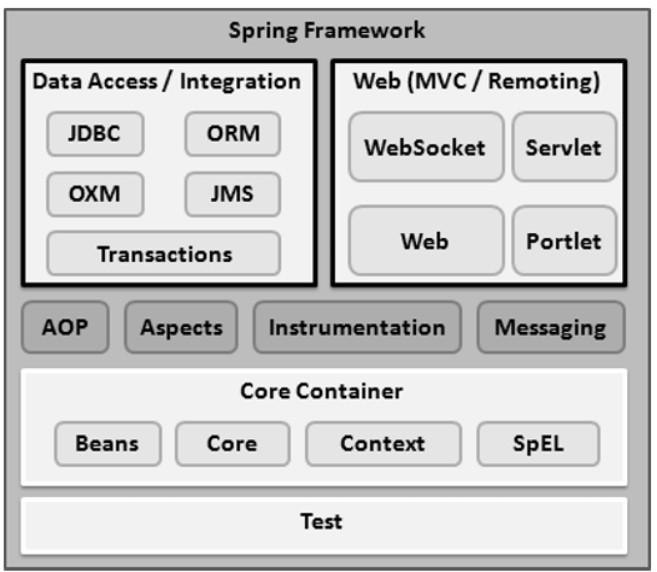
## **Benefits of Using the Spring Framework**

Following is the list of few of the great benefits of using Spring Framework −

* Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.
* Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about the ones you need and ignore the rest.
* Spring does not reinvent the wheel, instead it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and other view technologies.
* Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBeanstyle POJOs, it becomes easier to use dependency injection for injecting test data.
* Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.
* Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
* Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.
* Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).



## **Core Container**

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows −

* The **Core** module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
* The **Bean** module provides BeanFactory, which is a sophisticated implementation of the factory pattern.
* The **Context** module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
* The **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

## **Data Access/Integration**

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows −

* The **JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
* The **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
* The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
* The Java Messaging Service **JMS** module contains features for producing and consuming messages.
* The **Transaction** module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

## **Web**

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows −

* The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
* The **Web-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.
* The **Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
* The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

## **Miscellaneous**

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows −

* The **AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
* The **Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework.
* The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
* The **Messaging** module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
* The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

## Inversion Of Control (IOC) and Dependency Injection

These are the design patterns that are used to remove dependency from the programming code. They make the code easier to test and maintain. Let's understand this with the following code:

**class** Employee{

Address address;

Employee(){

address=**new** Address();

}

}

In such case, there is dependency between the Employee and Address (tight coupling). In the Inversion of Control scenario, we do this something like this:

**class** Employee{

Address address;

Employee(Address address){

**this**.address=address;

}

}

Thus, IOC makes the code loosely coupled. In such case, there is no need to modify the code if our logic is moved to new environment.

In Spring framework, IOC container is responsible to inject the dependency. We provide metadata to the IOC container either by XML file or annotation.

## Advantage of Dependency Injection

* makes the code loosely coupled so easy to maintain
* makes the code easy to test

## Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

##### 1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

##### 2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

##### 3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

##### 4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

##### 5) Fast Development

The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

##### 6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as [JMS](https://www.javatpoint.com/jms-tutorial), [JDBC](https://www.javatpoint.com/java-jdbc), JPA and JTA.

##### 7) Declarative support

It provides declarative support for caching, validation, transactions and formatting

# Spring Modules

In Spring, those objects that form the backbone of your application and that are managed by the Spring IoC *container* are referred to as *beans*. A bean is simply an object that is instantiated, assembled and otherwise managed by a Spring IoC container; other than that, there is nothing special about a bean

<https://docs.spring.io/spring/docs/2.5.3/reference/beans.html#beans-factory-collaborators>

# Spring IoC Container

Spring IoC is the mechanism to achieve loose-coupling between Objects dependencies. To achieve loose coupling and dynamic binding of the objects at runtime, objects dependencies are injected by other assembler objects. Spring IoC container is the program that **injects** dependencies into an object and make it ready for our use

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:

1. BeanFactory
2. ApplicationContext

## Using BeanFactory

The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

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

BeanFactory factory=new XmlBeanFactory(resource);

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

## Using ApplicationContext

The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

ApplicationContext context =

    new ClassPathXmlApplicationContext("applicationContext.xml");

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

##### Difference between BeanFactory and the ApplicationContext

The org.springframework.beans.factory.BeanFactory and the org.springframework.context.ApplicationContext interfaces acts as the IoC container. The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.

