|  |  |  |
| --- | --- | --- |
| **Feature** | **Comparable** | **Comparator** |
| Implementation | Implemented by the class of objects to sort | Implemented by a separate class |
| Purpose | Defines natural ordering | Defines custom ordering |
| Method | compareTo(),it takes one parameter  compareTo(Object o) | compare(),it takes two parameter  compare(Object o1, Object o2) |
| Package | java.lang | java.util |
| Use Cases  qa | Single sorting order, inherent to the object | Multiple sorting orders, external comparison |

Comparable and Comparator:

Comparable and Comparator are both interfaces used for sorting objects.

HashMap and HashTable :

HashMap and HashTable are both implementations of the Map interface, used to store key-value pairs. The key difference lies in their thread-safety: HashTable is synchronized and thread-safe, while HashMap is not, offering better performance in single-threaded environments. Additionally, HashMap allows one null key and multiple null values, whereas HashTable does not allow null keys or values.

What is the purpose of the `finalize()` method?

It is a method in object super class called by garbage collector before an o bject is deleted.

It allows an object to perform cleanup operation before being removed from memory.

The purpose of finalise method is to release resources that is allocated by unused object, before removing unused objects by garbage collector.

**Finalization**

Before destroying an object, the garbage collector calls the [finalize()](https://www.geeksforgeeks.org/finalize-method-in-java-and-how-to-override-it/)method to perform cleanup activities. The method is defined in the [Object class](https://www.geeksforgeeks.org/object-class-in-java/)as follows:

*@Override*

*protected void finalize() throws Throwable {*

*System.out.println("GC cleaning up...");*

*}*

**Note:**

* The finalize() method is called by Garbage Collector, not JVM.
* The default implementation of finalize() is empty, so overriding it is recommended for resource cleanup.
* The finalize() method is called only once per object.
* If an uncaught exception is thrown by the finalize() method, the exception is ignored, and the finalization of that object terminates.

**Syntax:**

protected void finalize throws Throwable{}

Garbage collection:

In java, garbage collector means unreferenced objects.  
It is the process of reclaiming runtime unused memory automatically.IN other words, It is a way to destroy unused objects.

How to work Garbage Collection?  
The GC of JVM collects only those objects that are created by new keyword. So, if you have created new object without new keyword , you can use finalize method to perform cleanup processing.

 The Java Virtual Machine (JVM) automatically handles this process.

**Finalization**

Before destroying an object, the garbage collector calls the [finalize()](https://www.geeksforgeeks.org/finalize-method-in-java-and-how-to-override-it/)method to perform cleanup activities. The method is defined in the [Object class](https://www.geeksforgeeks.org/object-class-in-java/)as follows:

*@Override*

*protected void finalize() throws Throwable {*

*System.out.println("GC cleaning up...");*

*}*

**Note:**

* The finalize() method is called by Garbage Collector, not JVM.
* The default implementation of finalize() is empty, so overriding it is recommended for resource cleanup.
* The finalize() method is called only once per object.
* If an uncaught exception is thrown by the finalize() method, the exception is ignored, and the finalization of that object terminates.

Difference between wait and sleep :

| **Wait()** | **Sleep()** |
| --- | --- |
| Wait() method belongs to Object class. | Sleep() method belongs to Thread class. |
| Wait() method releases lock during Synchronization. | Sleep() method does not release the lock on object during Synchronization. |
| Wait() should be called only from Synchronized context. | There is no need to call sleep() from Synchronized context. |
| Wait() is not a static method. | Sleep() is a static method. |
| Wait() Has Three Overloaded Methods:   * wait() * wait(long timeout) * wait(long timeout, int nanos) | Sleep() Has Two Overloaded Methods:   * sleep(long millis)millis: milliseconds * sleep(long millis,int nanos) nanos: Nanoseconds |
| public final void wait(long timeout) | public static void sleep(long millis) throws Interrupted\_Execption |

**What is IoC?**

IoC is a design principle in which the control of object creation and management is transferred from the application code to a container or framework. In simpler terms, instead of the application itself creating and managing its dependencies, the framework does this for you.

In simple words, the control of creating objects and managing the spring components is taken care of by the Spring containers.

**How does IoC work in Spring?**

In Spring, IoC is primarily implemented through **Dependency Injection (DI)**. Here’s how it works:

1. **Bean Creation**: Spring creates objects (beans) and manages their lifecycle. These beans are defined in configuration files (XML) or annotated classes.

**Beans**: Components (classes) are marked with annotations like @Component, @Service, @Repository, or @Controller. These are automatically detected and registered as **beans** in the Spring container.

1. **Dependency Injection**: **Dependency Injection**: Spring injects dependencies into beans using:
   * **Constructor Injection**: Dependencies are provided through the constructor.
   * **Setter Injection**: Dependencies are provided through setter methods.
   * **Field Injection**: Dependencies are injected directly into fields (though this is less common and generally not recommended due to issues with testability and immutability).  (not recommended for testing)
2. **Configuration**: Spring Boot uses @SpringBootApplication, which includes @ComponentScan and @Configuration, to automatically scan and configure beans.

**Benefits of IoC**

1. **Loose Coupling**: By delegating the creation and management of dependencies to the framework, classes are less dependent on each other. This makes the code more modular and easier to maintain.
2. **Enhanced Testability**: Since dependencies are injected, it’s easier to replace real dependencies with mock objects during testing.
3. **Configuration Flexibility**: You can change the configuration without altering the code. For example, switching from one implementation of a service to another can be done by changing the configuration.

