**Spring Framework Introduction:**

Spring not only manages Web, it can handle other aspects of the application as well, so it is usually termed as end to end application development framework. So using spring we can develop not only Web we can develop core java, JEE, remoting (rmi), distributed, persistency and various other application types.

How many application types spring is supporting. Spring supports all the application development types in par with JEE. If spring supports all the application types in par with JEE why should I go to spring, I can use JEE to get the benefits?

JEE is an API. API stands for Application Programming Interfaces; API contains interfaces and classes, using which we can build applications. Generally these API has more number of classes, and these classes are interlinked or inter-related with each other. For e.g. if I want to execute a sql query,

I need statement Object in jdbc, but to create a Statement I need to know Connection and to create a Connection I should know DriverManager. In this way each class is inter-related with each other. So, building an application with few features of JEE also it demands programs to know all the classes, unless he end up in learning all he will not be able to start developing the applications. So,

API's are complex to understand and code. As API's are huge (contains more number of classes), it takes more time in learning and learning curve would be high. Apart from this API's will not provide boiler plate code.

Boiler plate code means this is the piece of code that has to be repeated across various parts of the application with minimal or no changes. For e.g. I want to execute a sql query, to execute a sql query I need to write the following lines of code.

**try{**

**Class.forName("DriverClassName");**

**Connection con = DriverManager.getConnection(url, un, pwd);**

**Statement stmt = con.createStatement();**

**ResultSet rs = stmt.executeQuery(); If(rs.next()) {**

**// grab the values**

**}catch(SqlException sqe) {**

**}**

Executing sql query is a common functionality that everyone has in their projects.

To execute a sql query everyone has to write more or less the same piece of logic.

When it is same, even then also I have to write the same duplicate logic all over

so, it is called boiler plate code.

Instead if we go to spring framework to execute sql query, spring provides one simple class to do this, we never need to write the whole logic rather we should

just call a method on the spring provided class. Internally spring classes will take

care of doing the above and will provide output.

So, in API's boiler plate code has to be written by programmer, this leads to

writing more and more number of lines of code. This has the following impacts

1. More amount of efforts are required as we need to write more code

2. Developers productivity will go down

3. As more lines of code chances of increasing the bugs will be also high

4. Maintenance cost involved in modifying the code also will be high

In spring as it provides boiler plate logic, we can avoid all the above said

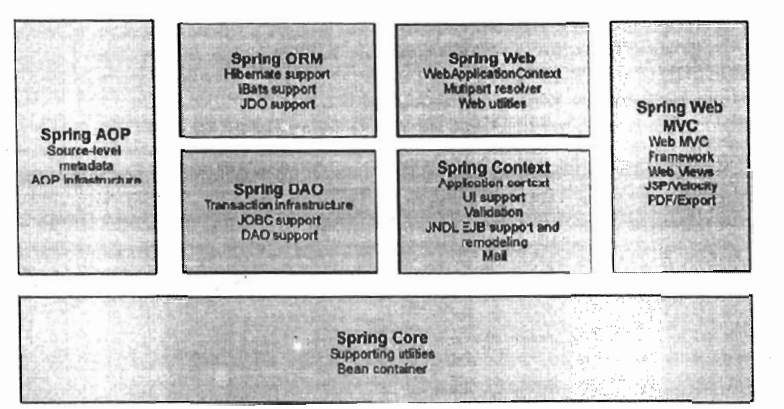
problems. That is the reason rather than working with JEE we are using spring.

