattern(regexp="")
   b. Second: Implementing Proxy design pattern where proxy will be the checker who will validate it. And then it'll call the real one which will persist it into DB.
   c. Third: Interviewer expected multiple checkers and each checker may not be able to validate a particular Maker. Told Chain of Responsibility design pattern, where 1 checker will check and do validation of it's part. Then it'll delegate to the next checker.

3. Morgan Stanley: Multiple threads are sending duplicate trades. Deduplicate them and store in DB
   a. Set<T> set=new CopyOnWriteHashSet<>()
   b. Interviewer told me this wastes memory. Told to use ConcurrentHashMap which doesn't and Set also internally uses HashMap.

No broker

dictionary is given character stream is coming each stream has suffix in dictionary? -> suffix
wordList = [xyz, abc, j, jx, ax]

Stream - axhjxyz
            0101101
Morgan Stanley coding question: Smallest missing number
{4,5,7,8,9,10,12,13} -> 6
“String” => “stringstring”
“Gstrin” -> boolean