**Example**

Here’s a simple example to illustrate IoC with Spring:

// Service interface

public interface GreetingService {

    String greet(String name);

}

// Service implementation

public class GreetingServiceImpl implements GreetingService {

    @Override

    public String greet(String name) {

        return "Hello, " + name;

    }

}

// Client class

public class Client {

    private GreetingService greetingService;

    // Constructor injection

    public Client(GreetingService greetingService) {

        this.greetingService = greetingService;

    }

    public void showGreeting(String name) {

        System.out.println(greetingService.greet(name));

    }

}

// Spring configuration (XML)

    <bean id="greetingservice" class="com.example.greetingserviceimpl"></bean id="greetingservice" class="com.example.greetingserviceimpl">

    <bean id="client" class="com.example.client"></bean id="client" class="com.example.client">

        <constructor-arg ref="greetingservice"></constructor-arg ref="greetingservice">

In this example:

* The GreetingServiceImpl bean is created and managed by Spring.
* The Client bean is also created by Spring, and its dependency (GreetingService) is injected via the constructor.

### **Example:**

// Service class

* @Service
* public class MyService {
* public String serve() {
* return "Service is working!";
* }
* }
* // Controller class
* @RestController
* public class MyController {
* private final MyService myService;
* // Constructor Injection
* public MyController(MyService myService) {
* this.myService = myService;
* }
* @GetMapping("/test")
* public String test() {
* return myService.serve();
* }
* }

In this example:

* Spring Boot automatically creates an instance of MyService.
* It injects it into MyController via the constructor.
* You didn’t manually create or manage these objects — **Spring did**.

* **2. What is Dependency Injection, and how is it implemented in Spring?**

**Dependency Injection (DI)** is a design pattern used to implement Inversion of Control (IoC), allowing the creation of dependent objects outside of a class and providing those objects to a class in various ways. This helps in making the code more modular, testable, and maintainable.

**How Dependency Injection Works:**

1. **Inversion of Control (IoC)**: The control of object creation and binding is transferred from the class itself to an external entity.
2. **Dependency Injection**: Dependencies are injected into a class, rather than the class creating them itself.

**Types of Dependency Injection:**

1. **Constructor Injection**: Dependencies are provided through a class constructor.
2. **Setter Injection**: Dependencies are provided through setter methods.
3. **Field Injection**: Dependencies are injected directly into fields.

**Implementation in Spring:**

Spring Framework provides comprehensive support for DI, primarily through the use of IoC containers. Here’s how it’s typically implemented:

1. **Configuration**:
   * **XML Configuration**: Define beans and their dependencies in an XML file.
   * **Java Configuration**: Use @Configuration and @Bean annotations to define beans.
   * **Annotation-based Configuration**: Use annotations like @Component, @Autowired, @Service, etc.
2. **Bean Definition**:
   * **XML Example**:
   * **Java Example**:
   * **Annotation Example**:

**Benefits of Dependency Injection:**

* **Decoupling**: Reduces the tight coupling between components.
* **Testability**: Makes unit testing easier by allowing mock dependencies.
* **Maintainability**: Simplifies the management of dependencies.

1. **Describe the difference between Spring Singleton and Prototype scopes.**

**Key Differences:**

* Instance Creation: Singleton creates one instance per container, while Prototype creates a new instance each time it is requested.
* Lifecycle Management: Singleton beans are managed by the container throughout their lifecycle, whereas Prototype beans are managed only during creation.
* Memory Usage: Singleton beans can be more memory-efficient since only one instance is created, while Prototype beans can consume more memory due to multiple instances**.**
* **<bean id="singletonbean" class="com.example.singletonbean" scope="singleton"/>**

**Annotation based:**

* **@Bean**
* **@Scope("singleton")**
* **public SingletonBean singletonBean() {**
* **return new SingletonBean();**
* **}**
* <bean id="prototypebean" class="com.example.prototypebean" scope="prototype"/>
* **Annotation based:**
* @Bean
* @Scope("prototype")
* public PrototypeBean prototypeBean() {
* return new PrototypeBean();
* }

1. **What is the purpose of the BeanFactory and ApplicationContext in Spring?**

In Spring, BeanFactory and ApplicationContext are two key interfaces used for managing beans and their dependencies. They both serve as IoC containers but have different features and use cases.

BeanFactory:

Purpose: The BeanFactory is the simplest container providing basic DI capabilities. It is responsible for instantiating, configuring, and managing beans.

The most common implementation class used for this BeanFactory is **XmlBeanFactory** available in **org.springframework.beans.factory.xml package.**

Features:

Lazy Initialization: Beans are created only when they are requested.

Lightweight: Suitable for lightweight applications or scenarios where memory consumption is a concern.

Usage: Typically used in scenarios where only basic DI features are needed without the overhead of additional functionalities.

ApplicationContext:

Purpose: The ApplicationContext is a more advanced container that builds on the BeanFactory, providing additional enterprise-specific functionalities.

There are so many implementation classes that can be used such as **ClassPathXmlApplicationContext**, **FileSystemXmlApplicationContext**, **AnnotationConfigWebApplicationContext** etc.

**Features:**

Eager Initialization: Beans are created at startup, which can lead to faster performance at runtime.

Internationalization: Supports message resource handling for internationalization (i18n).

Event Propagation: Supports event propagation and handling.

AOP Integration: Provides integration with Spring's Aspect-Oriented Programming (AOP) features.

Web Application Context: Specialized for web applications, providing features like web-scoped beans.

Usage: Preferred for most enterprise applications due to its rich feature set.

**Key Differences:**

Initialization: BeanFactory uses lazy initialization, while ApplicationContext uses eager initialization by default.

Features: ApplicationContext provides more advanced features like event handling, AOP, and internationalization, which are not available in BeanFactory.

Usage Context: BeanFactory is suitable for lightweight applications, whereas ApplicationContext is ideal for enterprise-level applications with complex requirements.