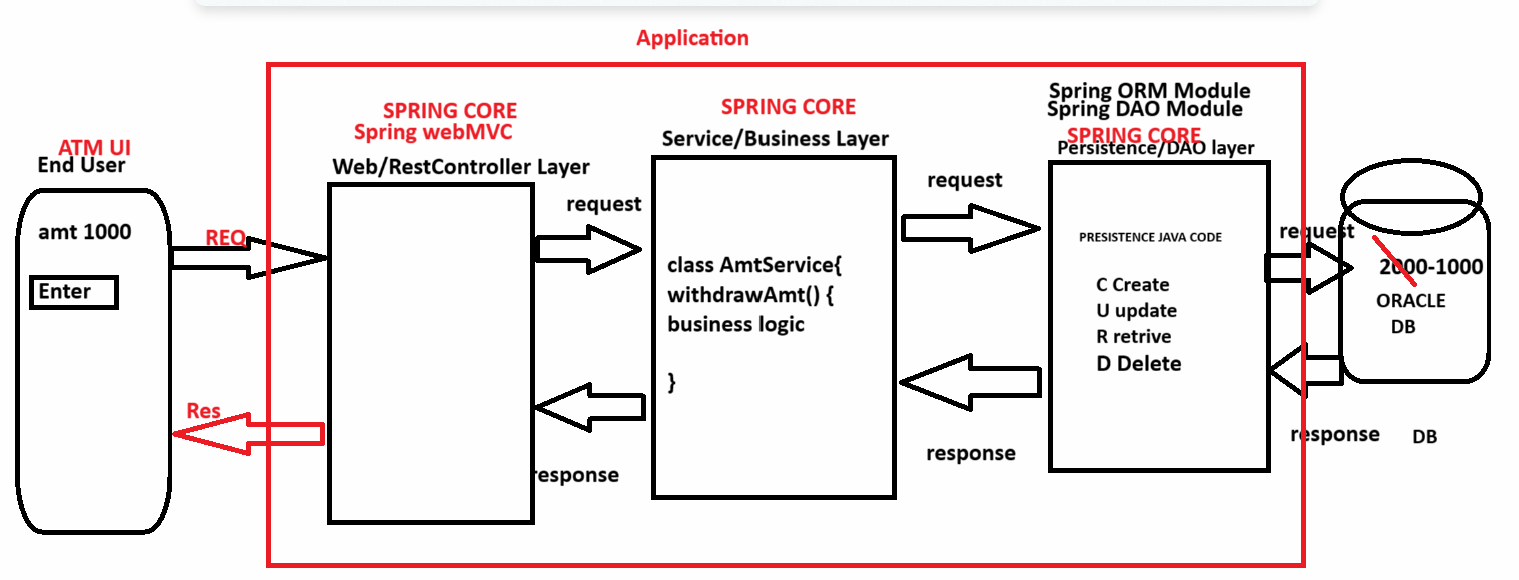
If spring supports all the development types in par with JEE, can I call spring as a replacement of JEE? Spring never replaces JEE rather spring complements JEE. Our application classes are going to talk to spring provided classes and spring classes internally talks to JEE provided classes. This means without JEE spring cannot work rather spring enriches and make JEE easier to use for the developer. This means spring is complementing JEE to use it easily.

A diagram of a course

AI-generated content may be incorrect.

If spring is providing various applications development types in par with JEE, is spring is light weight application development framework or heavy weight? If we look at the spring architecture, it has been designed keeping in view of light weight. The basic module in the spring is core, it spans across the breath of the spring, it is the module on which all the rest of the modules are built-on. Apart from core there are AOP, Jdbc, Transactions and MVC. In this way spring has several modules and all are dependent and spring core module. Refer to the below diagram to understand the same.





Here if we observe carefully each and every module don't have intersection or cross lines with other, this means there is least or no dependency with other module. Let's say if we want to build persistence application, programmer has to learn/use only spring core and spring Dao (jdbc) to develop application, he don't need to use all the other modules to build the same. This shows spring is flexible enough in offering what you want rather than what is unnecessary. So it is called light weight application framework.

**Spring Container**

The core component of the [spring framework](https://www.geeksforgeeks.org/spring/) is the [Inversion of Control (IOC)](https://www.geeksforgeeks.org/spring-understanding-inversion-of-control-with-example/) container, which is responsible for managing the lifecycle of objects called **beans**and their dependencies. Spring provides two types of containers:

* **BeanFactory Container**
  + This is the basic container that provides support for dependency injection.
  + In this, beans are instantiated only when explicitly requested.
  + It is lightweight and suitable for resource-constrained environments.
* **ApplicationContext Container**
  + This is an advanced container built on top of the BeanFactory.
  + It includes all the features of BeanFactory and adds extra functionalities such as internationalization, event propagation, and integration with other Spring modules.
  + In this, beans are created and configured at startup.