*BeanFactory***loads beans on-demand, while***ApplicationContext***loads all beans at startup**. Thus, *BeanFactory* is lightweight as compared to *ApplicationContext*

*ApplicationContext*enhances *BeanFactory* in a more framework-oriented style and provides several features that are suitable for enterprise applications.

For instance, it **provides**[messaging (i18n or internationalization)](https://www.baeldung.com/spring-classpathxmlapplicationcontext#2-internationalization-with-messagesource) functionality, [event publication](https://www.baeldung.com/spring-events) functionality, **annotation-based dependency injection**, and **easy integration with Spring AOP features**.

Apart from this, the *ApplicationContext* supports almost all types of bean scopes, but the *BeanFactory* only supports two scopes — *Singleton* and *Prototype*. Therefore, it's always preferable to use *ApplicationContext*when building complex enterprise applications

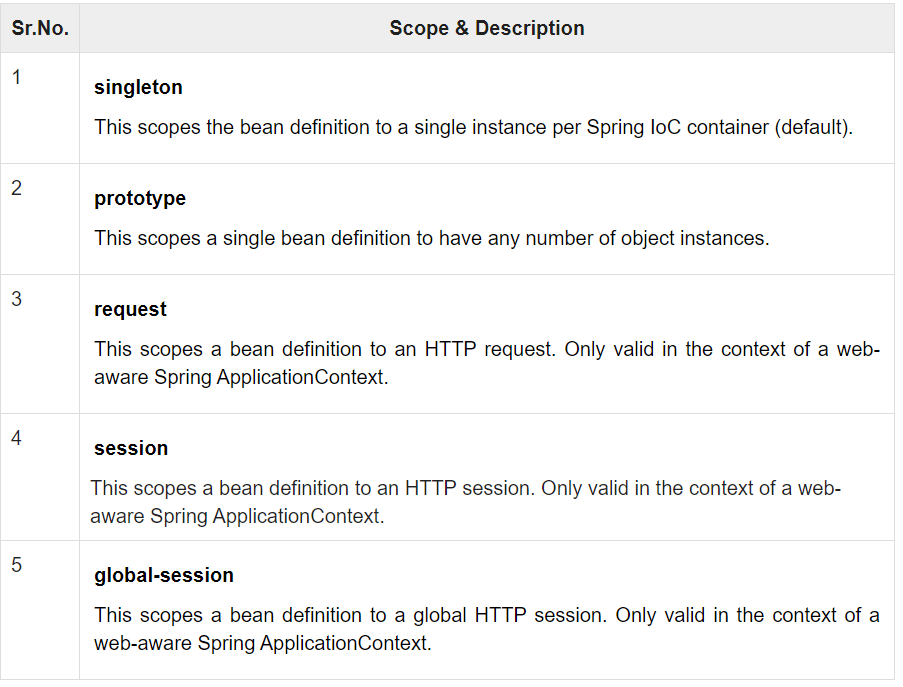
In short, the BeanFactory provides the configuration framework and basic functionality, while the ApplicationContext adds more enterprise-centric functionality to it.

# BEANS

|  |  |
| --- | --- |
| 1 | class  This attribute is mandatory and specifies the bean class to be used to create the bean. |
| 2 | name  This attribute specifies the bean identifier uniquely. In XMLbased configuration metadata, you use the id and/or name attributes to specify the bean identifier(s). |
| 3 | scope  This attribute specifies the scope of the objects created from a particular bean definition and it will be discussed in bean scopes chapter. |
| 4 | constructor-arg  This is used to inject the dependencies and will be discussed in subsequent chapters. |
| 5 | properties  This is used to inject the dependencies and will be discussed in subsequent chapters. |
| 6 | autowiring mode  This is used to inject the dependencies and will be discussed in subsequent chapters. |
| 7 | lazy-initialization mode  A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at the startup. |
| 8 | initialization method  A callback to be called just after all necessary properties on the bean have been set by the container. It will be discussed in bean life cycle chapter. |
| 9 | destruction method  A callback to be used when the container containing the bean is destroyed. It will be discussed in bean life cycle chapter. |

## Bean scopes

The Spring Framework supports the following five scopes, three of which are available only if you use a web-aware ApplicationContext.



