**What are the Features of Spring Boot?**

There are many useful features of Spring Boot. Some of them are mentioned below:

**Auto-configuration** – Spring Boot automatically configures dependencies by using @EnableAutoconfiguration annotation and reduces boilerplate code.

**Spring Boot Starter POM** – These Starter POMs are pre-configured dependencies for functions like database, security, maven configuration etc.

**Spring Boot CLI** (Command Line Interface) – This command line tool is generally for managing dependencies, creating projects and running the applications.

**Actuator** – Spring Boot Actuator provides health check, metrics and monitors the endpoints of the application. It also simplifies the troubleshooting management.

**Embedded Servers** – Spring Boot contains embedded servers like Tomcat and Jetty for quick application run. No need of external servers.

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

Spring Boot is a framework that creates stand-alone, production grade Spring based applications. So, this framework has so many advantages.

**Easy to use**: The majority of the boilerplate code required to create a Spring application is reduced by Spring Boot.

**Rapid Development**: Spring Boot’s opinionated approach and auto-configuration enable developers to quickly develop apps without the need for time-consuming setup, cutting down on development time.

**Scalable**: Spring Boot apps are intended to be scalable. This implies they may be simply scaled up or down to match your application’s needs.

**Production-ready**: Metrics, health checks, and externalized configuration are just a few of the features that Spring Boot includes and are designed for use in production environments.

**IOC Container:**

IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

to instantiate the application class

to configure the object

to assemble the dependencies between the objects

There are two types of IoC containers. They are:

**BeanFactory :**

Resource resource=new ClassPathResource("applicationContext.xml");

BeanFactory factory=new XmlBeanFactory(resource);

* It is the root interface for accessing the Spring container.
* Provides basic IoC container functionalities
* **Basic IoC Features:** Creates and manages beans and their dependencies.
* **Lazy Initialization:** By default, beans are lazily initialized. This means they are instantiated only when explicitly requested.
* **Resource Handling**: Uses Resource interface to access resources.

**ApplicationContext :**

ApplicationContext context =

new ClassPathXmlApplicationContext("applicationContext.xml");

* A sub-interface of BeanFactory.
* Provides advanced features on top of basic IoC functionalities.
* **All BeanFactory Features:** Inherits all functionalities of BeanFactory.
* **Internationalization:** Provides message resource handling for internationalization.
* **Web Application Context:** Has a variant for web-based applications, which understands Servlet contexts.
* **Eager Initialization:** By default, singletons are eagerly initialized to detect bean configuration issues early.

**Why use ApplicationContext over BeanFactory?**

While BeanFactory serves its purpose by providing basic IoC container functionalities, ApplicationContext is more feature-rich.

**Here are reasons you might prefer it:**

Ease of Configuration: ApplicationContext can integrate with Spring's AOP features, which is beneficial for applying cross-cutting concerns like logging and transactions.

Built-in Enterprise Features: ApplicationContext supports internationalization, JNDI access, event propagation, and various application layers, making it a more holistic choice for enterprise applications.

Web Support: If you're building a web application, the WebApplicationContext is an extension of ApplicationContext tailored for such environments.

**DI:**

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled.

What are the different bean scopes in spring?

---------------------------------------------------

1) singleton : The bean instance will be CREATED only once and same instance will be returned by the IOC container. It is the default scope.

2) prototype : The bean instance will be created each time when requested.

3) request : The bean instance will be created per HTTP request.

4) session : The bean instance will be created per HTTP session.

5) globalsession : The bean instance will be created per HTTP global session. It can be used in portlet context only.

2) How is HQL query created?

The HQL query is created with the help of the following syntax:

Session.createQuery

13) How can we add criteria to a SQL query?

A criterion is added to a SQL query by using the Session.createCriteria.

**What are the states of the object in hibernate?**

There are 3 states of the object (instance) in hibernate.

**Transient**: The object is in a transient state if it is just created but has no primary key (identifier) and not associated with a session.

**Persistent**: The object is in a persistent state if a session is open, and you just saved the instance in the database or retrieved the instance from the database.