**Key feature of Spring Framework**

The features of the Spring framework, such as IoC, AOP, and transaction management, make it unique among Java frameworks. Some of the most important features include:

* IoC Container
* Data Access Framework (JDBC abstraction layer)
* Spring MVC
* Transaction Management
* Spring Web Services
* Spring TestContext Framework

**Spring IOC Container**

Spring IoC is a design principle where the control of object creation and lifecycle is transferred from the developer to the framework. The IoC container is responsible for:

1. Creating objects (Beans)
2. Configuring dependencies via Dependency Injection (DI)
3. Managing the entire lifecycle of beans, from instantiation to destruction
4. Reading configuration metadata (XML, Java Config, or Annotations)

***Note:*** *The objects managed by the container are called* ***beans.***

The below diagram demonstrates how the Container makes use of Configuration metadata and Java POJO classes to manage beans.

**Working of IoC Container**

* The container reads configuration metadata (XML, Java annotations, or Java-based configuration) to understand how beans should be creates and wired together.
* It uses Dependency Injection (DI) to inject dependencies into beans at runtime.
* The container manages the entire lifecycle of beans, from instantiation to destruction.

**BeanFactory vs ApplicationContext**

The below table demonstrates the key difference between BeanFactory and ApplicationContext

| **Feature** | **BeanFactory** | **ApplicationContext** |
| --- | --- | --- |
| **Annotation Support** | No | Yes |
| **Bean Instantiation/Wiring** | Yes | Yes |
| **Internationalization** | No | Yes |
| **Enterprise Services** | No | Yes |
| **ApplicationEvent Publication** | No | Yes |
| **Automatic BeanPostProcessor Registration** | No | Yes |
| **Loading Mechanism** | Lazy Loading | Aggressive Loading |
| **Automatic BeanFactoryPostProcessor Registration** | No | Yes |

**Why use Spring Framework?**

* **Modularity:** Use only the modules we need.
* **Flexibility**: Supports multiple configuration styles (XML, annotations, Java code).
* **Scalability:** Suitable for small applications as well as large enterprise systems.
* **Community Support:** Backed by a large and active community.

**Spring - Autowiring**

**Autowiring** in the Spring framework can inject dependencies automatically. The Spring container detects those dependencies specified in the configuration file and the relationship between the beans. This is referred to as **Autowiring in Spring**. To enable Autowiring in the Spring application we should use[@Autowired](https://www.geeksforgeeks.org/spring-autowired-annotation/)annotation. Autowiring in Spring internally uses constructor injection. An autowired application requires fewer lines of code comparatively but at the same time, it provides very little flexibility to the programmer.

**Modes of Autowiring**

| **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. |

**Bean Scopes**

refer to the **lifecycle of a Bean**, which means when the object of a Bean is instantiated, how long it lives, and how many objects are created for that Bean throughout its lifetime. Basically, it controls the instance creation of the bean, and it is managed by the **Spring container.**

**Bean Scopes in Spring**

The [Spring](https://www.geeksforgeeks.org/introduction-to-spring-framework/) framework provides **five scopes**for a bean. We can use three of them only in the context of a web-aware **Spring ApplicationContext,** and the rest of the two are available for both an IoC container and a Spring-MVC container. The following are the different scopes provided for a bean:

1. **Singleton**: Only one instance will be created for a single bean definition per Spring IoC container, and the same object will be shared for each request made for that bean.
2. **Prototype:** A new instance will be created for a single bean definition every time a request is made for that bean.
3. **Request**: A new instance will be created for a single bean definition every time an HTTP request is made for that bean. But only valid in the context of a web-aware Spring ApplicationContext.
4. **Session:** Scopes a single bean definition to the lifecycle of an HTTP Session. But only valid in the context of a web-aware Spring ApplicationContext.
5. **Global-Session**: Scopes a single bean definition to the lifecycle of a global HTTP Session. It is also only valid in the context of a web-aware Spring ApplicationContext.