## The singleton scope

If a scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

The default scope is always singleton. However, when you need one and only one instance of a bean, you can set the scope property to singleton in the bean configuration file, as shown in the following code snippet −

<!-- A bean definition with singleton scope -->

<bean id = "..." class = "..." scope = "singleton">

<!-- collaborators and configuration for this bean go here -->

</bean>

## The prototype scope

If the scope is set to prototype, the Spring IoC container creates a new bean instance of the object every time a request for that specific bean is made. As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.

To define a prototype scope, you can set the scope property to prototype in the bean configuration file, as shown in the following code snippet −

<!-- A bean definition with prototype scope -->

<bean id = "..." class = "..." scope = "prototype">

<!-- collaborators and configuration for this bean go here -->

</bean>

## Bean Life Cycle

##### Initialization callbacks

In the case of XML-based configuration metadata, you can use the init-method attribute to specify the name of the method that has a void no-argument signature. For example −

<bean id = "exampleBean" class = "examples.ExampleBean" init-method = "init"/>

Following is the class definition −

public class ExampleBean {

public void init() {

// do some initialization work

}

}

##### Destruction callbacks

In the case of XML-based configuration metadata, you can use the destroy-method attribute to specify the name of the method that has a void no-argument signature. For example −

<bean id = "exampleBean" class = "examples.ExampleBean" destroy-method = "destroy"/>

Following is the class definition −

public class ExampleBean {

public void destroy() {

// do some destruction work

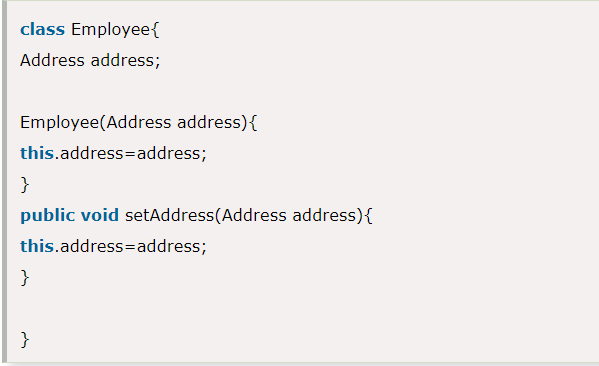
}

}

## Dependency Injection in Spring

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.

The Dependency Injection is a design pattern that removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing



In such case, instance of Address class is provided by external souce such as XML file either by constructor or setter method.

##### Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

* By Constructor
* By Setter method

# Dependency Injection by Constructor Example

We can inject the dependency by constructor. The <constructor-arg> subelement of <bean> is used for constructor injection. Here we are going to inject

1. primitive and String-based values
2. Dependent object (contained object)
3. Collection values etc.

# Constructor Injection with Collection Example

We can inject collection values by constructor in spring framework. There can be used three elements inside the constructor-arg element.

It can be:

1. list
2. set
3. map

Each collection can have string based and non-string based values.

# Inheriting Bean in Spring

By using the **parent** attribute of **bean**, we can specify the inheritance relation between the beans. In such case, parent bean values will be inherited to the current bean.

# Dependency Injection by setter method

We can inject the dependency by setter method also. The **<property>** subelement of **<bean>** is used for setter injection. Here we are going to inject

1. primitive and String-based values
2. Dependent object (contained object)
3. Collection values etc.

