[http://www.journaldev.com/2696/spring-interview-questions-and-answers#spring-overview](http://www.journaldev.com/2696/spring-interview-questions-and-answers" \l "spring-overview)

**Spring IoC Containers**

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction. The Spring container uses dependency injection (DI) to manage the components that make up an application.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram is a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.



Spring provides following two distinct types of containers

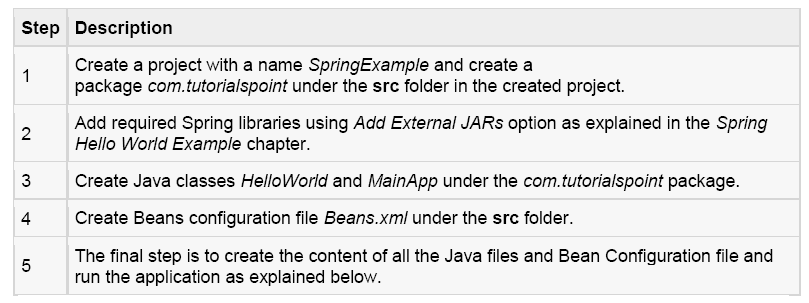
1. **Spring BeanFactory Container**

This is the simplest container providing basic support for DI and defined by the org.springframework.beans.factory.BeanFactory interface. The BeanFactory and related interfaces, such as **BeanFactoryAware**, **InitializingBean**, **DisposableBean**, are still present in spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with spring.

There are a number of implementations of the BeanFactory interface that come supplied straight out-of-the-box with Spring. The most commonly used BeanFactory implementation is the **XmlBeanFactory** class. This container reads the configuration metadata from an XML file and uses it to create a fully configured system or application.

The BeanFactory is usually preferred where the **resources are limited** like mobile devices or applet based applications. So use an ApplicationContext unless you have a good reason for not doing so.

**Example :**Let us have working Eclipse IDE in place and follow the following steps to create a Spring application:





**2). Spring ApplicationContext Container**

The Application Context is spring's more advanced container. Similar to BeanFactory it can load bean definitions, wire beans together and dispense beans upon request. Additionally it adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the *org.springframework.context.ApplicationContext* interface.

The ApplicationContext includes all functionality of the BeanFactory, it is generally recommended over the BeanFactory. BeanFactory can still be used for light weight applications like mobile devices or applet based applications.

The most commonly used ApplicationContext implementations are:

* **FileSystemXmlApplicationContext**: This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.
* **ClassPathXmlApplicationContext** This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.
* **WebXmlApplicationContext**: This container loads the XML file with definitions of all beans from within a web application.

We already have seen an example on ClassPathXmlApplicationContext container in Spring Hello World Example, and we will talk more about XmlWebApplicationContext in a separate chapter when we will discuss web based Spring applications. So let see one example on FileSystemXmlApplicationContext.

****

There are following two important points to note about the main program:

**1.** First step is to create factory object where we used framework API FileSystemXmlApplicationContext to create the factory bean after loading the bean configuration file from the given path. The FileSystemXmlApplicationContext() API takes care of creating and initializing all the objects ie. beans mentioned in the XML bean configuration file.

**2.** Second step is used to get required bean using getBean() method of the created context. This method uses bean ID to return a generic object which finally can be casted to actual object. Once you have object, you can use this object to call any class method.

**3.** Following is the content of the bean configuration file Beans.xml

****

**Spring Bean Definition**

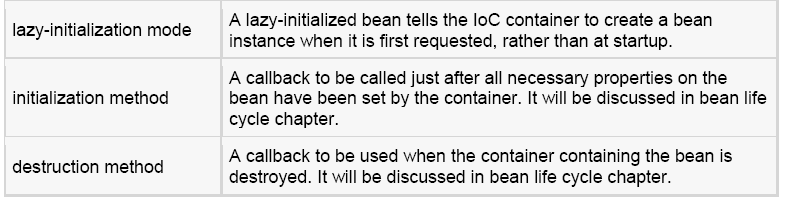
The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions which you have already seen in previous chapters.

The bean definition contains the information called **configuration metadata** which is needed for the container to know the followings:

* How to create a bean
* Bean's lifecycle details
* Bean's dependencies

All the above configuration metadata translates into a set of the following properties that make up each bean definition.

****

****

**Spring Configuration Metadata :**

Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written. There are following three important methods to provide configuration metadata to the Spring Container:

1. XML based configuration file.

2. Annotation-based configuration

3. Java-based configuration

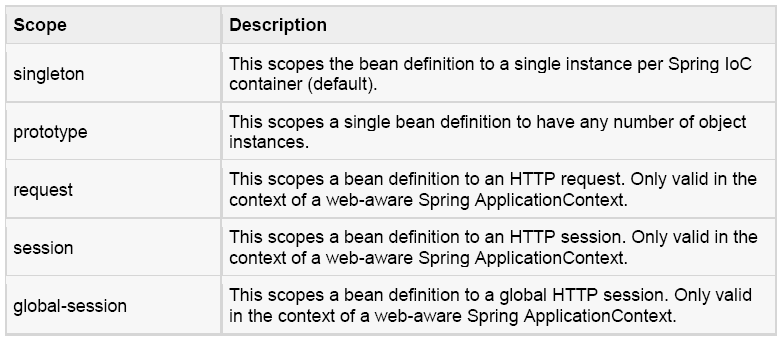
` Sample of XML based configuration file with different bean definitions including lazy initialization, initialization method and destruction method:

****

**Spring Bean Scopes**

When defining a <bean> in Spring, you have the option of declaring a scope for that bean. For example, To force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be **prototype**. Similar way if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be **singleton**.

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

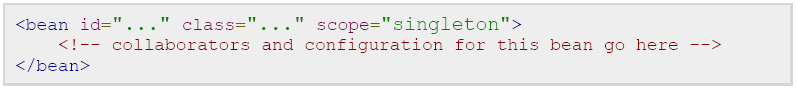
****

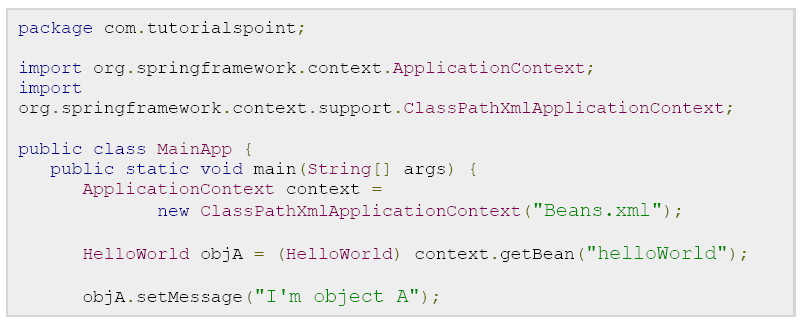
**The singleton scope:**

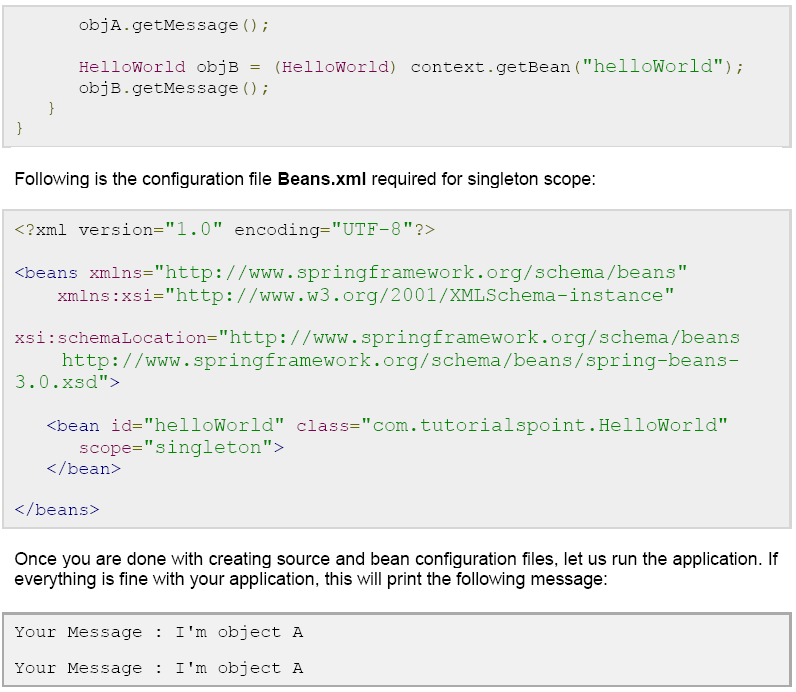
If scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

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 below:

****

****

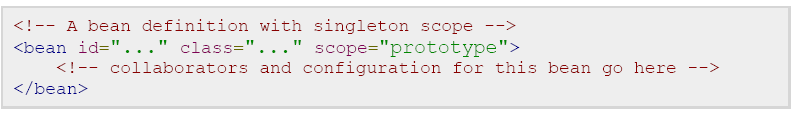
****

****

**The prototype scope :**

If scope is set to prototype, the Spring IoC container creates 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 below:

****

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

****

**Spring Bean Life Cycle**

The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.