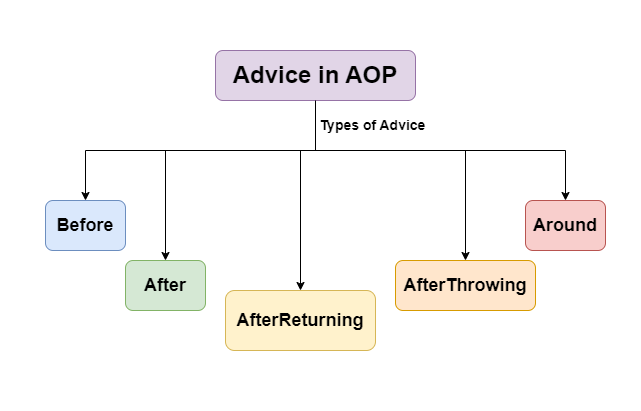
|  |  |  |
| --- | --- | --- |
| **Bean Scopes Supported**  **Annotation Support** | Supports only Singleton and Prototype bean scopes.  Does not support annotations; requires configuration in XML files. | Supports all types of bean scopes, including Singleton, Prototype, Request, Session, etc  Supports annotation-based configuration for bean autowiring. |

1. **Explain the AOP (Aspect-Oriented Programming) concept in the context of Spring.**

**Aspect-Oriented Programming (AOP) in Spring Boot** is a powerful feature that enhances modularity by handling cross-cutting concerns such as **logging, security**, and **transaction management** separately from business logic.

**Understanding AOP Concepts**

* **Aspect**: An Aspect is a modular unit of cross-cutting concerns. For example, a logging aspect can be applied across various methods in different classes..
* **Advice:** This is the action taken by an aspect at a particular join point. There are five types of advice:
  + **Before**: Executed before the method call.
  + **After**: Executed after the method call, regardless of its outcome.
  + **AfterReturning**: Executed after the method returns a result, but not if an exception occurs.
  + **Around:** Surrounds the method execution, allowing you to control the method execution and its result.
  + **AfterThrowing**: Executed if the method throws an exception.



* **Join Point:** A specific point in the execution of a program, such as method execution or exception handling, where an aspect can be applied.
* **Pointcut**: A Pointcut is a predicate that defines where advice should be applied. It matches join points using expressions.

**Example: Implementing Logging with AOP in Spring**

package com.example.aspect;

@Aspect // Marks the class as an aspect, which contains cross-cutting concerns

@Component // Registers this aspect as a Spring bean

public class LoggingAspect {

@Pointcut("execution(public void com.example.service.\*.\*(..))")

// Defines a pointcut that matches the execution of any public method in classes under com.example.service package

public void allServiceMethods() {}

@Before("allServiceMethods()")

// Advice that runs before the execution of methods matched by the pointcut

public void logBefore(JoinPoint joinPoint) {

System.out.println("Before method: " + joinPoint.getSignature().getName());

}

@After("allServiceMethods()")

// Advice that runs after the execution of methods matched by the pointcut, regardless of their outcome

public void logAfter(JoinPoint joinPoint) {

System.out.println("After method: " + joinPoint.getSignature().getName());

}

@AfterReturning(pointcut = "allServiceMethods()", returning = "result")

// Advice that runs after a method matched by the pointcut returns successfully

public void logAfterReturning(JoinPoint joinPoint, Object result) {

System.out.println("Method returned: " + result);

}

@AfterThrowing(pointcut = "allServiceMethods()", throwing = "error")

// Advice that runs if a method matched by the pointcut throws an exception

public void logAfterThrowing(JoinPoint joinPoint, Throwable error) {

System.out.println("Method threw exception: " + error);

}

@Around("allServiceMethods()")

// Advice that runs before and after the execution of methods matched by the pointcut

public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {

System.out.println("Before and after method: " + joinPoint.getSignature().getName());

return joinPoint.proceed(); // Proceed with the next advice or target method invocation

}

}

## Explain the role of the DispatcherServlet in Spring MVC.

The **DispatcherServlet** is a core component of the Spring MVC framework. It acts as the **front controller** in the MVC (Model-View-Controller) design pattern. Here’s a breakdown of its role:

1. **Request Handling**: When a request is made to a Spring MVC application, the DispatcherServlet receives it first. It acts as a central point for handling all incoming HTTP requests.
2. **Routing**: The DispatcherServlet uses handler mappings to determine which controller should handle the request. It maps the request URL to the appropriate controller method.
3. **Controller Execution**: Once the appropriate controller is identified, the DispatcherServlet invokes the corresponding method to process the request.
4. **View Resolution**: After the controller processes the request and returns a ModelAndView object, the DispatcherServlet uses view resolvers to determine which view (e.g., JSP, Thymeleaf) should be rendered.
5. **Response Generation**: Finally, the DispatcherServlet renders the view and sends the response back to the client.

In essence, the DispatcherServlet orchestrates the entire request processing workflow in a Spring MVC application, ensuring that requests are handled efficiently and responses are generated correctly.

## What are Spring annotations, and can you give examples of some commonly used ones?

### Spring annotations are metadata that provide information about the behavior of Spring components. They help configure and manage beans, define dependencies, and control the flow of the application without requiring extensive XML configuration. Here are some commonly used Spring annotations:

### **@Component**: Marks a Java class as a Spring component. It is a generic stereotype for any Spring-managed component.

### **@Service**: Specialization of @Component, indicating that the class is a service. It is used to mark business logic classes.

### **@Repository**: Specialization of @Component, indicating that the class is a repository. It is used for data access logic.