# Difference between constructor and setter injection

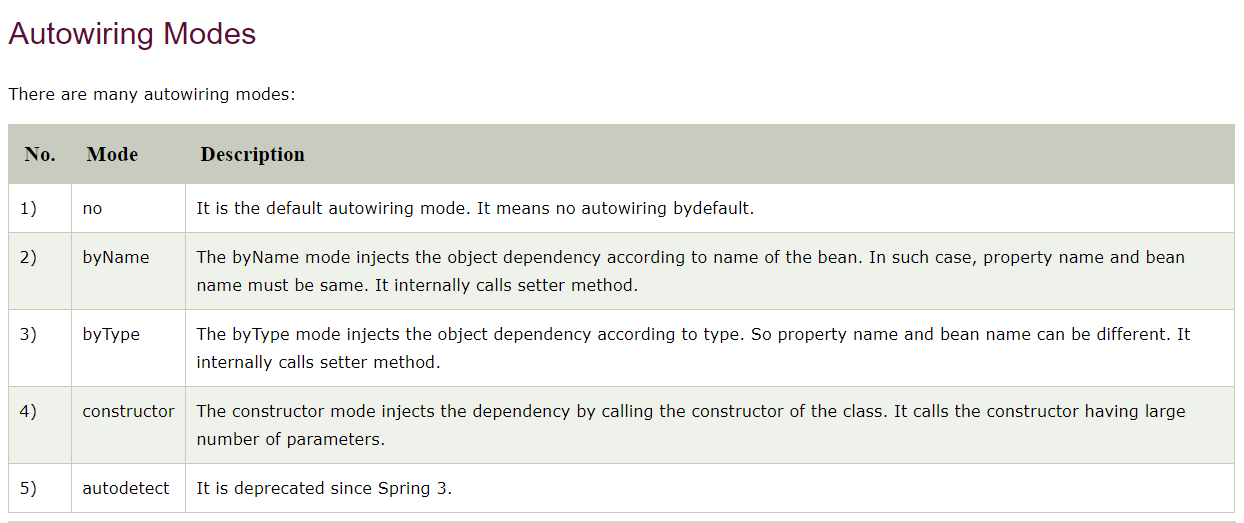
There are many key differences between constructor injection and setter injection.

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

# Autowiring in Spring

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.



## 1) byName autowiring mode

In case of byName autowiring mode, bean id and reference name must be same.

It internally uses setter injection.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

But, if you change the name of bean, it will not inject the dependency.

Let's see the code where we are changing the name of the bean from b to b1.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

## 2) byType autowiring mode

In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type.

It internally uses setter injection.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="byType"></bean>

In this case, it works fine because you have created an instance of B type. It doesn't matter that you have different bean name than reference name.

But, if you have multiple bean of one type, it will not work and throw exception.

Let's see the code where are many bean of type B.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="b2" **class**="org.sssit.B"></bean>
3. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

In such case, it will throw exception.

## 3) constructor autowiring mode

In case of constructor autowiring mode, spring container injects the dependency by highest parameterized constructor.

If you have 3 constructors in a class, zero-arg, one-arg and two-arg then injection will be performed by calling the two-arg constructor.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="constructor"></bean>

## 4) no autowiring mode

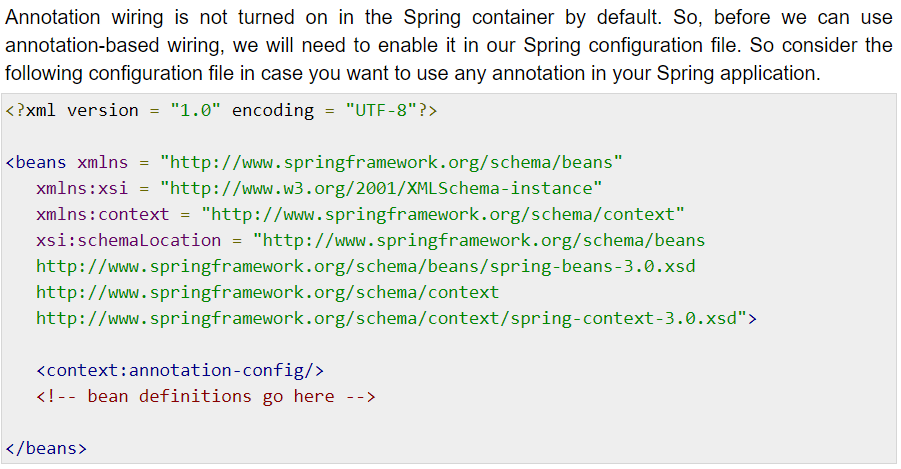
In case of no autowiring mode, spring container doesn't inject the dependency by autowiring.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="no"></bean>