Though, there are lists of the activities that take place behind the scenes between the time of bean Instantiation and its destruction, but this chapter will discuss only two important bean lifecycle callback methods which are required at the time of bean initialization and its destruction.

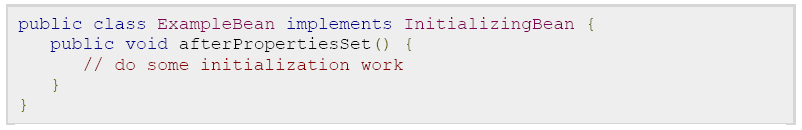
To define setup and teardown for a bean, we simply declare the <bean> with **init-method** and/or **destroy-method** parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroy-method specifies a method that is called just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

**Initialization callbacks:**

The *org.springframework.beans.factory.InitializingBean* interface specifies a single method:

****

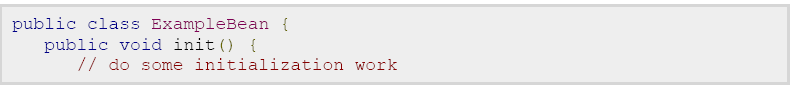
So you can simply implement above interface and initialization work can be done inside afterPropertiesSet() method as follows:

****

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:

****

Following is the class definition:

****

}

}

**Destruction callbacks:**

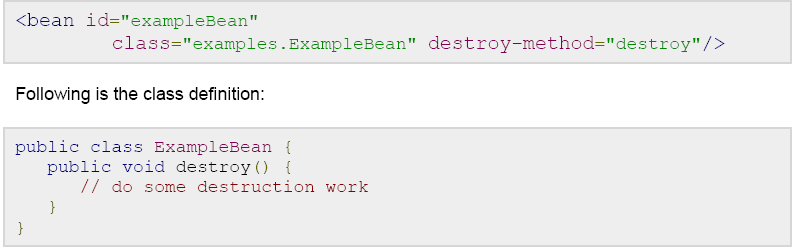
The *org.springframework.beans.factory.DisposableBean* interface specifies a single method:



So you can simply implement above interface and finalization work can be done inside destroy() method as follows:



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:



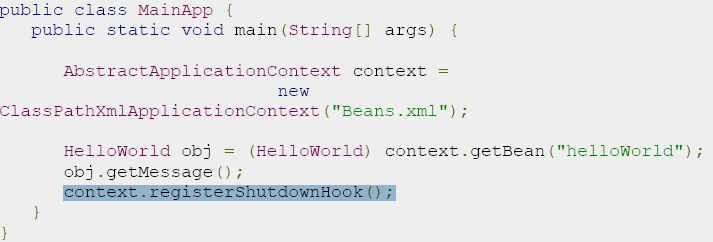
If you are using Spring's IoC container in a non-web application environment; for example, in a rich client desktop environment; you register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released.

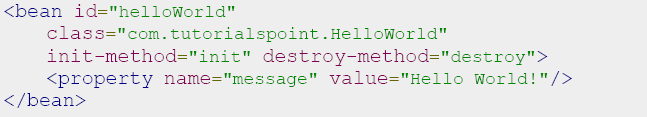
It is recommended that you do not use the InitializingBean or DisposableBean callbacks, because XML configuration gives much flexibility in terms of naming your method.

Example :

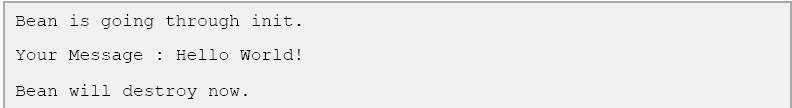


Following is the content of the **MainApp.java** file. Here you need to register a shutdown hook **registerShutdownHook()** method that is declared on the AbstractApplicationContext class. This will ensures a graceful shutdown and calls the relevant destroy methods.



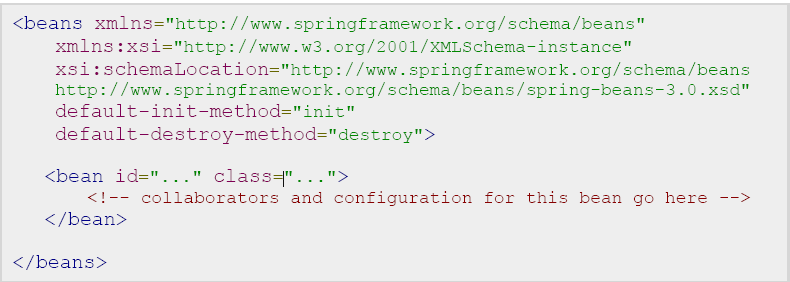


**Output:**

****

**Default initialization and destroy methods :**

If you have too many beans having initialization and or destroy methods with the same name, you don't need to declare **init-method** and **destroy-method** on each individual bean. Instead framework provides the flexibility to configure such situation using **default-init-method** and **default-destroy-method** attributes on the <beans> element as follows:



**Spring Bean Post Processors**

The **BeanPostProcessor** interface defines callback methods that you can implement to provide your own instantiation logic, dependency-resolution logic etc. You can also implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean by plugging in one or more BeanPostProcessor implementations.

You can configure multiple BeanPostProcessor interfaces and you can control the order in which these BeanPostProcessor interfaces execute by setting the **order** property provided the BeanPostProcessor implements the **Ordered** interface.

The BeanPostProcessors operate on bean (or object) instances which means that the Spring IoC container instantiates a bean instance and then BeanPostProcessor interfaces do their work.

An **ApplicationContext** automatically detects any beans that are defined with implementation of the **BeanPostProcessor** interface and registers these beans as post-processors, to be then called appropriately by the container upon bean creation.



This is very basic example of implementing BeanPostProcessor, which prints a bean name before and after initialization of any bean. You can implement more complex logic before and after instantiating a bean because you have access on bean object inside both the post processor methods.

Here is the content of **InitHelloWorld.java** file:



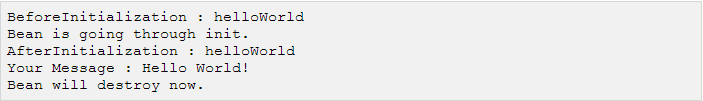
Following is the content of the **MainApp.java** file. Here you need to register a shutdown hook **registerShutdownHook()** method that is declared on the AbstractApplicationContext class. This will ensures a graceful shutdown and calls the relevant destroy methods.



Following is the configuration file **Beans.xml** required for init and destroy methods:



Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:



**Spring Bean Definition Inheritance**

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on.

A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.

Spring Bean definition inheritance has nothing to do with Java class inheritance but inheritance concept is same. You can define a parent bean definition as a template and other child beans can inherit required configuration from the parent bean.

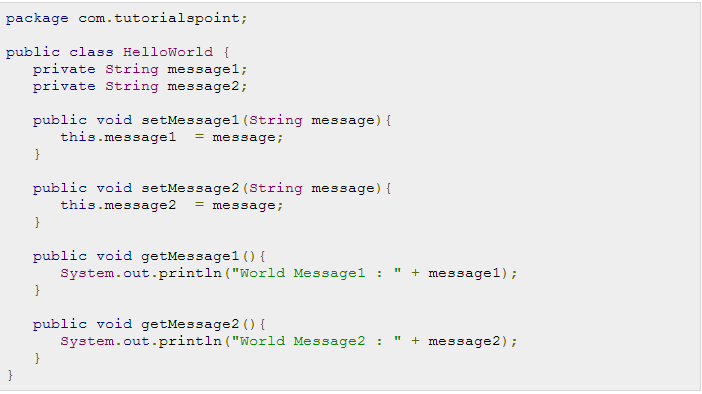
When you use XML-based configuration metadata, you indicate a child bean definition by using the **parent** attribute, specifying the parent bean as the value of this attribute.

## Example:

Following is the configuration file **Beans.xml** where we defined "helloWorld" bean which has two properties *message1* and *message2*. Next "helloIndia" bean has been defined as a child of "helloWorld" bean by using **parent** attribute. The child bean inherits *message2* property as is, and overrides *message1* property and introduces one more property *message3*.



Here is the content of **HelloWorld.java** file:



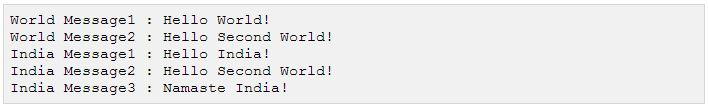
Here is the content of **HelloIndia.java** file:



Following is the content of the **MainApp.java** file:



Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:



If you observed here, we did not pass message2 while creating "helloIndia" bean, but it got passed because of Bean Definition Inheritance.