**Detached**: The object is in a detached state if a session is closed. After detached state, the object comes to persistent state if you call lock() or update() method.

**To create a singleton class, we must follow the steps, given below:**

Ensure that only one instance of the class exists.Provide global access to that instance by:

Declaring all constructors of the class to be private.

Providing a static method that returns a reference to the instance. The lazy initialization concept is used to write the static methods.

The instance is stored as a private static variable.

public final class ClassSingleton {

private static ClassSingleton INSTANCE;

private String info = "Initial info class";

private ClassSingleton() {

}

public static ClassSingleton getInstance() {

if(INSTANCE == null) {

INSTANCE = new ClassSingleton();

}

return INSTANCE;

}

// getters and setters

}

------------------

How to make an immutable class in hibernate?

If you mark a class as mutable="false", the class will be treated as an immutable class. By default, it is mutable="true".

public static void FirstNonRepeat(String s)

{

for (int i = 0; i < s.length(); i++) {

if (s.indexOf(s.charAt(i), s.indexOf(s.charAt(i)) + 1) == -1) {

System.out.println("First non-repeating character is "+ s.charAt(i));

break;

}

}

return;

}

public static void main (String[] args) {

String s = "geeksforgeeks";

FirstNonRepeat(s);

}

second highest max salary :

select \*from employee

group by salary

order by salary desc limit 1,1;

**plugins:**

Plugins are software components that extend the functionality of another software application. In Java Spring Boot, plugins can be used to add new features, customize existing features, or integrate with third-party systems.

There are many different plugins available for Spring Boot, and the best ones for you will depend on your specific needs. However, some of the most popular plugins include:

The Spring Boot Maven Plugin: This plugin provides Spring Boot support in Maven, letting you package executable jar or war archives and run an application “in-place”.

The Spring Boot Gradle Plugin: This plugin provides Spring Boot support in Gradle, letting you package executable jar or war archives, run Spring Boot applications, and use the dependency management provided by spring-boot-dependencies.

A **JDBC driver** : is a software component enabling a Java application to interact with a database.

JDBC drivers are the software components which implements interfaces in JDBC APIs to enable java application to interact with the database.

**@Component** : is a general-purpose annotation used to declare any class as a Spring-managed bean class.

**@Service** is a specialization of the @Component annotation, to declare any class as a Business logic (Service) class.

**@Required**: It applies to the bean setter method. It indicates that the annotated bean must be populated at configuration time with the required property, else it throws an exception BeanInitilizationException.

**@Autowired**: Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to autowire spring bean on setter methods, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.

Example

@Component

public class Customer

{

private Person person;

@Autowired

public Customer(Person person)

{

this.person=person;

}

}

**@Configuration**: It is a class- annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

level

Example

@Configuration

public class Vehicle

{

@BeanVehicle engine()

{

return new Vehicle();

}

}

**@ComponentScan**: It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components. When @ComponentScan is used at the class level, Spring scans the specified packages and their sub-packages, looking for classes annotated with stereotype annotations such as @Component, @Service, @Repository, etc. It then creates instances of these classes and registers them as beans in the Spring application context.

Example

@ComponentScan(basePackages = "com.javatpoint")

@Configuration

public class ScanComponent

{

// ...

}

**@Bean**: It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.

Example

@Bean

public BeanExample beanExample()

{

return new BeanExample ();

}

Spring Framework Stereotype Annotations

**@Component**: It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with @Component is found during the classpath. The Spring Framework pick it up and configure it in the application context as a Spring Bean.

Example

@Component

public class Student

{

.......

}

**@Controller**: The @Controller is a class-level annotation. It is a specialization of @Component. It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with **@RequestMapping** annotation.