**Difference Between Singleton and Prototype Bean**

| **Singleton** | **Prototype** |
| --- | --- |
| Only one instance is created for a single bean definition per Spring IoC container. | A new instance is created for a single bean definition every time a request is made for that bean. |
| Same object is shared for each request made for that bean. i.e. The same object is returned each time it is injected. | For each new request a new instance is created. i.e. A new object is created each time it is injected. |
| By default, scope of a bean is singleton. So we don't need to declare a bean as singleton explicitly. | By default, scope is not prototype, so you have to declare the scope of a bean as prototype explicitly. |
| Singleton scope should be used for stateless beans. | While prototype scope is used for all beans that are stateful. |

**Spring - init() and destroy()**

During the Spring Application Development, sometimes when the spring beans are created developers are required to execute the initialization operations and the cleanup operations before the bean is destroyed. In the spring framework, we can use the init-method and the destroy-method labels in the bean configuration.

**init() Method**

In the real-world application **init()**method is the one you will find everywhere. init-method is the attribute of the spring <bean> tag. It is utilized to declare a custom method for the bean that will act as the bean initialization method. We can define it as follows.

*<bean id="student" class="com.amiya.Student" init-method="myPostConstruct">*

Here **myPostConstruct()** method is the bean initialization method in the Student class. This initialization method is called after initializing bean properties. We can use such an initialization method to validate the value of bean properties or initialize any task. The InitializingBean interface in Spring performs the same task but it is highly coupled to Spring, so we should prefer the init-method.

**Why init()?**

* You can add custom code/logic during bean initialization
* It can be used for setting up resources like database/socket/file etc.

**destroy() Method**

The destroy() method will be called before the bean is removed from the container. destroy-method is a bean attribute using which we can assign a custom bean method that will be called just before the bean is destroyed. To use the destroy-method attribute, we do as follows.

*<bean id="student" class="com.amiya.Student" destroy-method="myPreDestroy">*

Here**myPreDestroy()**method will be defined in the Student class. Spring will call this method just before destroying the bean. destroy-method is used to release resources or perform some destruction task. DisposableBean interface in spring performs the same task but it is highly coupled to spring, so we should prefer destroy-method

**Spring Framework Standalone Collections**

Spring Framework allows to inject of collection objects into a bean through constructor dependency injection or setter dependency injection using **<list>,<map>,<set>**. create standalone Collection is reusable collection beans, util schema is used.

**Spring – Core Annotation**

**Spring Annotations** are a form of metadata that provides data about a program. Annotations are used to provide supplemental information about a program. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

[Spring Framework](https://www.geeksforgeeks.org/spring/) is one of the most popular frameworks for building Java applications. It provides a wide range of annotations that simplify configuration, dependency injection, and bean management. In this article, we will explore**Spring Core Annotations, their types, and how they are used in Spring applications.**

**Types of Spring Framework Annotations**

The below diagram demonstrates the types of Spring Framework Annotations

* **Spring Core Annotations:**Used for dependency injection, bean configuration, and context management.
* **Spring Web Annotations:**Used for building web applications and RESTful services.
* **Spring Boot Annotations:**Simplify Spring Boot application configuration.
* **Spring Scheduling Annotations:** Used for scheduling tasks.
* **Spring Data Annotations:**Used for data access and persistence.
* **Spring Bean Annotations:**Used for defining and managing beans.

**Spring Core Annotations**

Spring annotations present in the**org.springframework.beans.factory.annotation** and **org.springframework.context.annotation** packages are commonly known as Spring Core annotations. We can divide them into two categories:

* **DI-Related Annotations:**Used for dependency injection.
* **Context Configuration Annotations:**Used for configuring the Spring application context.

**DI-Related Annotations**

These annotations are used to manage dependency injection (DI) in Spring. The most commonly used DI-related annotations are listed below

**1. @Autowired:**This annotation is used to inject dependencies automatically. It can be applied to fields, setter methods, and constructors.

**Example of Field Injection:**

@Component

**public** **class** **Student** {

@Autowired

**private** Address address;

}

**Example of Constructor Injection:**

@Component

**public** **class** **Student** {

**private** Address address;

@Autowired

**public** Student(Address address) {

**this**.address = address;

}

}

**Example of Setter Injection:**

@Component

**public** **class** **Student** {

**private** Address address;

@Autowired

**public** void setAddress(Address address) {

**this**.address = address;

}

}

**2. @Qualifier:** The @Qualifier annotation is used to resolve the autowiring conflict when there are multiple beans of the same type. The @Qualifier annotation can be used on any class annotated with @Component or on methods annotated with @Bean. This annotation can also be applied to constructor arguments or method parameters.