## Bean Definition Template:

You can create a Bean definition template which can be used by other child bean definitions without putting much effort. While defining a Bean Definition Template, you should **not** specify **class** attribute and should specify abstract attribute with a value of true as shown below:

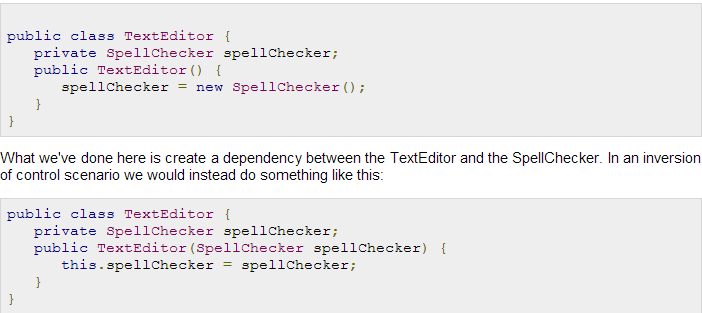


The parent bean cannot be instantiated on its own because it is incomplete, and it is also explicitly marked as *abstract*. When a definition is abstract like this, it is usable only as a pure template bean definition that serves as a parent definition for child definitions.

**Spring Dependency Injection**

Every java based application has a few objects that work together to present what the end-user sees as a working application. When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing. Dependency Injection (or sometime called wiring) helps in gluing/bonding these classes together and same time keeping them independent.

Consider you have an application which has a text editor component and you want to provide spell checking. Your standard code would look something like this:

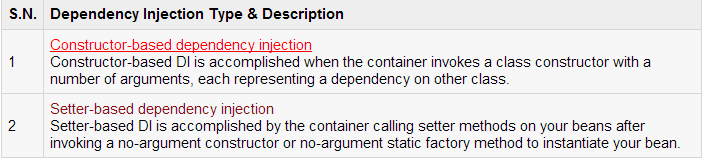
****

Here TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to TextEditor at the time of TextEditor instantiation and this entire procedure is controlled by the Spring Framework.

Here, we have removed the total control from TextEditor and kept it somewhere else (ie. XML configuration file) and the dependency ( ie. class SpellChecker) is being injected into the class TextEditor through a **Class Constructor**. Thus flow of control has been "inverted" by Dependency Injection (DI) because you have effectively delegated dependances to some external system.

Second method of injecting dependency is through **Setter Methods** of TextEditor class where we will create SpellChecker instance and this instance will be used to call setter methods to initialize TextEditor's properties.

Thus, DI exists in two major variants and following two sub-chapters will cover both of them with examples:



You can mix both, Constructor-based and Setter-based DI but it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.

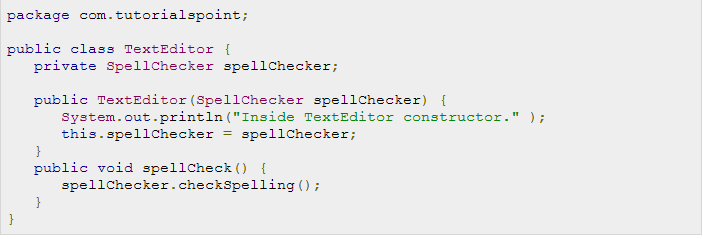
Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies, and does not know the location or class of the dependencies rather everything is taken care by the Spring Framework.

**Constructor-based Dependency Injection:**

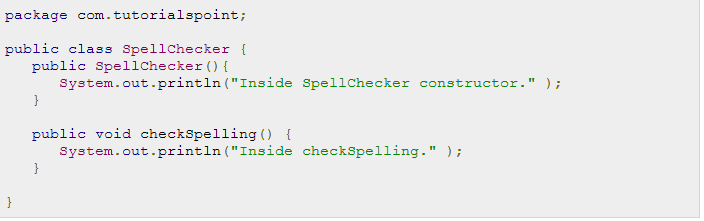
Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.

## Example:

Here is the content of **TextEditor.java** file:

****

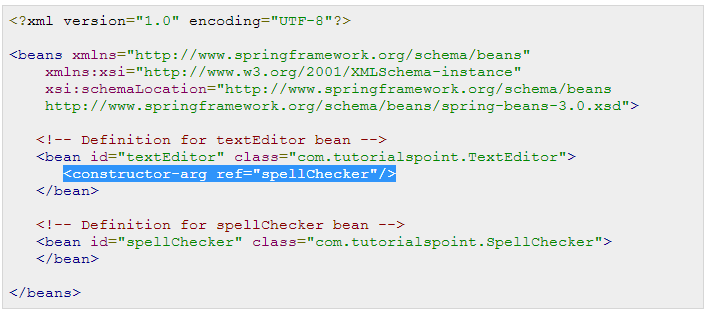
Following is the content of another dependent class file **SpellChecker.java**:



Following is the content of the **MainApp.java** file:



Following is the configuration file **Beans.xml** which has configuration for the constructor-based injection:

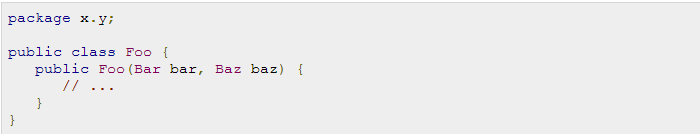


Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

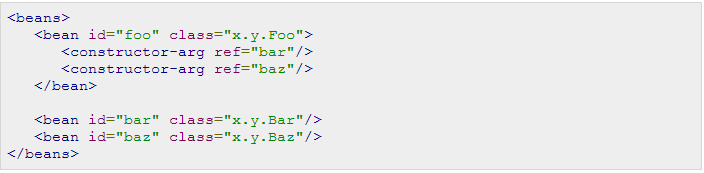
## 

## Constructor arguments resolution:

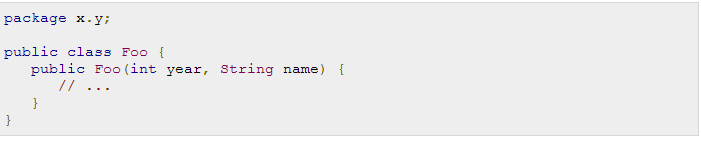
There may be a ambiguity exist while passing arguments to the constructor in case there are more than one parameters. To resolve this ambiguity, the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor. Consider the following class:



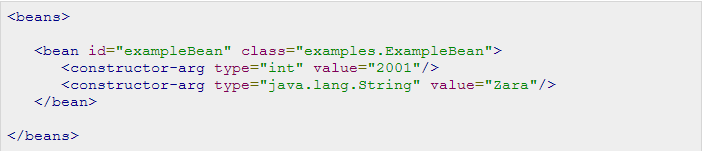
The following configuration works fine:



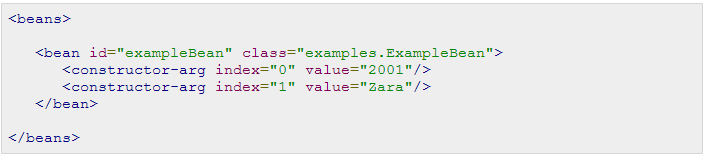
Let us check one more case where we pass different types to the constructor. Consider the following class:



The container can also use type matching with simple types if you explicitly specify the type of the constructor argument using the type attribute. For example:



Finally and the best way to pass constructor arguments, use the index attribute to specify explicitly the index of constructor arguments. Here the index is 0 based. For example:



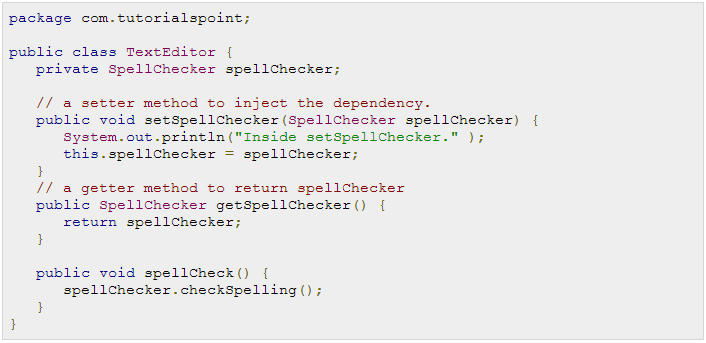
A final note, in case you are passing a reference to an object, you need to use **ref** attribute of <constructor-arg> tag and if you are passing a value directly then you should use **value** attribute as shown above.