Example

@Controller

@RequestMapping("books")

public class BooksController

{

@RequestMapping(value = "/{name}", method = RequestMethod.GET)

public Employee getBooksByName()

{

return booksTemplate;

}

}

**@Service**: It is also used at class level. It tells the Spring that class contains the business logic.

package com.javatpoint;

@Service

public class TestService

{

public void service1()

{

//business code

}

}

**@Repository**: It is a class-level annotation. The repository is a DAOs (Data Access Object) that access the database directly. The repository does all the operations related to the database.

package com.javatpoint;

@Repository

public class TestRepository

{

public void delete()

{ //persistence code } }

**Spring Boot Annotations :**

**@EnableAutoConfiguration:** It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. @SpringBootApplication.

**@SpringBootApplication**: It is a combination of three annotations @EnableAutoConfiguration, @ComponentScan, and @Configuration.

Spring MVC and REST Annotations

**@RequestMapping**: It is used to map the web requests. It has many optional elements like consumes, header, method, name, params, path, produces, and value. We use it with the class as well as the method.

Example

@Controller

public class BooksController

{

@RequestMapping("/computer-science/books")

public String getAllBooks(Model model)

{

//application code

return "bookList";

}

**@GetMapping**: It maps the HTTP GET requests on the specific handler method. It is used to create a web service endpoint that fetches It is used instead of using: @RequestMapping(method = RequestMethod.GET)

**@PostMapping**: It maps the HTTP POST requests on the specific handler method. It is used to create a web service endpoint that creates It is used instead of using: @RequestMapping(method = RequestMethod.POST)

**@PutMapping**: It maps the HTTP PUT requests on the specific handler method. It is used to create a web service endpoint that creates or updates It is used instead of using: @RequestMapping(method = RequestMethod.PUT)

**@DeleteMapping**: It maps the HTTP DELETE requests on the specific handler method. It is used to create a web service endpoint that deletes a resource. It is used instead of using: @RequestMapping(method = RequestMethod.DELETE)

**@PatchMapping**: It maps the HTTP PATCH requests on the specific handler method. It is used instead of using: @RequestMapping(method = RequestMethod.PATCH)

**@RequestBody**: It is used to bind HTTP request with an object in a method parameter. Internally it uses HTTP MessageConverters to convert the body of the request. When we annotate a method parameter with @RequestBody, the Spring framework binds the incoming HTTP request body to that parameter.

**@ResponseBody**: It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.

**@PathVariable**: It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.

**@RequestParam**: It is used to extract the query parameters form the URL. It is also known as a query parameter. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.

**@RequestHeader**: It is used to get the details about the HTTP request headers. We use this annotation as a method parameter. The optional elements of the annotation are name, required, value, defaultValue. For each detail in the header, we should specify separate annotations. We can use it m ultiple time in a method

**@RestController**: It can be considered as a combination of @Controller and @ResponseBody annotations. The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.

**@RequestAttribute**: It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

In Spring Boot applications that use JPA, the **@Entity** annotation is typically applied to domain model classes. It indicates that instances of the class can be persisted to a database using an ORM (Object-Relational Mapping) framework, such as Hibernate.

It indicates that instances of the class can be persisted to a database using an ORM (Object-Relational Mapping) framework, such as Hibernate.

40. How do you sum all the elements in an array?

Use for loop to iterate through the array and keep adding the elements in that array.

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

int sum = 0;

for (int i : array)

sum += i;

System.out.println(sum);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

String a[] = { "A", "E", "I" };

String b[] = { "O", "U" };

List list = new ArrayList(Arrays.asList(a));

list.addAll(Arrays.asList(b));

Object[] c = list.toArray();

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

**Spring Boot – Transaction Management Using @Transactional Annotation**

-----------------------------

@Transactional annotation is the metadata used for managing transactions in the Spring Boot application. To configure Spring Transaction, this annotation can be applied at the class level or method level. In an enterprise application, a transaction is a sequence of actions performed by the application that together pipelined to perform a single operation. For example, booking a flight ticket is also a transaction where the end user has to enter his information and then make a payment to book the ticket.