**Example:**

@Component

**public** **class** **VehicleService** {

@Autowired

@Qualifier("bike")

**private** Vehicle vehicle;

}

**3. @Primary:**This indicates that a particular bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency. If exactly one ‘primary’ bean exists among the candidates, it will be the autowired value. Let’s understand this statement with an example

**Example:**In some cases, we need to register more than one bean of the same type. In this example employee1() and employee2() beans of the Employee type:

@Configuration

**public** **class** **Config** {

@Bean

**public** Employee employee1() {

**return** **new** Employee("Employee1");

}

@Bean

@Primary

**public** Employee employee2() {

**return** **new** Employee("Employee2");

}

}

In this case, if we try to run the application Spring will throw ***NoUniqueBeanDefinitionException***. To resolve this issue Spring offers the @Primary annotation.

@Configuration

public class Config {

@Bean

public Employee Employee1() {

return new Employee("Employee1");

}

@Bean

@Primary

public Employee Employee2() {

return new Employee("Employee2");

}

}

**4. @Bean:**This annotation is used to define a bean in a configuration class. It is typically used in @Configuration classes.

**Example:**

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** MyService myService() {

**return** **new** MyService();

}

}

**5. @Lazy:** This annotation delays the initialization of a bean until it is first requested.

**Example:**

@Component

@Lazy

**public** **class** **MyService** {

*// Bean will be initialized only when requested*

}

**6. @Value:**This annotation is used to inject values from properties files or environment variables into fields.

**Example:**

@Component

**public** **class** **MyService** {

@Value("${app.name}")

**private** String appName;

}

**7. @Scope:**This annotation defines the scope of a bean, such as singleton, prototype, request, or session.

**Example:**

@Component

@Scope("prototype")

**public** **class** **MyService** {

*// Bean will be created each time it is requested*

}

**8. @Lookup:**This annotation is used for method injection. It allows a method to return a new instance of a bean each time it is called.

**Example:**

@Component

**public** **abstract** **class** **MyService** {

@Lookup

**public** **abstract** MyBean getMyBean();

}

**9. @Required:** This annotation was used to enforce that a particular property must be injected.

***Note:*** *This annotation is deprecated as of Spring 5.1*

**Context Configuration Annotations**

These annotations are used to configure the Spring application context.

**1. @Configuration:**This annotation indicates that a class is a configuration class and can define beans using @Bean methods.

**Example:**

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** MyService myService() {

**return** **new** MyService();

}

}

**2. @ComponentScan:**This annotation is used to specify the packages to scan for Spring components like @Component, @Service, @Repository, etc.

**Example:**

@Configuration

@ComponentScan("com.example")

**public** **class** **AppConfig** {

}

**3. @Import:**This annotation is used to import one or more @Configuration classes into another configuration class.

**Example:**

@Configuration

@Import({Config1.class, Config2.class})

**public** **class** **AppConfig** {

}

**4. @ImportResource:**This annotation is used to import XML configuration files into a Java-based configuration class.

**Example:**

@Configuration

@ImportResource("classpath:app-config.xml")

**public** **class** **AppConfig** {

}

**5. @PropertySource:**This annotation is used to load properties files into the Spring environment.

**Example:**

@Configuration

@PropertySource("classpath:app.properties")

**public** **class** **AppConfig** {

}

**6. @Profile:**This annotation is used to conditionally load beans based on active profiles.

**Example:**

@Component

@Profile("dev")

**public** **class** **DevDataSource** {

*// Bean will be loaded only if the "dev" profile is active*

}

**7. @Conditional:** This annotation is used to conditionally register beans based on specific conditions.

**Example:**

@Bean

@Conditional(MyCondition.class)

**public** MyService myService() {

**return** **new** MyService();

}

**Spring @Value Annotation**

The @Value annotation in Spring is one of the most important annotations. It is used to assign default values to variables and method arguments. It allows us to inject values from spring environment variables, system variables, and properties files. It also supports Spring Expression Language (SpEL).

Key Points about @Value:

The @Value annotation is used to inject values into fields, methods, or constructor parameters in Spring beans, typically from property files, environment variables, or expressions.