### **@Controller**: Marks a class as a Spring MVC controller. It is used to handle web requests.

### **@RestController**: Combination of @Controller and @ResponseBody. It is used to create RESTful web services.

### **@Autowired**: Used for automatic dependency injection. It can be applied to fields, setter methods, and constructors.

### **@Qualifier**: Used in conjunction with @Autowired to specify which bean should be injected when there are multiple beans of the same type.

### **@RequestMapping**: Used to map web requests to specific handler methods in a controller.

### **@GetMapping, @PostMapping, @PutMapping, @DeleteMapping**: Specialized versions of @RequestMapping for handling specific HTTP methods.

### **@Transactional**: Used to define the scope of a single database transaction.

Stereotype annotations in Spring Boot are used to mark classes with specific roles, allowing Spring to automatically manage them as beans within the application context. These annotations simplify configuration and enhance code readability by clearly defining the purpose of different components.

Here are some key stereotype annotations:

* **@Component**:

A generic annotation for any Spring-managed class. It signals that a class is a component that Spring should manage.

* **@Service**:

Indicates a class contains business logic, typically residing in the service layer.

* **@Repository**:

Used for classes that perform data access operations, such as interacting with a database.

* **@Controller**:

Marks a class as a web request handler, typically used for handling user requests in a web application.

* **@RestController**:

A specialized annotation that combines @Controller and @ResponseBody, used for RESTful web service controllers.

* **@Configuration**:

Indicates that a class contains bean definitions.

.**difference between requestmapping and getmapping**.:

**@RequestMapping**: Used to map web requests to specific handler methods in a controller while @GetMapping is specifically designed for handling HTTP GET requests. @GetMapping is a shortcut for @RequestMapping when you only need to specify that a method handles GET requests.

* @RequestMapping:
  + Can be used to map methods to multiple HTTP methods (GET, POST, PUT, DELETE, etc.) by specifying the method parameter.
  + Can be applied to both class and method levels.
  + Provides more flexibility in specifying request mapping criteria, such as headers, parameters, and media types.
  + Example: @RequestMapping(value="/users", method=RequestMethod.GET).
* @GetMapping:
  + A shortcut for @RequestMapping specifically for handling HTTP GET requests.
  + Only applies to HTTP GET requests.
  + Simplifies the syntax when handling GET requests, as you don't need to explicitly specify method=RequestMethod.GET.
  + Example: @GetMapping("/users").

**Spring Boot:-**

1. **What is Spring Boot, and how does it simplify the development of Spring applications?**

**Spring Boot** is a framework built on top of the Spring framework that simplifies the development of Spring applications. It provides a set of tools and conventions to make it easier to create stand-alone, production-ready Spring applications with minimal configuration. Here’s how Spring Boot simplifies development:

**Key Features and Benefits**

1. **Auto-Configuration**: Spring Boot automatically configures your application based on the dependencies you include. This reduces the need for manual configuration and boilerplate code.
2. **Starter POMs**: Spring Boot provides starter POMs (Project Object Models) that bundle commonly used dependencies into a single dependency. For example, spring-boot-starter-web includes all the necessary dependencies for building web applications.
3. **Embedded Servers**: Spring Boot includes embedded servers like Tomcat, Jetty, and Undertow, allowing you to run your application as a standalone executable JAR. This eliminates the need for external server deployment.
4. **Production-Ready Features**: Spring Boot provides built-in support for monitoring, metrics, health checks, and externalized configuration, making it easier to manage and deploy applications in production.
5. **Command-Line Interface (CLI)**: The Spring Boot CLI allows you to quickly prototype and run Spring applications using Groovy scripts.
6. **Spring Initializr**: Spring Boot offers the Spring Initializr, a web-based tool that helps you generate a Spring Boot project with the necessary dependencies and configurations.
   * You can access it at start.spring.io.
7. **Minimal Configuration**: Spring Boot reduces the need for extensive XML or Java-based configuration, allowing developers to focus on writing business logic.
8. **Rapid Prototyping**: With embedded servers and auto-configuration, developers can quickly prototype and test their applications.

**12. Explain the concept of "convention over configuration" in Spring Boot.**

The concept of **"convention over configuration"** in Spring Boot means that the framework provides sensible defaults for configuration, allowing developers to get started quickly without needing to specify every detail. This approach minimizes the amount of configuration needed to set up a Spring application, making development faster and easier.

**Key Aspects of Convention Over Configuration**

1. **Sensible Defaults**: Spring Boot comes with pre-configured settings that are commonly used in most applications. For example, it automatically configures a default embedded web server (like Tomcat) and sets up common dependencies based on the included starter POMs.
2. **Auto-Configuration**: Spring Boot's auto-configuration feature automatically configures beans and settings based on the classpath, property settings, and other factors. This means you don't have to manually configure each component unless you need to customize the default behavior.
3. **Starter Dependencies**: Spring Boot provides starter dependencies that bundle commonly used libraries and frameworks. For example, spring-boot-starter-data-jpa includes dependencies for Spring Data JPA, Hibernate, and a database connection pool. This reduces the need to manually specify each dependency.
4. **Minimal Configuration**: With convention over configuration, you only need to specify configuration settings when you want to override the defaults. This keeps the configuration files clean and concise.
5. **Embedded Server**: By including spring-boot-starter-web, Spring Boot automatically configures an embedded Tomcat server. You don't need to set up a server manually.
6. **Component Scanning**: The @SpringBootApplication annotation enables component scanning, so Spring Boot automatically detects and registers the MyController bean.

13. How does Spring Boot handle application properties, and what is the purpose of the `[application.properties](http://application.properties)` (or `application.yml`) file?