**Setter-based Dependency Injection**

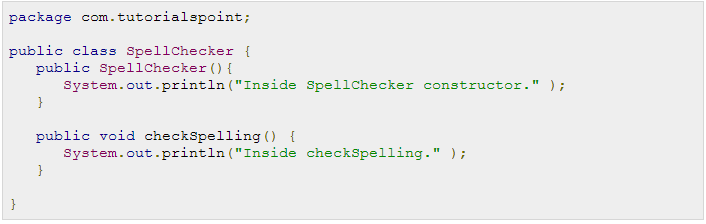
Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

## Example:

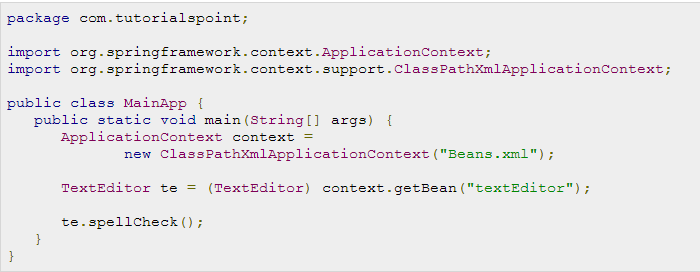
Here is the content of **TextEditor.java** file:

****

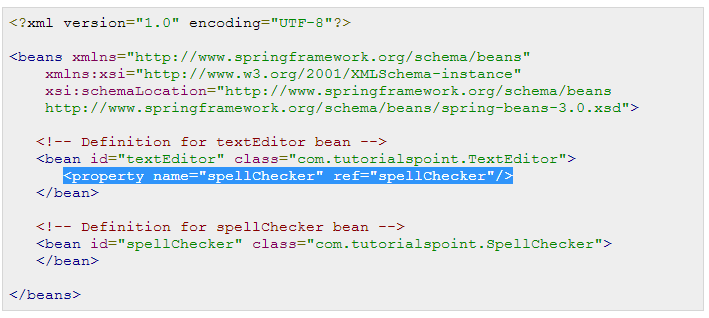
Here you need to check naming convention of the setter methods. To set a variable **spellChecker** we are using **setSpellChecker()** method which is very similar to Java POJO classes. Let us create the content of another dependent class file **SpellChecker.java**:

****

Following is the content of the **MainApp.java** file:

****

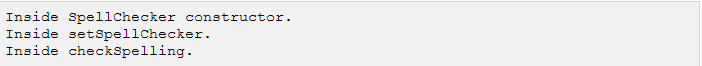
Following is the configuration file **Beans.xml** which has configuration for the setter-based injection:

****

You should note the difference in Beans.xml file defined in constructor-based injection and setter-based injection. The only difference is inside the <bean> element where we have used <constructor-arg> tags for constructor-based injection and <property> tags for setter-based injection.

Second important point to note is that in case you are passing a reference to an object, you need to use **ref**attribute of <property> tag and if you are passing a value directly then you should use **value** attribute.

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

****

## XML Configuration using p-namespace:

If you have many setter methods then it is convenient to use **p-namespace** in the XML configuration file. Let us check the difference:

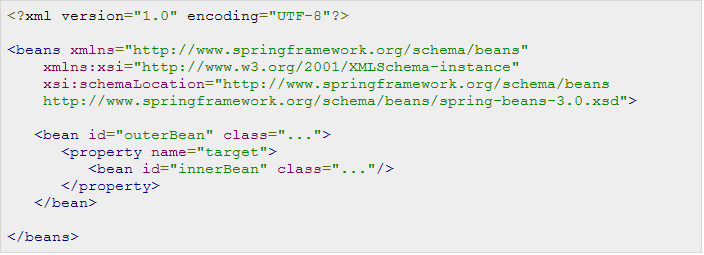
Let us take the example of a standard XML configuration file with <property> tags:

****

Here you should not the difference in specifying primitive values and object references with p-namespace. The **-ref** part indicates that this is not a straight value but rather a reference to another bean.

**Spring Injecting Inner Beans**

As you know Java inner classes are defined within the scope of other classes, similarly, **inner beans** are beans that are defined within the scope of another bean. Thus, a <bean/> element inside the <property/> or <constructor-arg/> elements is called inner bean and it is shown below.

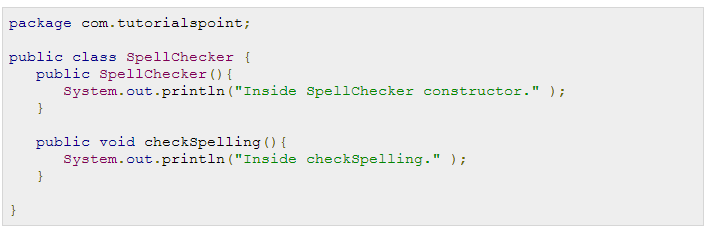
****

## Example:

Here is the content of **TextEditor.java** file:

****

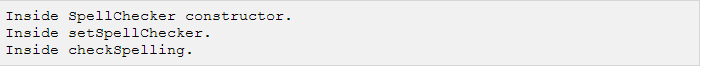
Following is the content of another dependent class file **SpellChecker.java**:

****

Following is the content of the **MainApp.java** file:

****

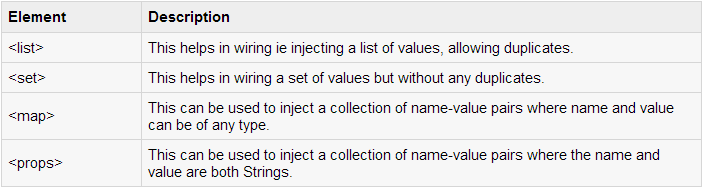
Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

****

**Spring Injecting Collection**

You have seen how to configure primitive data type using **value** attribute and object references using **ref**attribute of the <property> tag in your Bean configuration file. Both the cases deal with passing singular value to a bean.

Now what about if you want to pass plural values like Java Collection types List, Set, Map, and Properties. To handle the situation, Spring offers four types of collection configuration elements which are as follows:

****

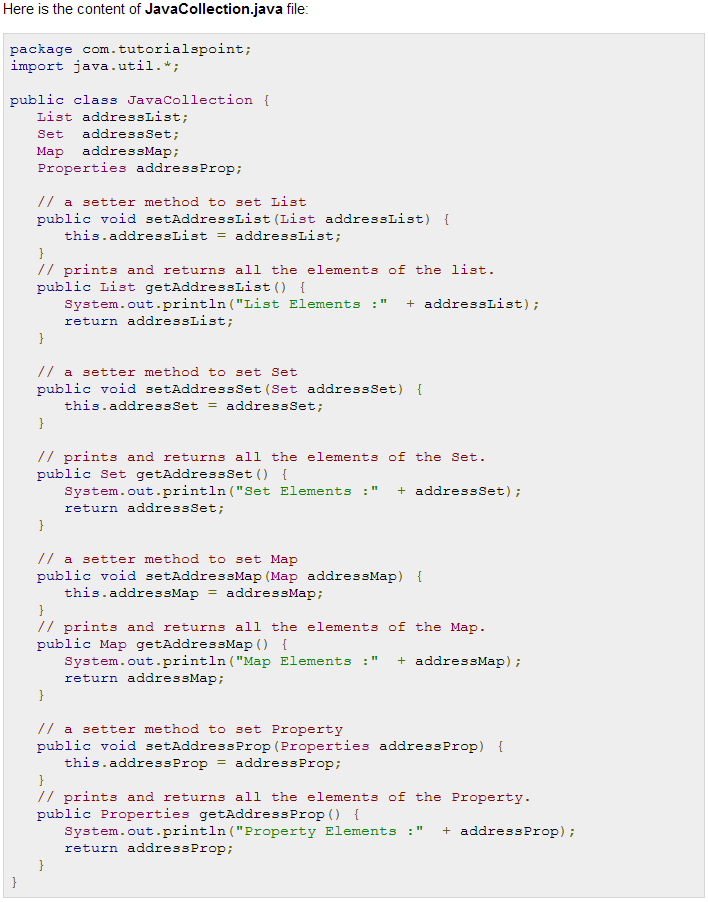
You can use either <list> or <set> to wire any implementation of java.util.Collection or an **array**.

You will come across two situations (a) Passing direct values of the collection and (b) Passing a reference of a bean as one of the collection elements.

## Example:

Let us have working Eclipse IDE in place and follow the following steps to create a Spring application:

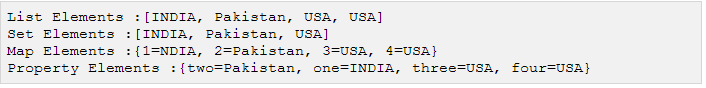
****

****

Following is the configuration file **Beans.xml** which has configuration for all the type of collections:

****

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

****

## Injecting Bean References:

Following Bean definition will help you understand how to inject bean references as one of the collection's element. Even you can mix references and values all together as shown below:

****

To use above bean definition, you need to define your setter methods in such a way that they should be able to handle references as well.

## Injecting null and empty string values

If you need to pass an empty string as a value then you can pass it as follows:

****

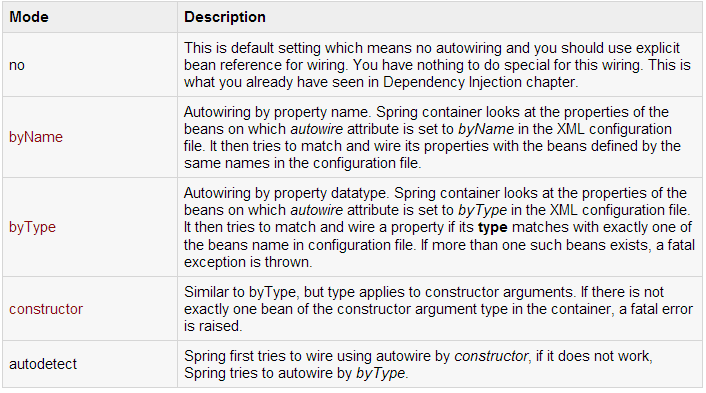
**Spring Beans Auto-Wiring**

You have learnt how to declare beans using the <bean> element and inject <bean> with using <constructor-arg> and <property> elements in XML configuration file.