It also supports SpEL, allowing dynamic values or expressions to be evaluated and injected.

The @Value annotation also binds values from application.properties or application.yml to Spring beans

The @Value annotation also converts the injected value to the required data type.

**Spring @Configuration Annotation**

The **@Configuration annotation**in Spring is one of the most important annotations. It indicates that a class contains @Bean definition methods, which the Spring container can process to generate Spring Beans for use in the application. This annotation is part of the Spring Core framework.

Let's understand the @Configuration annotation in [Spring](https://www.geeksforgeeks.org/spring/) with an example project.

**Steps to Implement @Configuration Annotation**

Suppose we already have a Java project, and all the Spring JAR files are imported into that project. No,w let’s create a simple class named College, and inside the class.

**Spring IoC (Inversion of Control)**

The Spring IoC Container is the core of the Spring Framework. It creates and manages objects (beans) and injects dependencies. The IoC Container retrieves object configuration from:

* XML Configuration Files
* Java Configuration Classes
* Java Annotations

Since object creation and lifecycle management are handled by the IoC Container, developers do not need to manually instantiate dependencies. This reduces tight coupling in the application.

**Spring Dependency Injection**

Dependency Injection (DI) is a key feature provided by Spring IoC. The Spring Core module injects dependencies into objects via different injection methods, ensuring that components are loosely coupled.

There are two types of DI in Spring:

* Setter Dependency Injection (SDI)
* Constructor Dependency Injection (CDI)

**1. Setter Dependency Injection (SDI)**

In Setter Injection, dependencies are injected using setter methods. The property to be injected is declared inside the <property> tag in the XML configuration file.

**Example**:

*<bean id="myBean" class="com.example.MyClass">*

*<property name="dependency" ref="myDependency"/>*

*</bean>*

**2. Constructor Dependency Injection (CDI)**

In Constructor Injection, dependencies are passed via the class constructor. The dependency is set using the <constructor-arg> tag in the XML configuration file.

**Example:**

*<bean id="myBean" class="com.example.MyClass">*

*<constructor-arg ref="myDependency"/>*

*</bean>*

**Constructor Injection**

In Constructor Injection, Dependencies are injected using the class constructor. To configure Constructor Dependency Injection in a bean, we use the <constructor-arg> tag in the bean configuration file (beans.xml).

**Why Use Constructor Injection?**

Constructor injection is a preferred approach in many cases because of the following reasons, which are listed below:

* **Immutability**: Dependencies are set at the time of object creation, making the object immutable.
* **Clear Dependencies:** It makes dependencies explicit and ensures that the object is fully initialized when created.
* **Testability**: It simplifies unit testing since dependencies can be easily injected via the constructor.

**Spring - Setter Injection with Map**

**2. Setter-based dependency injection** –  This is done by the container calling setter methods on beans after invoking a no-argument constructor or no-argument static factory method to instantiate the bean. We need to use the **<property>** sub-element of the <bean> tag for setter injection.

*<bean id="beanId" class="BeanClass">*

*<property name="secondBean" ref="SecondBean"/>*

*<property name="message" value="This is message."/>*

*</bean>*

**Setter Injection with Map**

Setter Injection with Map is a technique in Spring Framework that allows us to inject a collection of key-value pairs into a bean using setter methods. This is particularly useful when we need to manage dynamic or configurable data, such as framework details, environment properties, or any other key-value-based information. Instead of hardcoding these values, we can inject them via Spring’s configuration, making our application more flexible and maintainable.

***This approach is ideal for scenarios where dependencies are optional or need to be injected dynamically at runtime.***

**Different Methods to Create a Spring Bean**

Here we are going to discuss how to create a Spring Bean in 3 different ways as follows:

1. Creating Bean Inside an XML Configuration File (beans.xml)
2. Using @Component Annotation
3. Using @Bean Annotation

**Method 1: Creating Bean Inside an XML Configuration File (beans.xml)**

One of the most popular ways to create a spring bean is to define a bean in an XML configuration file something like this.

<bean id="AnyUniqueId" class="YourClassName">

</bean>

Let us create a simple class Student having two attributes id and studentName and later creating a simple method to print the details of the student.