**These files allow you to externalize configuration, making it easier to manage and change settings without modifying the code.**

**Profiles**

**You can define environment-specific configurations using profiles. For example, you might have application-dev.properties for development and application-prod.properties for production. Activate a profile by setting the spring.profiles.active property.**

**# application.properties**

**spring.profiles.active=dev**

**# application-dev.properties**

**server.port=8081**

**# application-prod.properties**

**server.port=8080**

**Accessing Properties in Code**

**You can access these properties in your Spring Boot application using the @Value annotation or the Environment object.**

**@Value("${server.port}")**

**private int serverPort;**

**@Autowired**

**private Environment env;**

**public void printConfig() {**

**System.out.println("Server Port: " + env.getProperty("server.port"));**

**}**

| **Feature** | **application.properties** | **application.yml** |
| --- | --- | --- |
| **Format** | **Key-value pairs** | **YAML (hierarchical structure)** |
| **Readability** | **Simple for flat configurations** | **More readable for nested configurations** |
| **Complexity Handling** | **Better for simple configurations** | **Preferred for complex configurations** |
| **Syntax** | **key=value** | **Indentation-based** |
| **Example** | **server.port=8080** | **server:\n port: 8080** |
| **Nested Properties** | **Can be cumbersome** | **Easier to manage** |
| **Use Case** | **Simple applications** | **Complex applications** |
| **Personal Preference** | **Flat structure** | **Hierarchical structure** |

14. Describe the role of the `@SpringBootApplication` annotation in a Spring Boot application.

The @SpringBootApplication annotation is a key component in Spring Boot applications. It serves as a convenient shorthand for several other annotations and configurations. Here’s a detailed look at its role:

**Role of @SpringBootApplication**

1. **Combination of Annotations**:
   * **@Configuration**: Indicates that the class can be used by the Spring IoC container as a source of bean definitions.
   * **@EnableAutoConfiguration**: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. It enables the auto-configuration feature of Spring Boot.
   * **@ComponentScan**: Enables component scanning so that Spring can discover and register beans in the application context.
2. **Simplifies Configuration**:
   * By combining these three annotations, @SpringBootApplication reduces the need for multiple annotations and simplifies the configuration of the application.
3. **Auto-Configuration**:
   * The @EnableAutoConfiguration part of @SpringBootApplication automatically configures your Spring application based on the dependencies you have added. For example, if you have spring-boot-starter-web in your classpath, it will automatically configure a web server.
4. **Component Scanning**:
   * The @ComponentScan part of @SpringBootApplication enables scanning for Spring components, configurations, and services in the package where the application is located, allowing Spring to find and register beans.

difference between native query and named query:

In Java Persistence API (JPA), *@Query* and *NamedQuery* are both ways to execute database operations using JPQL (Java Persistence Query Language). *@Query* is an annotation used on repository methods to specify a JPQL or native SQL query. *NamedQuery*, on the other hand, is defined statically in the entity class and referenced by a unique name.

**Key Points**

1. *@Query* is used to define the query right above the method in the repository interface.

2. *NamedQuery* is defined at the entity level using the *@NamedQuery* annotation and is named for later use.

3. *@Query* can easily leverage Spring Data repository capabilities like pagination and sorting.

4. *NamedQuery* queries are statically checked at application startup, potentially catching errors early.

**3. Differences**

|  |  |
| --- | --- |
| **@Query/nativeQuery** | **NamedQuery** |
| Defined directly on the repository method. | Defined on the entity class and referenced by name. |
| Queries are not checked until the method is called. | Queries are checked at application startup. |
| Can use native SQL or JPQL.  Must need to enable nativeQuery = true | Typically uses JPQL and is part of JPA standard. |

**4. Example**

@Entity

@NamedQuery(name = "User.findByName", query = "SELECT u FROM User u WHERE u.name = :name")

public class User {

// ... entity fields, getters and setters

}

@Query(value = "SELECT \* FROM users WHERE status = ?1", nativeQuery = true)

List findByStatus(String status);

// Using NamedQuery by referencing the name

User findByName(@Param("name") String name);

}

15. Explain the difference between Spring Boot Starters and Spring Boot Auto-Configuration.

**Spring Boot Starters**

**Purpose**: Starters are a set of convenient dependency descriptors that you can include in your project. They bundle commonly used libraries and frameworks, making it easier to get started with specific functionalities.

**Key Features**:

1. **Convenience**: Starters provide a ready-made set of dependencies for common tasks, reducing the need to manually specify each dependency.
2. **Consistency**: Ensures that you have all the necessary dependencies for a particular functionality, avoiding version conflicts.
3. **Examples**:
4. spring-boot-starter-web: Includes dependencies for building web applications (Spring MVC, Tomcat, etc.).

* spring-boot-starter-data-jpa: Includes dependencies for JPA and Hibernate.

**Spring Boot Auto-Configuration**

**Purpose**: Auto-Configuration automatically configures your Spring application based on the dependencies you have added and the settings you have provided. It eliminates the need for manual configuration.

**Key Features**:

1. **Automatic Configuration**: Spring Boot scans the classpath and applies default configurations based on the included dependencies.
2. **Customization**: You can override the default configurations by providing your own settings in application.properties or application.yml.
3. **Examples**:
   * If spring-boot-starter-web is included, Spring Boot will automatically configure a web server (Tomcat) and set up Spring MVC.
   * If spring-boot-starter-data-jpa is included, Spring Boot will automatically configure a DataSource, EntityManagerFactory, and TransactionManager.