## Dependency Injection with Factory Method in Spring

Check programs

# Annotation Based Configuration



## @Required Annotation

<https://www.tutorialspoint.com/spring/spring_required_annotation.htm>

## @Autowired

<https://www.tutorialspoint.com/spring/spring_autowired_annotation.htm>

The **@Autowired** annotation provides more fine-grained control over where and how autowiring should be accomplished. The @Autowired annotation can be used to autowire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.

## @Autowired on Setter Methods

You can use **@Autowired** annotation on setter methods to get rid of the <property> element in XML configuration file. When Spring finds an @Autowired annotation used with setter methods, it tries to perform **byType** autowiring on the method.

## @Autowired on Properties

You can use **@Autowired** annotation on properties to get rid of the setter methods. When you will pass values of autowired properties using <property> Spring will automatically assign those properties with the passed values or references. So with the usage of @Autowired on properties

## @Autowired on Constructors

You can apply @Autowired to constructors as well. A constructor @Autowired annotation indicates that the constructor should be autowired when creating the bean, even if no <constructor-arg> elements are used while configuring the bean in XML file. Let us check the following example.

## @Autowired with (required = false) option

By default, the @Autowired annotation implies the dependency is required similar to @Required annotation, however, you can turn off the default behavior by using **(required=false)** option with @Autowired.

## @Qualifier

There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property. In such cases, you can use the **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired.

<https://www.tutorialspoint.com/spring/spring_qualifier_annotation.htm>

# JSR-250 Annotations

Spring also supports JSR-250 based annotations which include @PostConstruct, @PreDestroy and @Resource annotations. Though these annotations are not really required because you already have other alternates, yet let us get a brief idea about them.

## @PostConstruct and @PreDestroy Annotations

To define the setup and teardown for a bean, we simply declare the <bean> with **init-method** and/or **destroy-method** parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, the destroy-method specifies a method that is called just before a bean is removed from the container.

You can use **@PostConstruct** annotation as an alternate of initialization callback and **@PreDestroy** annotation as an alternate of destruction callback as explained in the below example.

## @Resource

The @Resource annotation in spring performs the autowiring functionality. This annotation follows the autowire=byName semantics in the XML based configuration i.e. it takes the name attribute for the injection.

This annotation takes an optional name argument. In case no name attribute is specified with this annotation, the default name is interpreted from the field-name or the setter method (i.e. the bean property name). Always **remember** that if the @Resource annotation doesn’t find the bean with the name it will automatically switch it’s autowiring technique to autowire=byType (i.e. @Autowired annotation).