The Spring container can **autowire** relationships between collaborating beans without using <constructor-arg> and <property> elements which helps cut down on the amount of XML configuration you write for a big Spring based application.

## Autowiring Modes:

There are following autowiring modes which can be used to instruct Spring container to use autowiring for dependency injection. You use the **autowire** attribute of the <bean/> element to specify autowire mode for a bean definition.



You can use **byType** or **constructor** autowiring mode to wire arrays and other typed-collections.

## Limitations with autowiring:

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions. Though, autowiring can significantly reduce the need to specify properties or constructor arguments but you should consider the limitations and disadvantages of autowiring before using them.



**Spring Autowiring 'byName':**

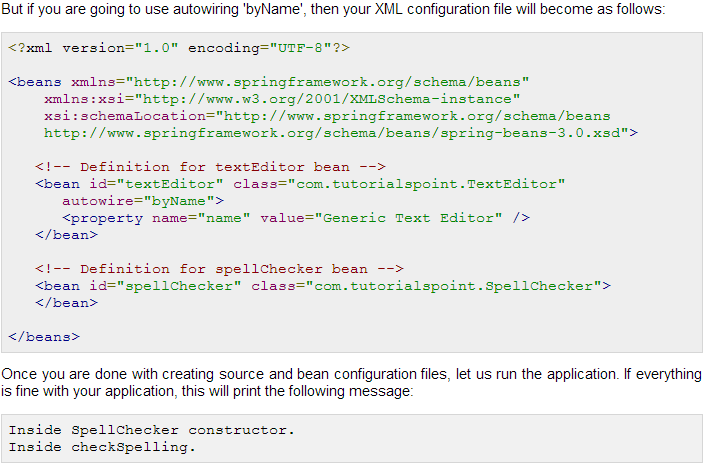
This mode specifies autowiring by property name. Spring container looks at the beans on which *auto-wire*attribute is set to *byName* in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file. If matches are found, it will inject those beans otherwise, it will throw exceptions.

For example, if a bean definition is set to autowire *byName* in configuration file, and it contains a*spellChecker* property (that is, it has a *setSpellChecker(...)* method), Spring looks for a bean definition named *spellChecker*, and uses it to set the property. Still you can wire remaining properties using <property> tags. Following example will illustrate the concept.

Let us have working Eclipse IDE in place and follow the following steps to create a Spring application:

****

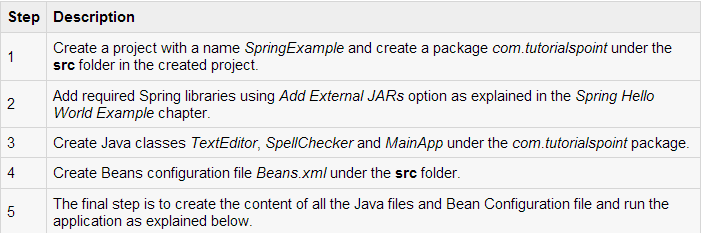
****

****

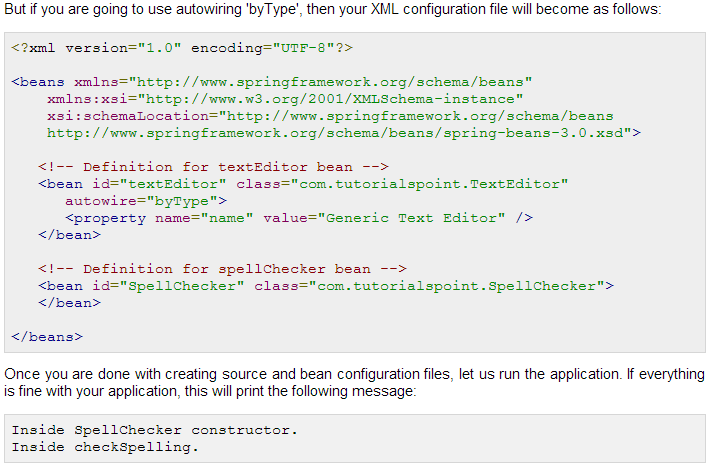
**Spring Autowiring 'byType':**

This mode specifies autowiring by property type. Spring container looks at the beans on which *autowire*attribute is set to *byType* in the XML configuration file. It then tries to match and wire a property if its **type**matches with exactly one of the beans name in configuration file. If matches are found, it will inject those beans otherwise, it will throw exceptions.

For example, if a bean definition is set to autowire *byType* in configuration file, and it contains a *spellChecker*property of *SpellChecker* type, Spring looks for a bean definition named *SpellChecker*, and uses it to set the property. Still you can wire remaining properties using <property> tags. Following example will illustrate the concept where you will find no difference with above example except XML configuration file has been changed. Let us have working Eclipse IDE in place and follow the following steps to create a Spring application:







**Spring Autowiring by “Constructor”:**

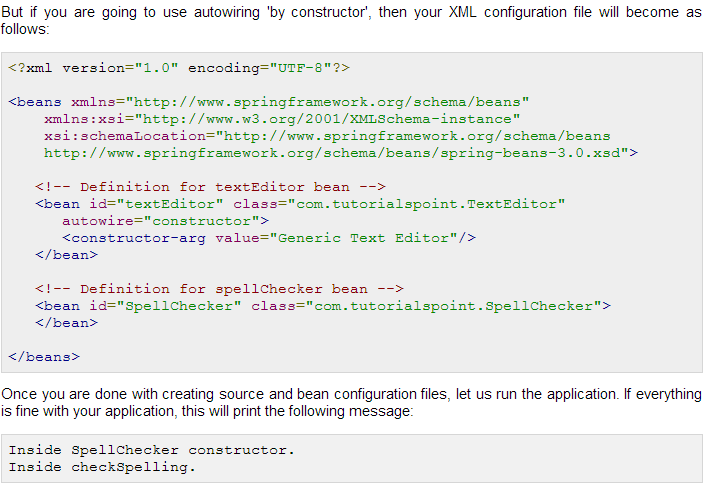
This mode is very similar to *byType*, but it applies to constructor arguments. Spring container looks at the beans on which *autowire* attribute is set to *constructor* in the XML configuration file. It then tries to match and wire its constructor's argument with exactly one of the beans name in configuration file. If matches are found, it will inject those beans otherwise, it will throw exceptions.

For example, if a bean definition is set to autowire by *constructor* in configuration file, and it has a constructor with one of the arguments of *SpellChecker* type, Spring looks for a bean definition named *SpellChecker*, and uses it to set the constructor's argument. Still you can wire remaining arguments using <constructor-arg> tags. Following example will illustrate the concept.

Let us have working Eclipse IDE in place and follow the following steps to create a Spring application:

****

****



**AOP with Spring Framework**

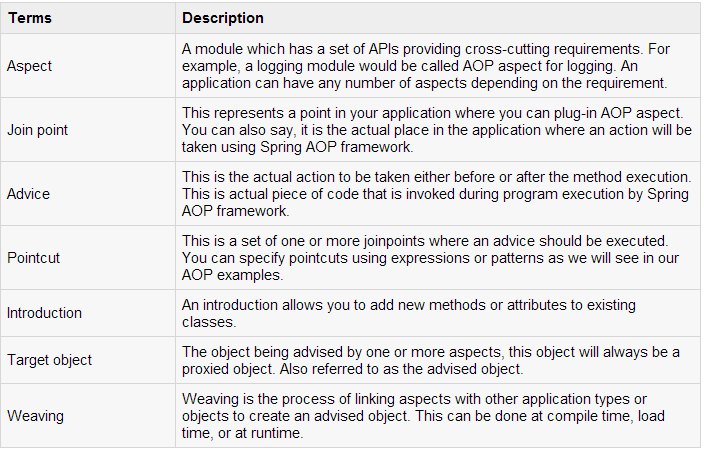
One of the key components of Spring Framework is the **Aspect oriented programming (AOP)** framework. Aspect Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like logging, auditing, declarative transactions, security, and caching etc.

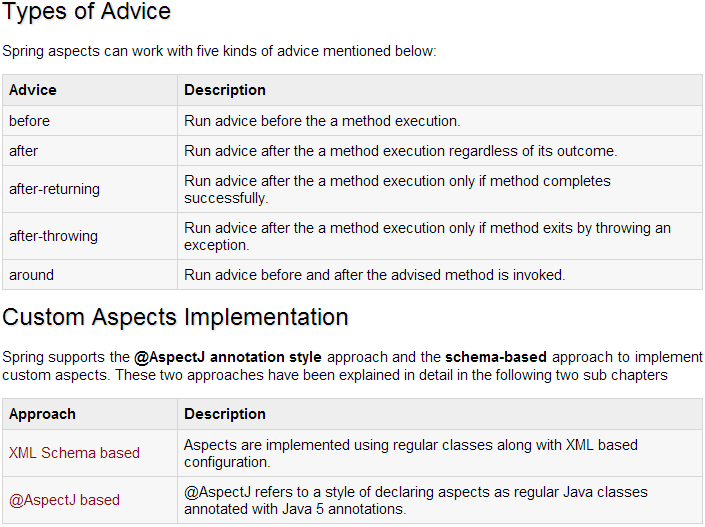
The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect. AOP is like triggers in programming languages such as Perl, .NET, Java and others.