**Example**

*// Java Program to Illustrate Student Class*

*// Class*

**public** **class** **Student** {

*// Class data members*

**private** int id;

**private** String studentName;

*// Method*

**public** void displayInfo()

{

*// Print statement*

System.out.println("Student Name is " + studentName

+ " and Roll Number is " + id);

}

}

Now let’s create an XML file named **beans.xml** file in the project classpath. And inside this beans.xml file, we have to define our Student bean something like this. And that's it. In this way, you can create beans in spring.

**Example**

<?xml version="1.0" encoding="UTF-8"?>

**<beans** xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

https://www.springframework.org/schema/beans/spring-beans.xsd"**>**

**<bean** id="studentAmiya" class="Student"**>**

**</bean>**

**</beans>**

**Method 2: Using @Component Annotation**

Spring Annotations are a form of metadata that provides data about a program. Annotations are used to provide supplemental information about a program. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program. **@Component**is an annotation that allows Spring to automatically detect the custom beans.

**Example:**Suppose we have already a Java project and all the Spring JAR files are imported into that project. Now let's create a simple class named College and inside the class, we have a simple method. Below is the code for the **College.java** file.

**A.** File: College.java

*// Java Program to Illustrate College Class*

**package** **ComponentAnnotation**;

*// Class*

**public** **class** **College** {

*// Method*

**public** void test()

{

*// Print statement*

*// whenever this method is called*

System.out.println("Test College Method");

}

}

Now let's create a Bean for this class. So we can use **@Component**annotation for doing the same task. So we can modify our **College.java** file something like this. And that's it.

**B.** College.java

*// Java Program to Illustrate College Class*

**package** **ComponentAnnotation**;

*// Importing required classes*

**import** **org.springframework.stereotype.Component**;

@Component("collegeBean")

*// Class*

**public** **class** **College** {

*// Method*

**public** void test()

{

*// Print statement*

System.out.println("Test College Method");

}

}

**Method 3: Using @Bean Annotation**

One of the most important annotations in spring is the **@Bean annotation**which is applied on a method to specify that it returns a bean to be managed by Spring context. Spring Bean annotation is usually declared in Configuration classes methods.

Suppose we have already a Java project and all the Spring JAR files are imported into that project. Now let's create a simple class named College and inside the class, we have a simple method. Below is the code for the **College.java** file.

**A.** College.java

**package** **BeanAnnotation**;

**import** **org.springframework.stereotype.Component**;

**public** **class** **College** {

**public** void test(){

System.out.println("Test College Method");

}

}

Now let's create a Configuration class named **CollegeConfig**. Below is the code for the **CollegeConfig.java** file

**B.** CollegeConfig.java

**package** **ComponentAnnotation**;

**import** **org.springframework.context.annotation.ComponentScan**;

**import** **org.springframework.context.annotation.Configuration**;

@Configuration

**public** **class** **CollegeConfig** {

}

Here, we are going to create the spring beans using the **@Bean annotation**. To create the College class bean using the @Bean annotation inside the configuration class we can write something like this inside our **CollegeConfig.java** file. Please refer to the comments for a better understanding.

@Bean

// Here the method name is the

// bean id/bean name

public College collegeBean(){

// Return the College object

return new College();

}

**Implementation:**Below is the complete code for the **CollegeConfig.java** file that is below as follows:

*// Java Program to Illustrate Configuration in College Class*

**package** **BeanAnnotation**;

*// Importing required classes*

**import** **org.springframework.context.annotation.Bean**;

**import** **org.springframework.context.annotation.Configuration**;

@Configuration

**public** **class** **CollegeConfig** {

*// Using Bean annotation to create*

*// College class Bean*

@Bean

*// Here the method name is the*

*// bean id/bean name*

**public** College collegeBean()

{

*// Return the College object*

**return** **new** College();

}

}

**Spring - BeanPostProcessor**

pring Framework provides **BeanPostProcessor Interface**. It allows custom modification of new bean instances that are created by the **Spring Bean Factory.** If we want to implement some custom logic such as checking for marker interfaces or wrapping beans with proxies after the Spring container finishes instantiating, configuring, and initializing a bean, we can plug in BeanPostProcessor implementations.