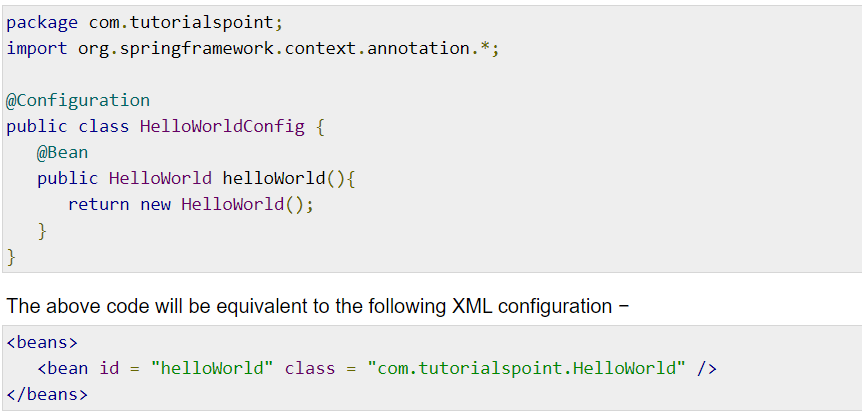
# Spring - Java Based Configuration

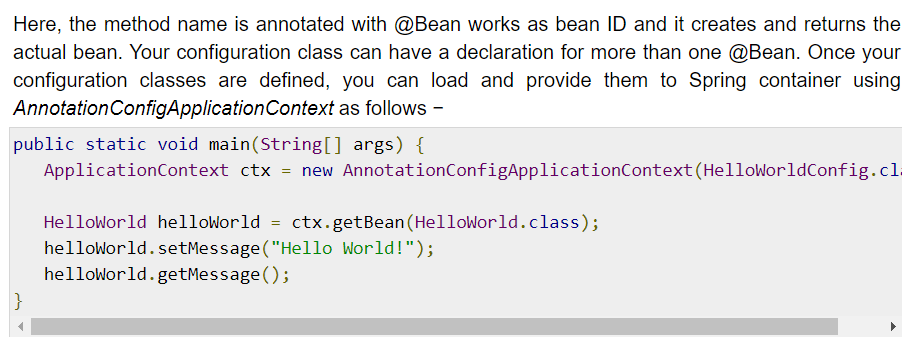
So far you have seen how we configure Spring beans using XML configuration file. If you are comfortable with XML configuration, then it is really not required to learn how to proceed with Java-based configuration as you are going to achieve the same result using either of the configurations available.

Java-based configuration option enables you to write most of your Spring configuration without XML

## @Configuration & @Bean Annotations

Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions. The **@Bean** annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context





## @Import

The **@Import** annotation allows for loading @Bean definitions from another configuration class. Consider a ConfigA class as follows −

@Configuration

public class ConfigA {

@Bean

public A a() {

return new A();

}

}

You can import above Bean declaration in another Bean Declaration as follows −

@Configuration

@Import(ConfigA.class)

public class ConfigB {

@Bean

public B b() {

return new B();

}

}

Now, rather than needing to specify both ConfigA.class and ConfigB.class when instantiating the context, only ConfigB needs to be supplied as follows −

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);

// now both beans A and B will be available...

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

## Lifecycle Callbacks

The @Bean annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's init-method and destroy-method attributes on the bean element −

public class Foo {

public void init() {

// initialization logic

}

public void cleanup() {

// destruction logic

}

}

@Configuration

public class AppConfig {

@Bean(initMethod = "init", destroyMethod = "cleanup" )

public Foo foo() {

return new Foo();

}

}

## Specifying Bean Scope

The default scope is singleton, but you can override this with the @Scope annotation as follows −

@Configuration

public class AppConfig {

@Bean

@Scope("prototype")

public Foo foo() {

return new Foo();

}

}

# JdbcTemplate

## **Problems of JDBC API**

The problems of JDBC API are as follows:

* We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
* We need to perform exception handling code on the database logic.
* We need to handle transaction.
* Repetition of all these codes from one to another database logic is a time consuming task.