**Comparison Table**

| **Feature** | **Spring Boot Starters** | **Spring Boot Auto-Configuration** |
| --- | --- | --- |
| **Purpose** | Provide convenient dependency bundles | Automatically configure application |
| **Functionality** | Bundles common libraries and frameworks | Applies default configurations based on classpath |
| **Configuration** | Reduces manual dependency management | Reduces manual bean and settings configuration |
| **Customization** | Ensures necessary dependencies are included | Allows overriding default configurations |
| **Examples** | spring-boot-starter-web, spring-boot-starter-data-jpa | Auto-configures web server, JPA settings |

**How They Work Together**

* **Starters**: You include starters in your project to get the necessary dependencies.
* **Auto-Configuration**: Spring Boot detects these dependencies and automatically configures your application accordingly.

16. What is Spring Boot Actuator, and how does it help in monitoring and managing applications?

Spring Boot Actuator is a sub-project of Spring Boot that provides a set of production-ready features to help you monitor and manage your Spring Boot applications. It includes various built-in endpoints that give insights into the application's health, metrics, environment, and more.

**Key Features of Spring Boot Actuator**

1. **Health Checks**:
   * The /actuator/health endpoint provides information about the health of your application. It can be customized to include additional health indicators such as database connectivity, disk space, etc.

curl localhost:8080/actuator/health

1. **Metrics**:
   * The /actuator/metrics endpoint exposes various metrics related to your application, such as memory usage, CPU usage, HTTP request statistics, and more.

curl localhost:8080/actuator/metrics

* + These metrics can be integrated with monitoring systems like Prometheus and Grafana for visualization

1

.

1. **Environment**:
   * The /actuator/env endpoint provides details about the environment in which the application is running, including system properties, environment variables, and configuration properties.

curl localhost:8080/actuator/metrics

1. **Auditing**:
   * Actuator supports auditing of application events, which can be useful for tracking changes and actions within the application.

. How does Spring Boot support the creation of RESTful APIs?

18. What is Spring Boot DevTools, and how can it be useful during development?

Applications that use spring-boot-devtools automatically restart whenever files on the classpath change. This can be a useful feature when working in an IDE, as it gives a very fast feedback loop for code changes

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

19. Explain the purpose of the Spring Boot CommandLineRunner interface?

**Spring Boot CLI** (Command Line Interface) is a command line software or tool. This tool is provided by the Spring framework for quickly developing and testing Spring Boot applications from the command prompt.

The Spring Boot CommandLineRunner interface serves to execute specific code immediately after the Spring application context has been fully initialized, but before the application becomes fully operational. It is a functional interface with a run method that accepts a String array of command-line arguments. This interface enables developers to perform initialization tasks, setup procedures, or execute logic that needs to occur once at the application's startup.

To use CommandLineRunner, a class needs to implement the interface and override the run method. Spring Boot will automatically detect and execute all beans implementing this interface after the application context is ready. Multiple CommandLineRunner beans can be defined, and their execution order can be controlled using the @Order annotation or Ordered interface.

**CLI in Spring Boot** contains a run method() that is executed after the application startup.

**Steps to print Hello World in Spring Boot CLI:**

1. Go to Spring Initializr
2. Fill in the details as per the requirements.
3. Click on Generate which will download the starter project
4. Extract the zip file.
5. Go to SpringBootAppApplication class
6. Run the SpringBootAppApplication class and wait for the Tomcat server

@Component

public class MyCommandLineRunner implements CommandLineRunner {

@Override

public void run(String... args) throws Exception {

// Code to be executed after application startup

System.out.println("Application started with arguments: " + Arrays.toString(args));

// Perform initialization tasks, etc.

}

}

21. Describe the Spring Bean lifecycle. How does the initialization and destruction of beans happen in Spring?

The Spring bean lifecycle encompasses the series of stages a bean goes through from its creation to its destruction within the Spring container. These stages include bean definition, instantiation, population of properties, initialization, usage, and destruction.

Initialization

Initialization of a bean involves the following steps:

Bean Instantiation:

The Spring container creates an instance of the bean, typically using the bean's constructor.

Dependency Injection:

Spring injects any required dependencies into the bean, either through constructor injection, setter injection, or field injection.

Initialization Callbacks:

Spring provides several ways to execute initialization logic:

InitializingBean Interface: If a bean implements the InitializingBean interface, the afterPropertiesSet() method is called after dependency injection.

@PostConstruct Annotation: Methods annotated with @PostConstruct are executed after dependency injection.

Custom init-method: A custom initialization method can be specified in the bean configuration (XML or annotation-based).

Destruction

Destruction of a bean occurs when the Spring container is shut down or when a bean is no longer needed. The following steps are involved:

Destruction Callbacks: Similar to initialization, Spring offers several ways to execute destruction logic:

DisposableBean Interface: If a bean implements the DisposableBean interface, the destroy() method is called before the bean is destroyed.

@PreDestroy Annotation: Methods annotated with @PreDestroy are executed before the bean is destroyed.

Custom destroy-method: A custom destruction method can be specified in the bean configuration.

Bean Destruction: The Spring container removes the bean from its management.

Scope and Lifecycle

The bean scope affects the lifecycle. Singleton beans have a lifecycle tied to the Spring container, while prototype beans have a shorter lifecycle, with the container not managing their destruction.

**Initialization and Destruction**

* **Initialization**:
  + During initialization, Spring ensures that all dependencies are injected and any initialization logic is executed. This includes calling methods defined by InitializingBean or custom init methods.
* **Destruction**:
  + During destruction, Spring ensures that any cleanup logic is executed. This includes calling methods defined by DisposableBean or custom destroy methods.

Explain the role of the @Transactional annotation and its attributes in Spring. How does it work under the hood?

The @Transactional annotation in Spring is used to manage transaction boundaries declaratively. It simplifies transaction management by allowing you to specify transaction behavior directly on methods or classes. Here's a detailed explanation:

**Role of @Transactional**

* **Transaction Management**: The primary role of @Transactional is to define the scope of a single database transaction. It ensures that the annotated method or class runs within a transaction context.
* **Declarative Approach**: Instead of managing transactions programmatically, you can use @Transactional to declaratively control transaction behavior.