**The Object class in Java is the root of the class hierarchy**. Every class in Java inherits the methods of the Object class. These methods are:

equals(): This method compares two objects for equality. It returns true if the two objects are equal, and false otherwise.

toString(): This method returns a string representation of the object.

hashCode(): This method returns a hash code for the object. A hash code is an integer value that is used to identify objects in hash tables.

getClass(): This method returns the Class object that represents the class of the object.

clone(): This method creates a copy of the object.

finalize(): This method is called by the garbage collector when the object is no longer needed.

wait(): This method causes the current thread to wait until another thread calls the notify() or notifyAll() method on the object.

notify(): This method wakes up a single thread that is waiting on the object.

notifyAll(): This method wakes up all threads that are waiting on the object.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**CyclicBarrier** :

A CyclicBarrier is a synchronizer that allows a set of threads to wait for each other to reach a common execution point, also called a barrier.

**JDBC**:

Spring JdbcTemplate is a powerful mechanism to connect to the database and execute SQL queries.

**Hibernate : get() and load()** are two methods which is used to fetch data for the given identifier. They both belong to Hibernate session class. Get() method return null, If no row is available in the session cache or the database for the given identifier whereas load() method throws object not found exception

**IOC Containers difference** :

Autowiring can be done :

Modes Description

------------------------------------------------------------------------------------------------------------------------------------------

No -------------> This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring.

byName-------> It uses the name of the bean for injecting dependencies.

byType -------> It injects the dependency according to the type of bean.

Constructor----> It injects the required dependencies by invoking the constructor.

Autodetect------> The autodetect mode uses two other modes for autowiring – constructor and byType.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Co-Varient return types :**

As we know that in the overriding, the [return type](https://www.scientecheasy.com/2020/06/return-type-in-java.html/) of subclass method must be the same as the superclass method return type.

But, this rule is applicable until Java 1.4 version. From Java 1.5 onwards, a new covariant return type feature was introduced by Sun Microsystems. By using this feature, it is possible to override any method by changing the return type only.

If the return type of [overriding method](https://www.scientecheasy.com/2020/07/method-overriding-java.html/) in the subclass is a subtype of the declared return type of overridden method instead of being exactly the same type, it is known as covariant return type in Java.

In simple words, if the return type of the overriding method is a subclass of the return type of the overridden method (instead of being exactly the same type), this feature is known as a covariant return type. Let’s understand it with the help of a very simple example.

class Superclass {

Superclass getMe() {

return this;

}

}

class Subclass extends Superclass {

@Override

Subclass getMe() {

return this;

}

}

In this example code, the return type of getMe() method in the subclass Subclass is a subclass of the return type in the superclass Superclass. This is allowed in due to covariant return type feature in Java programming.

There are mainly three rules for covariant return types in Java that you should keep in mind. They are as follows:

1. The return type of overriding method in the subclass should be either the same as the return type of [superclass or subclass](https://www.scientecheasy.com/2020/07/java-superclass-subclass.html/).

2. The return type of overriding method in the subclass should not be a parent of the parent method return type.

3. The covariant return type is applicable only for object types, not for primitive types.

**System.out.println() :**

Below we can understand the parts of the java system out println.

1. ****System:**** This is the final class that is defined in java.lang package.
2. ****out:**** This is an instance of PrintStream type, and the access specifiers are final and public.
3. ****println():**** This is a method in the PrintStream class.

### What is a Marker Interface : An empty[interface in Java](https://www.simplilearn.com/tutorials/java-tutorial/java-interface" \o "interface in Java" \t "https://www.simplilearn.com/tutorials/java-tutorial/_blank) is referred to as a Marker interface. Serializable and Cloneable are some famous examples of Marker Interface.

### If we are using with clone() then we have to implement clonnable interface.

We can clone object withput clone() as below : as follows

**Using the assignment operator:**

You can create a copy of an object by assigning the reference of the original object to a new variable. However, this will only create a shallow copy of the object, which means that any changes made to the copy will also be reflected in the original object.

class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

} //getter and setters

}

public class Main {

public static void main(String[] args) {

Person person1 = new Person("John Doe", 30);

Person person2 = person1; // This creates a shallow copy of person1

person2.setName("Jane Doe");

System.out.println(person1.getName()); // This will print "Jane Doe"

}

}

**Using a copy constructor:**

A copy constructor is a special constructor that takes an object of the same class as its argument and creates a new object that is a copy of the original object. This will create a deep copy of the object, which means that any changes made to the copy will not be reflected in the original object.