Spring AOP module provides interceptors to intercept an application, for example, when a method is executed, you can add extra functionality before or after the method execution.

## *AOP Terminologies:*

Before we start working with AOP, let us become familiar with the AOP concepts and terminology. These terms are not specific to Spring, rather they are related to AOP.

****

****

**XML Schema Based AOP with Spring:**

To use the aop namespace tags described in this section, you need to import the spring-aop schema as described below:

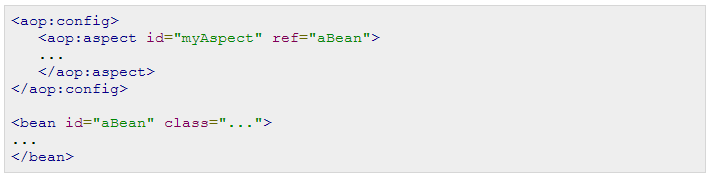


You will also need following AspectJ libraries on the CLASSPATH of your application. These libraries are available in the 'lib' directory of an AspectJ installation, otherwise you can download them from the internet.

* aspectjrt.jar
* aspectjweaver.jar
* aspectj.jar

## Declaring an aspect:

An **aspect** is declared using the **<aop:aspect>** element, and the backing bean is referenced using the **ref**attribute as follows:



Here "aBean" will be configured and dependency injected just like any other Spring bean as you have seen in previous chapters.

## Declaring a pointcut:

A **pointcut** helps in determining the join points (ie methods) of interest to be executed with different advices. While working with XML Schema based configuration, pointcut will be defined as follows:



## Declaring advices:

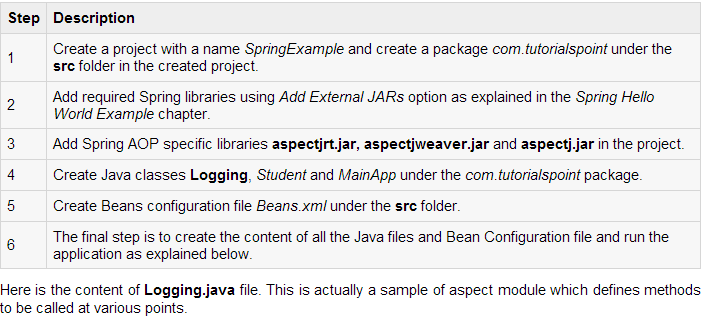
You can declare any of the five advices inside an <aop:aspect> using the <aop:{ADVICE NAME}> element as given below:

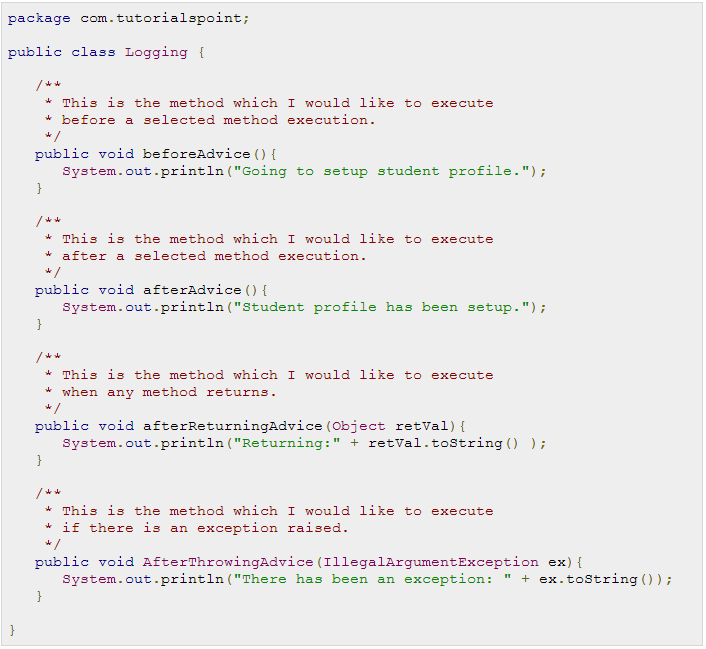


You can use same **doRequiredTask** or different methods for different advices. These methods will be defined as a part of aspect module.

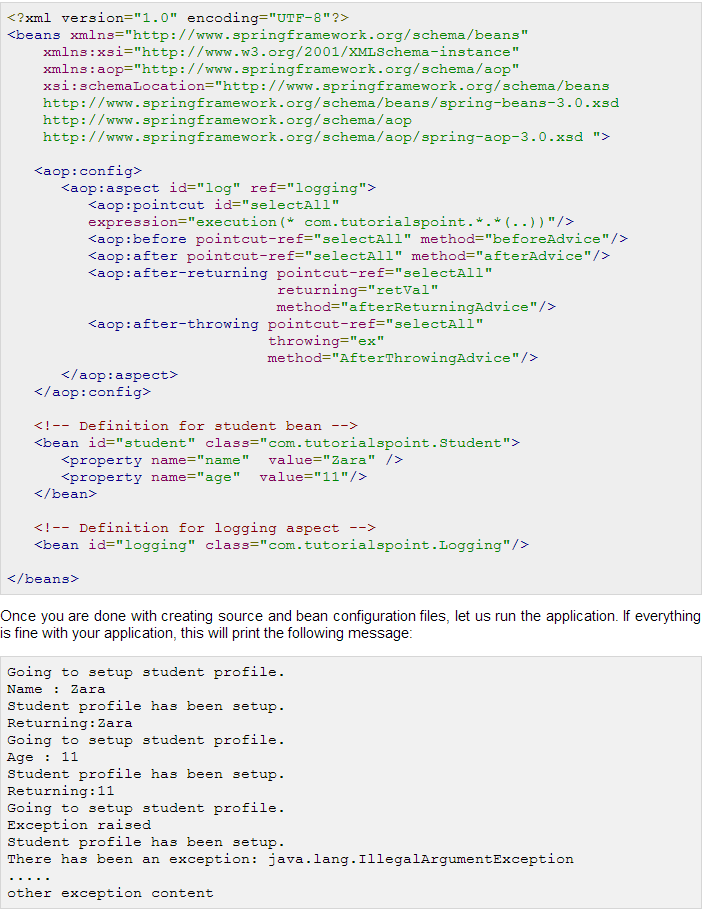
## XML Schema Based AOP Example

To understand above mentioned concepts related to XML Schema Based AOP, let us write an example which will implement few of the advices. To write our example with few advices, let us have working Eclipse IDE in place and follow the following steps to create a Spring application:









Let me explain that above defined <aop:pointcut> selects all the methods defined under the package com.tutorialspoint. Let us suppose, you want to execute your advice before or after a particular method, you can define your pointcut to narrow down your execution by replacing stars (\*) in pointcut definition with actual class and method names. Below is a modified XML configuration file to show the concept:



**@AspectJ Based AOP with Spring**

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. The @AspectJ support is enabled by including the following element inside your XML Schema-based configuration file.

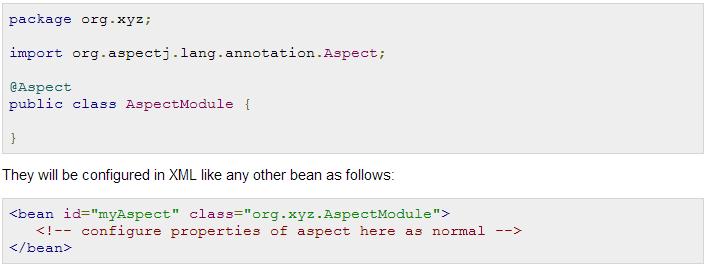
<aop:aspectj-autoproxy/>

You will also need following AspectJ libraries on the classpath of your application. These libraries are available in the 'lib' directory of an AspectJ installation, otherwise you can download them from the internet.

* aspectjrt.jar
* aspectjweaver.jar
* aspectj.jar

## Declaring an aspect:

Aspects classes are like any other normal bean and may have methods and fields just like any other class, except that they will be annotated with @Aspect as follows:

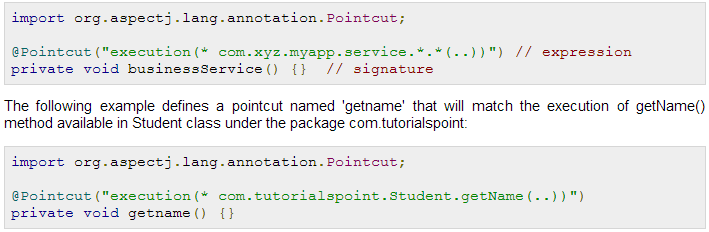


## Declaring a pointcut:

A **pointcut** helps in determining the join points (ie methods) of interest to be executed with different advices. While working with @AspectJ based configuration, pointcut declaration has two parts:

* A pointcut expression that determines exactly which method executions we are interested in.
* A pointcut signature comprising a name and any number of parameters. The actual body of the method is irrelevant and in fact should be empty.