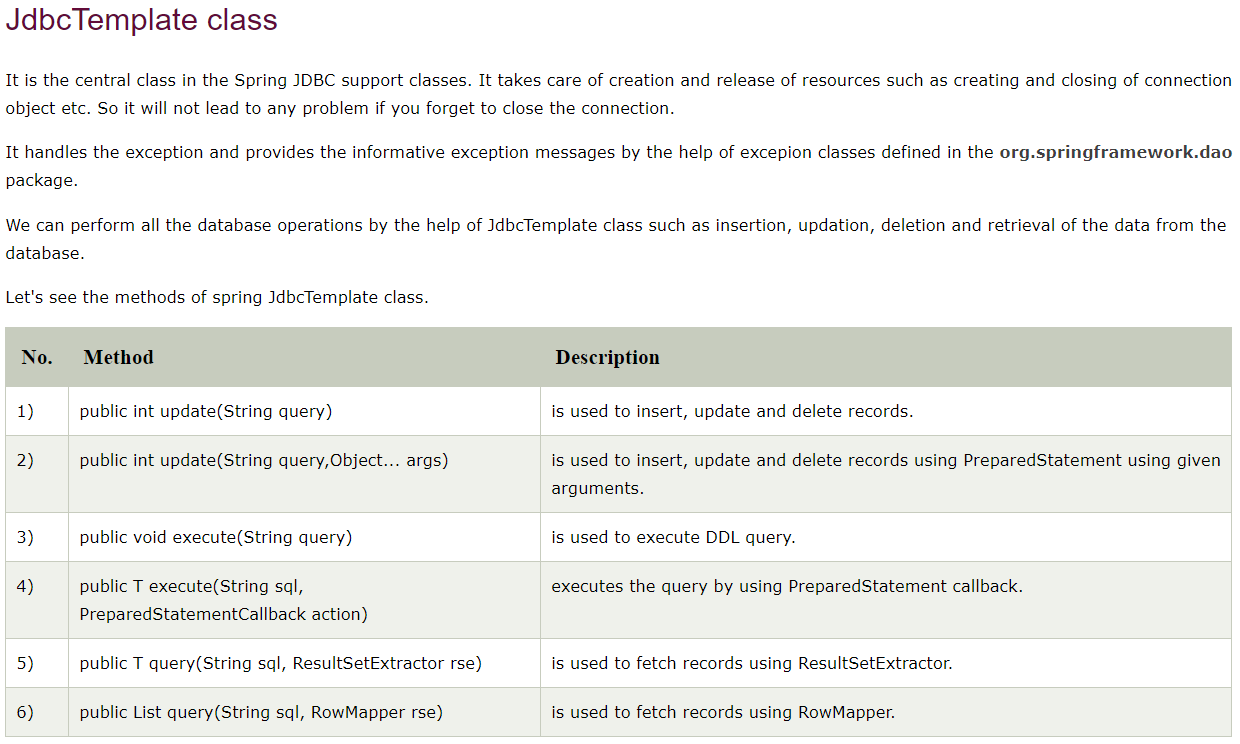
## **Advantage of Spring JdbcTemplate**

Spring JdbcTemplate eliminates all the above mentioned problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time.

## **Spring Jdbc Approaches**

Spring framework provides following approaches for JDBC database access:

* JdbcTemplate
* NamedParameterJdbcTemplate
* SimpleJdbcTemplate
* SimpleJdbcInsert and SimpleJdbcCall



## **JdbcTemplate Class**

The JDBC Template class executes SQL queries, updates statements, stores procedure calls, performs iteration over ResultSets, and extracts returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Instances of the *JdbcTemplate* class are *threadsafe* once configured. So you can configure a single instance of a *JdbcTemplate* and then safely inject this shared reference into multiple DAOs.

A common practice when using the JDBC Template class is to configure a *DataSource* in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes, and the JdbcTemplate is created in the setter for the DataSource.

# ResultSetExtractor

We can easily fetch the records from the database using **query()** method of **JdbcTemplate** class where we need to pass the instance of ResultSetExtractor.

#### Syntax of query method using ResultSetExtractor

1. **public** T query(String sql,ResultSetExtractor<T> rse)

## ResultSetExtractor Interface

**ResultSetExtractor** interface can be used to fetch records from the database. It accepts a ResultSet and returns the list.

# RowMapper

Like ResultSetExtractor, we can use RowMapper interface to fetch the records from the database using **query()** method of **JdbcTemplate** class. In the execute of we need to pass the instance of RowMapper now.

#### Syntax of query method using RowMapper

1. **public** T query(String sql,RowMapper<T> rm)

### RowMapper Interface

**RowMapper** interface allows to map a row of the relations with the instance of user-defined class. It iterates the ResultSet internally and adds it into the collection. So we don't need to write a lot of code to fetch the records as ResultSetExtractor.

#### Advantage of RowMapper over ResultSetExtractor

RowMapper saves a lot of code becuase it internally adds the data of ResultSet into the collection.

# Spring NamedParameterJdbcTemplate

Spring provides another way to insert data by named parameter. In such way, we use names instead of ?(question mark). So it is better to remember the data for the column.

## Simple example of named parameter query

1. insert into employee values (:id,:name,:salary)

# Spring - Transaction Management

Check below url for transaction management

<https://www.tutorialspoint.com/spring/spring_transaction_management.htm>

# Spring MVC