**Attributes of @Transactional**

The @Transactional annotation comes with several attributes that allow you to customize transaction behavior:

1. **propagation**:
   * Defines how transactions should be propagated. Common values include:
     + REQUIRED: Uses the current transaction or creates a new one if none exists.
     + REQUIRES\_NEW: Always creates a new transaction, suspending the current one if it exists.
     + SUPPORTS: Runs within a transaction if one exists, otherwise runs non-transactionally.
     + MANDATORY: Requires an existing transaction; throws an exception if none exists.
     + NOT\_SUPPORTED: Runs non-transactionally, suspending the current transaction if it exists.
     + NEVER: Runs non-transactionally; throws an exception if a transaction exists.
     + NESTED: Runs within a nested transaction if supported.
2. **isolation**:
   * Defines the isolation level of the transaction. Common values include:
     + DEFAULT: Uses the default isolation level of the underlying database.
     + READ\_UNCOMMITTED: Allows dirty reads.
     + READ\_COMMITTED: Prevents dirty reads.
     + REPEATABLE\_READ: Prevents dirty reads and non-repeatable reads.
     + SERIALIZABLE: Ensures full isolation, preventing dirty reads, non-repeatable reads, and phantom reads.
3. **timeout**:
   * Specifies the timeout for the transaction in seconds. If the transaction takes longer than the specified timeout, it will be rolled back.
4. **readOnly**:
   * Indicates whether the transaction is read-only. This can be used to optimize transaction performance for read-only operations.
5. **rollbackFor**:
   * Specifies which exceptions should trigger a rollback. You can list multiple exception classes.
6. **noRollbackFor**:
   * Specifies which exceptions should not trigger a rollback. You can list multiple exception classes.

**How @Transactional Works Under the Hood**

1. **Proxy Creation**:
   * When a method or class is annotated with @Transactional, Spring creates a proxy for the bean. This proxy intercepts method calls to manage transaction boundaries.
2. **Transaction Interception**:
   * The proxy intercepts the method call and starts a transaction based on the attributes specified in @Transactional.
3. **Transaction Management**:
   * Spring uses a PlatformTransactionManager to manage the transaction. This can be a DataSourceTransactionManager, JpaTransactionManager, or another implementation depending on the persistence technology used.
4. **Commit or Rollback**:
   * After the method execution, the proxy decides whether to commit or roll back the transaction based on the method's outcome and the specified attributes (e.g., rollbackFor, noRollbackFor).

**Example**

Here's a simple example to illustrate:

@Service

public class MyService {

    @Transactional(propagation = Propagation.REQUIRED, isolation = Isolation.READ\_COMMITTED, timeout = 30, readOnly = false)

    public void performTransaction() {

        // Transactional code

    }

}

In this example, performTransaction() will run within a transaction with the specified attributes.y

23. What are Spring profiles, and how can they be used to manage different environment-specific configurations?

Spring Profiles is a powerful feature in the Spring Framework that allows developers to manage and organize different configurations for various environments such as **development**, **testing**, and **production**. Profiles help you enable or disable beans, configurations, and properties based on the currently active environment.

**What Are Spring Profiles?**

Spring Profiles allow you to define separate configurations for different environments. You can specify which profile is active at runtime, and Spring will load the appropriate beans and configurations based on that active profile.

**How to Use Spring Profiles**

**Defining Profiles with**@Profile**Annotation**

You can use the @Profile annotation on Spring components (e.g., @Service, @Controller, @Configuration) to specify that those beans should only be created when a specific profile is active.

**Example**: Suppose you have two different data sources — one for development and one for production.

* **Development Data Source**:

@Profile("dev")  
@Configuration  
public class DevDataSourceConfig {  
  
 @Bean  
 public DataSource dataSource() {  
 // Create a data source for development (e.g., H2 database)  
 return new H2DataSource();  
 }  
}

* **Production Data Source**:

@Profile("prod")  
@Configuration  
public class ProdDataSourceConfig {  
  
 @Bean  
 public DataSource dataSource() {  
 // Create a data source for production (e.g., MySQL)  
 return new MysqlDataSource();  
 }  
}

In this example, Spring will load either the development or production data source configuration based on the active profile.

**Activating Profiles**

You can activate a profile using several methods, depending on your deployment strategy:

**Via application.properties or application.yml**: You can activate a profile in your configuration file by setting the spring.profiles.active property.

* **In application.properties**:

spring.profiles.active=dev

26. What is Spring Data JPA, and how does it simplify data access in a Spring application?

**Key Features of Spring Data JPA**

1. **Repository Abstraction**:
   * Spring Data JPA introduces repository interfaces that provide CRUD operations and query methods without the need for boilerplate code. You can define your repository interfaces by extending JpaRepository, CrudRepository, or other repository interfaces.
2. **Query Methods**:
   * You can define query methods directly in your repository interfaces using method names. Spring Data JPA translates these method names into queries automatically. For example, findByLastName(String lastName) will generate a query to find entities by their last name.
3. **Custom Queries**:
   * If you need more complex queries, you can use the @Query annotation to define JPQL (Java Persistence Query Language) or native SQL queries directly in your repository methods.
4. **Pagination and Sorting**:
   * Spring Data JPA provides built-in support for pagination and sorting through the Pageable and Sort interfaces. This makes it easy to handle large datasets and retrieve data in a paginated manner.