**EX:** class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public Person(Person person) {

this.name = person.name;

this.age = person.age;

}//getter and setter

}

public class Main {

public static void main(String[] args) {

Person person1 = new Person("John Doe", 30);

Person person2 = new Person(person1); // This creates a deep copy of person1

person2.setName("Jane Doe");

System.out.println(person1.getName()); // This will print "John Doe"

}

}

**Define Wrapper Classes in Java.**

In Java, when you declare primitive datatypes, then Wrapper classes are responsible for converting them into objects(Reference types).

**Throw custom Exception :**

public class TestCustomException1

{

static void validate (int age) throws InvalidAgeException{

if(age < 18){

throw new InvalidAgeException("age is not valid to vote");

}

else {

System.out.println("welcome to vote");

}

}

public static void main(String args[])

{

try

{

// calling the method

validate(13);

}

catch (InvalidAgeException ex)

{

System.out.println("Caught the exception");

System.out.println("Exception occured: " + ex);

}

System.out.println("rest of the code...");

}

}

**This** : is a reference keyword. Used to refer to the current class properties like method,variable,instance and constructors.

**Super** : is a reference keyword. Used to refer to the immediate parent class object.

* >> operator does the job of right shifting the sign bits
* >>> operator is used in shifting out the zero-filled bits

### Why are generics used in Java Programming?

Compile-time type safety is provided by using generics. Compile-time type safety allows users to catch unnecessary invalid types at compile time. Generic methods and classes help programmers specify a single method declaration, a set of related methods, or related types with an available class declaration.

### What is the Daemon Thread : The Daemon thread can be defined as a thread with the least priority. This Daemon thread is designed to run in the background during the Garbage Collection in Java

**Lamda Syntax :**

*parameter* -> *expression*

*(parameter1, parameter2)* -> *expression*

*(parameter1, parameter2)* -> { *code block* }

*-----------------------------*

public class Main {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<Integer>();

numbers.add(5);

numbers.add(9);

numbers.add(8);

numbers.add(1);

numbers.forEach( (n) -> { System.out.println(n); } );

}} Output :

5  
9  
8  
1

**Lazy Loading in Hibernate :**

Lazy loading in Hibernate is a technique that defers the loading of related entities until they are actually needed. This can improve performance by reducing the amount of data that needs to be loaded into memory at once.

To enable lazy loading for a particular entity, you can use the @LazyCollection annotation. For example

@Entity  
public class Customer {  
  
@LazyCollection(LazyCollectionOption.EXTRA)  
private List<Order> orders;  
  
*// getters and setters*  
}

This will tell Hibernate to only load the orders collection when it is actually accessed.

Lazy loading can also be enabled globally for all entities by setting the hibernate.enable\_lazy\_load\_no\_trans property to true.

It is important to note that lazy loading can have some drawbacks. For example, if you try to access a lazily loaded entity outside of a transaction, Hibernate will throw an exception. Additionally, lazy loading can add some overhead to your application, as Hibernate needs to keep track of which entities have been loaded and which have not.

Overall, lazy loading is a powerful technique that can improve the performance of your Hibernate applications. However, it is important to be aware of the drawbacks before using it.

Spring MVC, the **@RequestBody**annotation binds the incoming HTTP request body to a method parameter, while the **@ResponseBody** annotation binds the return value to the outgoing HTTP response body. Both annotations use HTTP message converters to convert the HTTP request and response body to Java objects.

**Jedis**, a client library in Java for Redis. This popular in-memory data structure store can persist on a disk as well. It's driven by a keystore-based data structure to persist data and can be used as a database, cache, message broker, etc.

**Stream API :**

The Stream API in Java 8 is a powerful tool for processing collections of data. It provides a functional programming style for working with streams, which makes it easy to express complex operations in a concise and readable way.