* It works with all beans in the container.
* It can replace bean instances.
* It is used internally by Spring for features like **@Autowired** and **AOP.**

**Syntax:**

*org.springframework.beans.factory.config  
public interface BeanPostProcessor*

**Methods in the BeanPostProcessor Interface**

The BeanPostProcessor interface acts as a middleware in Spring's bean lifecycle, and it consists of two callback methods:

1. postProcessBeforeInitialization() Method (Before bean initialization)
2. postProcessAfterInitialization() Method (After bean initialization)

**Method 1: postProcessBeforeInitialization()**

To apply any custom logic to the given new bean instance before any bean initialization callbacks (like InitializingBean's afterPropertiesSet or a custom init-method), we can call this BeanPostProcessor method. The bean will already be populated with the property values, and the returned bean instance may be a wrapper around the original one.

**Syntax:**

*@Nullable  
  
default Object postProcessBeforeInitialization(Object bean, String beanName)   
throws BeansException*

**Parameters:**

* **bean:** The new bean instance.
* **beanName**: The name of the bean.

**Return Type**: Either the original or a wrapped bean instance to use. No subsequent BeanPostProcessors will be invoked, if null. And in case of errors, it throws BeansException.

**When it Runs:**

It runs after dependency injection, but before,

* **@PostConstruct** methods
* **InitializingBean.afterPropertiesSet()**
* Custom **init-method**

**Method 2: postProcessAfterInitialization()**

To apply any custom logic to the given new bean instance after any bean initialization callbacks (like InitializingBean's afterPropertiesSet or a custom init-method), we can call this BeanPostProcessor method.

The bean will already be populated with the property values and the returned bean instance may be a wrapper around the original one. This callback will be invoked for both the FactoryBean instance and the objects created by the FactoryBean. The post-processor can decide whether to apply to either the FactoryBean or created objects or both through the corresponding bean instance of the FactoryBean checks.

**Syntax:**

*@Nullable  
  
default Object postProcessAfterInitialization(Object bean, String beanName)  
throws BeansException*

**Parameters:**

* **bean:** The new bean instance
* **beanName:** The name of the bean

**Return Type:** Either the original or a wrapped bean instance to use. No subsequent BeanPostProcessors will be invoked, if null. And in case of errors, it throws BeansException.

It runs after all initialization callbacks complete.

**Registering and Ordering the BeanPostProcessors**

We can plug in one or more BeanPostProcessor implementations to include custom logic, in this case, we can control these multiple BeanPostProcessor instances by setting the order property or by implementing the Ordered interface.

An ApplicationContext can autodetect BeanPostProcessor beans in its bean definitions and apply those to any beans that are subsequently created. In case of a plain BeanFactory, we need to register the post-processors programmatically applying them to all beans created through the bean factory.

BeanPostProcessor beans that are autodetected in an ApplicationContext will be ordered according to PriorityOrdered and Ordered interfaces. In contrast, BeanPostProcessor beans that are registered programmatically with a BeanFactory will be applied in the order of registration.

**Spring - @PostConstruct and @PreDestroy Annotation**

**Spring @PostConstruct Annotation**

So generally, whenever we annotate a method in Spring Bean with @PostConstruct annotation, it gets executed after the spring bean is initialized. We can have only one method annotated with @PostConstruct annotation. This annotation is part of Common Annotations API and it’s part of JDK **module javax.annotation-api**.

**Spring @PreDestroy Annotation**

When we annotate a Spring Bean method with PreDestroy annotation, it gets called when the bean instance is getting removed from the context. Remember that if your [spring bean scope](https://www.geeksforgeeks.org/singleton-and-prototype-bean-scopes-in-java-spring/) is “prototype” then it’s not completely managed by the spring container and the PreDestroy method won’t get called. If there is a method named shutdown or close then the spring container will try to automatically configure them as callback methods when the bean is being destroyed.

**Adding @PostConstruct and @PreDestroy Annotation in the Project**

Developers, who are using Java 9+, both @PostConstruct and @PreDestroy annotations are part of Java EE. And since Java EE has been deprecated in Java 9 and removed in Java 11 we have to add an additional dependency to use these annotations:

**A.**If you are using Maven

<dependency>  
 <groupId>javax.annotation</groupId>  
 <artifactId>javax.annotation-api</artifactId>  
 <version>1.3.2</version>  
</dependency>