**Example**

**Here's a simple example to illustrate how Spring Data JPA works:**

**Entity Class**

**@Entity**

**public class User {**

**@Id**

**@GeneratedValue(strategy = GenerationType.IDENTITY)**

**private Long id;**

**private String firstName;**

**private String lastName;**

**// Getters and setters**

**}**

**Repository Interface**

**public interface UserRepository extends JpaRepository<user, long> {</user, long>**

**List findByLastName(String lastName);**

**}**

**Service Class**

**@Service**

**public class UserService {**

**@Autowired**

**private UserRepository userRepository;**

**public List getUsersByLastName(String lastName) {**

**return userRepository.findByLastName(lastName);**

**}**

**}**

27. Describe the use of Spring Boot's `@Conditional` annotation and provide examples of scenarios where it can be beneficial

**The @Conditional annotation in Spring Boot is used to conditionally register beans based on certain criteria. It allows you to control bean creation and configuration dynamically, depending on the environment or other conditions.**

**Use of @Conditional Annotation**

The @Conditional annotation can be applied to classes or methods to conditionally include beans in the Spring context. It works by evaluating a condition, and if the condition is met, the bean is registered; otherwise, it is not.

**Common Conditional Annotations**

Spring Boot provides several built-in conditional annotations:

1. **@ConditionalOnProperty**:
   * Registers a bean if a specific property is set in the application configuration.
   * Example: @ConditionalOnProperty(name = "feature.enabled", havingValue = "true")
2. **@ConditionalOnMissingBean**:
   * Registers a bean if a specific bean is not already present in the context.
   * Example: @ConditionalOnMissingBean(DataSource.class)
3. **@ConditionalOnBean**:
   * Registers a bean if a specific bean is already present in the context.
   * Example: @ConditionalOnBean(DataSource.class)
4. **@ConditionalOnClass**:
   * Registers a bean if a specific class is present on the classpath.
   * Example: @ConditionalOnClass(name = "com.example.SomeClass")
5. **@ConditionalOnMissingClass**:
   * Registers a bean if a specific class is not present on the classpath.
   * Example: @ConditionalOnMissingClass(name = "com.example.SomeClass")

**Examples of Scenarios**

**Example 1: Conditional Bean Registration Based on Property**

**@Configuration**

**public class FeatureConfig {**

**@Bean**

**@ConditionalOnProperty(name = "feature.enabled", havingValue = "true")**

**public FeatureService featureService() {**

**return new FeatureService();**

**}**

**}**

In this example, FeatureService will only be registered if the property feature.enabled is set to true in the application configuration.

**Example 3: Conditional Bean Registration Based on Existing Bean**

**@Configuration**

**public class CacheConfig {**

**@Bean**

**@ConditionalOnBean(CacheManager.class)**

**public CacheService cacheService(CacheManager cacheManager) {**

**return new CacheService(cacheManager);**

**}**

**}**

In this example, cacheService will only be registered if a CacheManager bean is already present in the context.

28. Explain the purpose and usage of Spring's `@Async` annotation for asynchronous method execution.

**Benefits of @Async Annotation**

@Async annotation is used when you wanna make any method asynchronize. Annotating a method of a bean with @Async will make it execute in a separate thread. In other words, the caller will not wait for the completion of the called method.

The @Async annotation in Spring enables asynchronous method execution. When a method is annotated with @Async, Spring executes it in a separate thread, allowing the calling method to continue its execution without waiting for the completion of the annotated method. This is particularly useful for tasks that are time-consuming or I/O-bound, as it prevents the main thread from being blocked, improving the application's responsiveness and performance.

To use @Async, it is necessary to first enable asynchronous processing in the Spring configuration, typically by using the @EnableAsync annotation. After enabling asynchronous processing, the @Async annotation can be applied to any public method of a Spring-managed bean. When the annotated method is called, Spring will submit it to a thread pool for execution.

Java

@Configuration  
@EnableAsync  
public class AsyncConfig {  
 *// Configuration for thread pool (optional)*  
}  
  
@Service  
public class MyService {  
 @Async  
 public void myAsyncMethod() {  
 *// Long-running task*  
 }  
}

In the example above, myAsyncMethod() will be executed asynchronously in a separate thread. The caller of this method will not wait for it to complete and will continue its execution.

It's important to note that methods annotated with @Async should not be called from within the same class, as this will bypass the Spring proxy and result in synchronous execution.

2. Flow of Spring Boot Application?

* Starts by calling main method of main class.
* The run method of spring Application is called. This method starts the application by creating an application context and initializing it.
* once the application is initialized, the run method starts the applications embedded web server.

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MyApplication

{

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

* **Start Application:**

When a Spring Boot application starts, the main() method typically annotated with @SpringBootApplication is invoked. This annotation combines @EnableAutoConfiguration, @ComponentScan, and @SpringBootConfiguration to enable auto-configuration, component scanning, and configuration detection. Spring Boot initializes the application context, setting up the environment for the application.

* **Component Scan:**

Spring Boot scans the classpath for classes annotated with @Component, @Service, @Repository, and @Controller, registering them as beans within the application context. This process identifies the components that make up the application's logic and structure.

* **Auto-Configuration:**

Based on the dependencies present in the project and the configuration settings, Spring Boot automatically configures beans. For instance, if a database dependency is detected, Spring Boot can configure a datasource. This automatic configuration simplifies setup and reduces boilerplate code.

* **Run Application:**

Spring Boot starts the embedded server, such as Tomcat, Jetty, or Netty, depending on the configured dependencies. The server begins listening for incoming requests, and the application is ready to handle them.

* **Handle Requests:**

Upon receiving an HTTP request, Spring Boot's DispatcherServlet routes the request to the appropriate controller method based on the request mapping annotations (e.g., @GetMapping, @PostMapping). The controller processes the request, interacts with services and repositories as needed, and returns a response to the client.