The following example defines a pointcut named 'businessService' that will match the execution of every method available in the classes under the package com.xyz.myapp.service:



## Declaring advices:

You can declare any of the five advices using @{ADVICE-NAME} annotations as given below. This assumes that you already have defined a pointcut signature method businessService():



## @AspectJ Based AOP Example

To understand above mentioned concepts related to @AspectJ based AOP, let us write an example which will implement few of the advices. To write our example with few advices, let us have working Eclipse IDE in place and follow the following steps to create a Spring application:









**Spring JDBC Framework Overview**

While working with database using plain old JDBC, it becomes cumbersome to write unnecessary code to handle exceptions, opening and closing database connections etc. But Spring JDBC Framework takes care of all the low-level details starting from opening the connection, prepare and execute the SQL statement, process exceptions, handle transactions and finally close the connection.

So what you have do is just define connection parameters and specify the SQL statement to be executed and do the required work for each iteration while fetching data from the database.

Spring JDBC provides several approaches and correspondingly different classes to interface with the database. I'm going to take classic and the most popular approach which makes use of **JdbcTemplate**class of the framework. This is the central framework class that manages all the database communication and exception handling.

## JdbcTemplate Class:

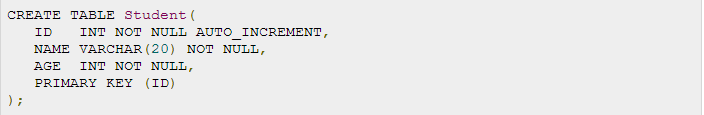
The JdbcTemplate class executes SQL queries, update statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the *org.springframework.dao* 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 JdbcTemplate 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.

## Configuring Data Source:

Let us create a database table **Student** in our database **TEST**. I assume you are working with MySQL database, if you work with any other database then you can change your DDL and SQL queries accordingly.



Now we need to supply a DataSource to the JdbcTemplate so it can configure itself to get database access. You can configure the DataSource in the XML file with a piece of code as shown below:

****

## Data Access Object (DAO):

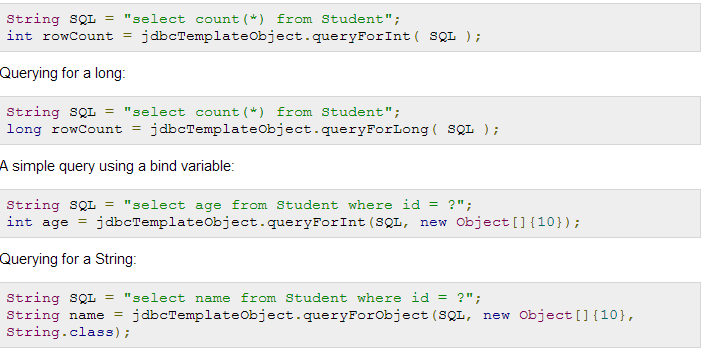
DAO stands for data access object which is commonly used for database interaction. DAOs exist to provide a means to read and write data to the database and they should expose this functionality through an interface by which the rest of the application will access them.

The Data Access Object (DAO) support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA or JDO in a consistent way.

## Executing SQL statements

Let us see how we can perform CRUD (Create, Read, Update and Delete) operation on database tables using SQL and jdbcTemplate object.

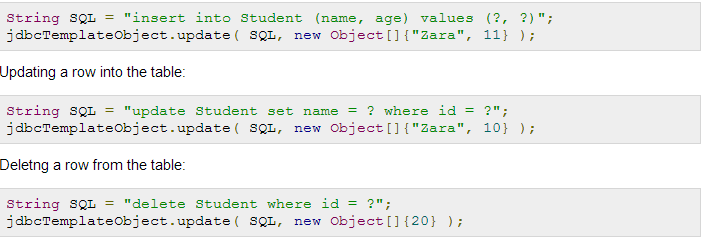
Querying for an integer:



Querying and returning an object

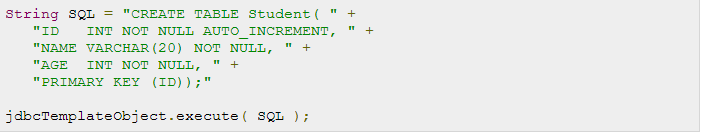


Inserting a row into the table:

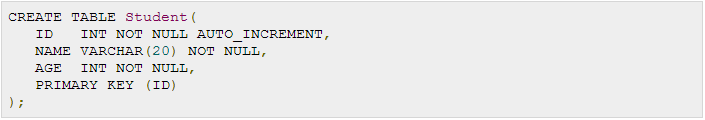


## Executing DDL Statements:

You can use the **execute(..)** method from *jdbcTemplate* to execute any SQL statements or DDL statements. Following is an example to use CREATE statement to create a table:



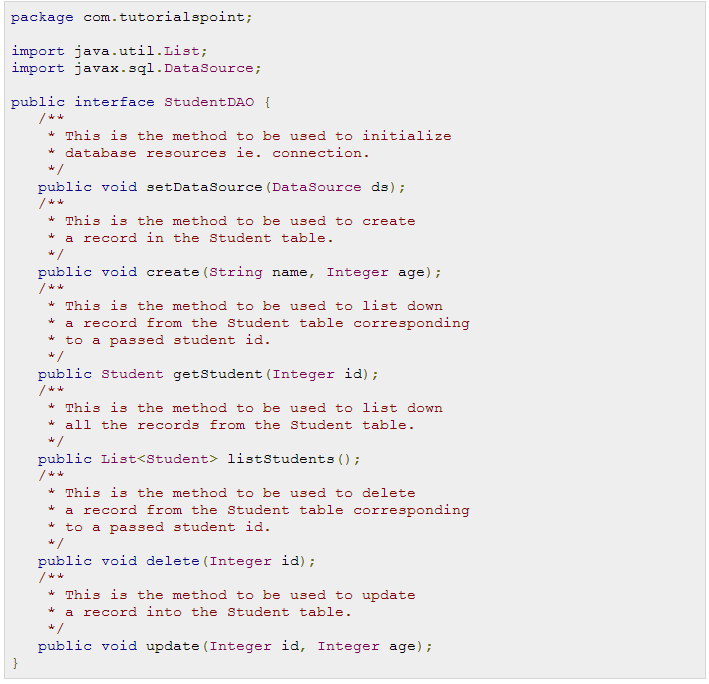
**Spring JDBC Example**

To understand the concepts related to Spring JDBC framework with JdbcTemplate class, let us write a simple example which will implement all the CRUD operations on the following Student table.  
****

Before proceeding, let us have working Eclipse IDE in place and follow the following steps to create a Spring application:

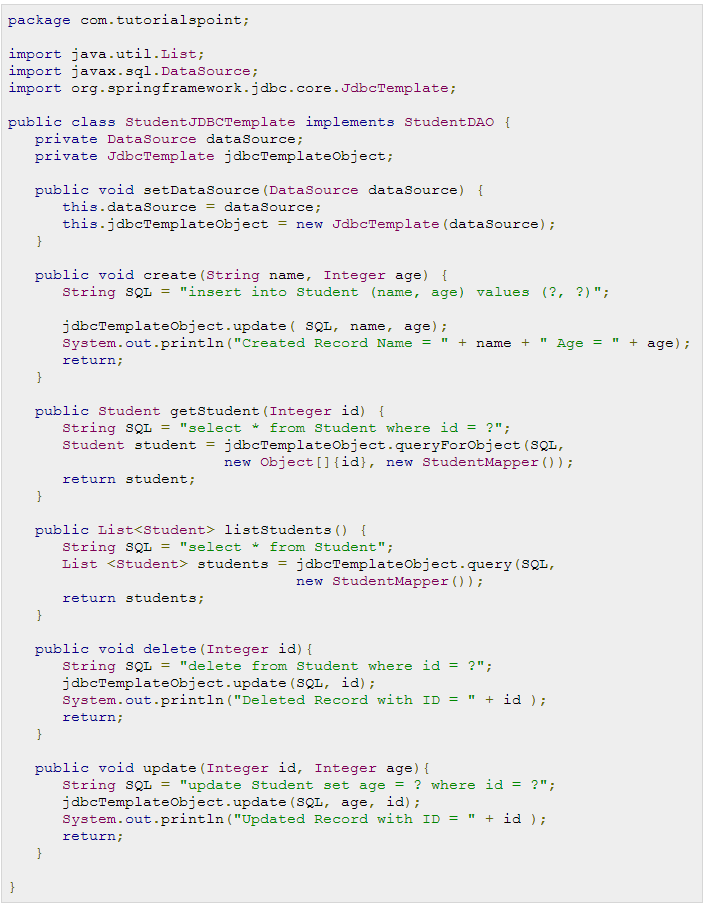
|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the**src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Add Spring JDBC specific latest libraries **mysql-connector-java.jar**,**org.springframework.jdbc.jar** and **org.springframework.transaction.jar** in the project. You can download required libraries if you do not have them already. |
| 4 | Create DAO interface *StudentDAO* and list down all the required methods. Though it is not required and you can directly write *StudentJDBCTemplate* class, but as a good practice, let's do it. |
| 5 | Create other required Java classes *Student*, *StudentMapper*, *StudentJDBCTemplate* and *MainApp*under the *com.tutorialspoint* package. |
| 6 | Make sure you already created **Student** table in TEST database. Also make sure your MySQL server is working fine and you have read/write access on the database using the give username and password. |
| 7 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 8 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Following is the content of the Data Access Object interface file **StudentDAO.java**:

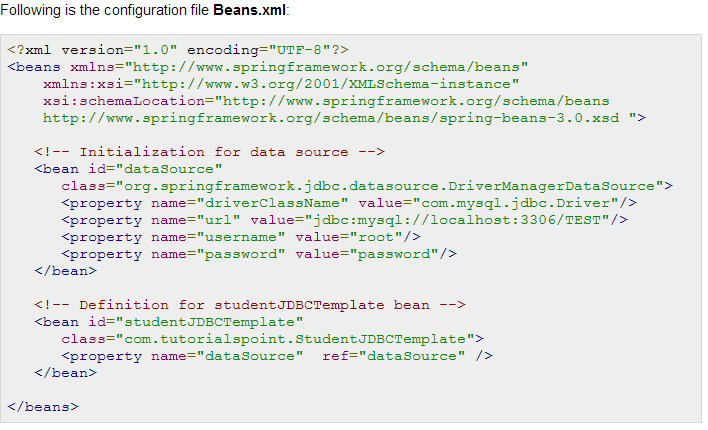
****

****

Following is the implementation class file **StudentJDBCTemplate.java** for the defined DAO interface StudentDAO:

****

****

****

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

------Records Creation--------

Created Record Name = Zara Age = 11

Created Record Name = Nuha Age = 2

Created Record Name = Ayan Age = 15

------Listing Multiple Records--------

ID : 1, Name : Zara, Age : 11

ID : 2, Name : Nuha, Age : 2

ID : 3, Name : Ayan, Age : 15

----Updating Record with ID = 2 -----

Updated Record with ID = 2

----Listing Record with ID = 2 -----

ID : 2, Name : Nuha, Age : 20

You can try delete operation yourself which I have not used in my example, but now you have one working application based on Spring JDBC framework which you can extend to add sophisticated functionality based on your project requirements. There are other approaches to access the database where you will use**NamedParameterJdbcTemplate** and **SimpleJdbcTemplate** classes, so if you are interested in learning these classes then kindly check reference manual for Spring Framework.

**SQL Stored Procedure in Spring**

The **SimpleJdbcCall** class can be used to call a stored procedure with IN and OUT parameters. You can use this approach while working with either of the RDBMS like Apache Derby, DB2, MySQL, Microsoft SQL Server, Oracle, and Sybase.

To understand the approach let us take our Student table which can be created in MySQL TEST database with the following DDL:

CREATE TABLE Student(

ID INT NOT NULL AUTO\_INCREMENT,

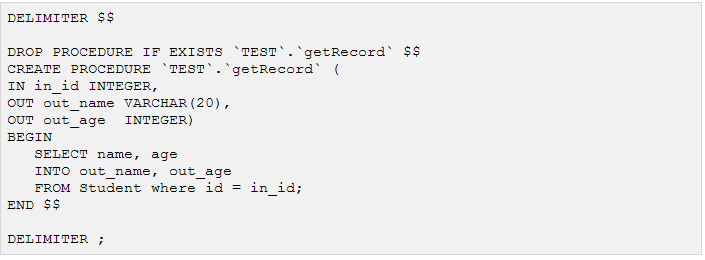
NAME VARCHAR(20) NOT NULL,

AGE INT NOT NULL,

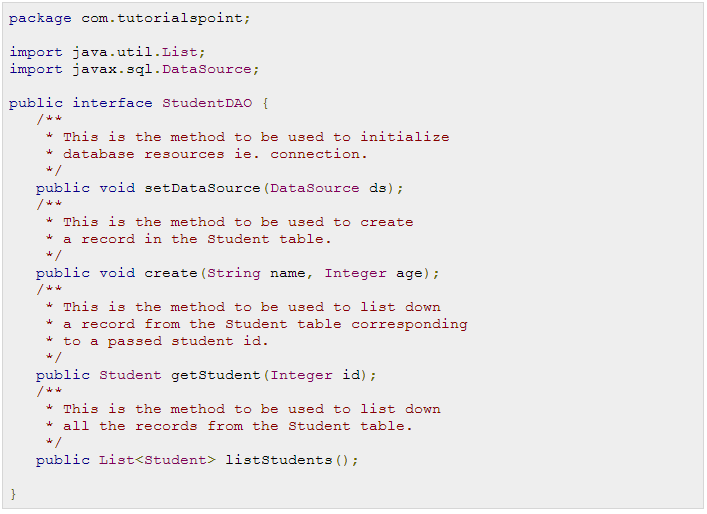
PRIMARY KEY (ID)

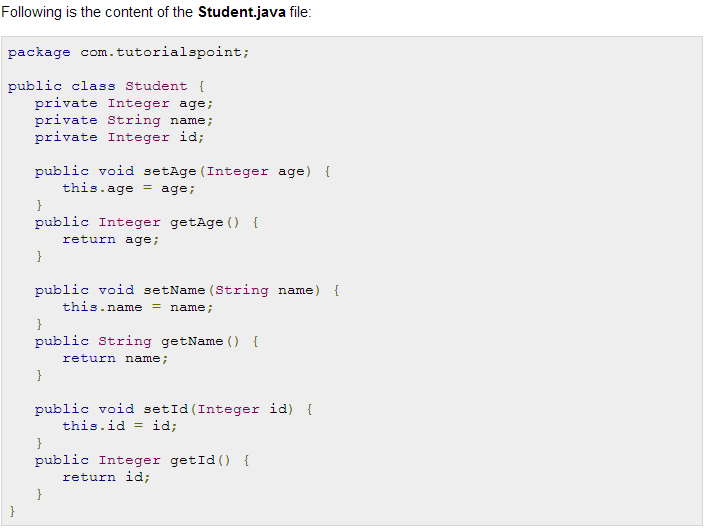
);

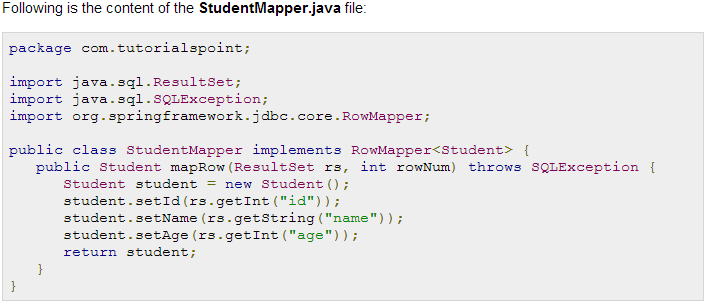
Next, consider the following MySQL stored procedure which takes student Id and returns corresponding student's name and age using OUT parameters. So let us create this stored procedure in your TEST database using MySQL command prompt:



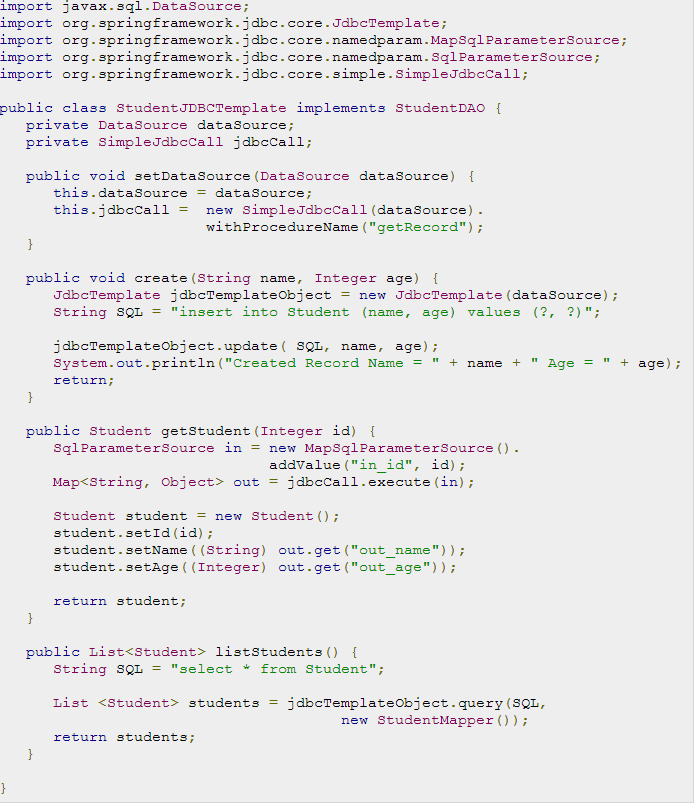
Now let us write our Spring JDBC application which will implement simple Create and Read operations on our Student table. Let us have working Eclipse IDE in place and follow the following steps to create a Spring application:







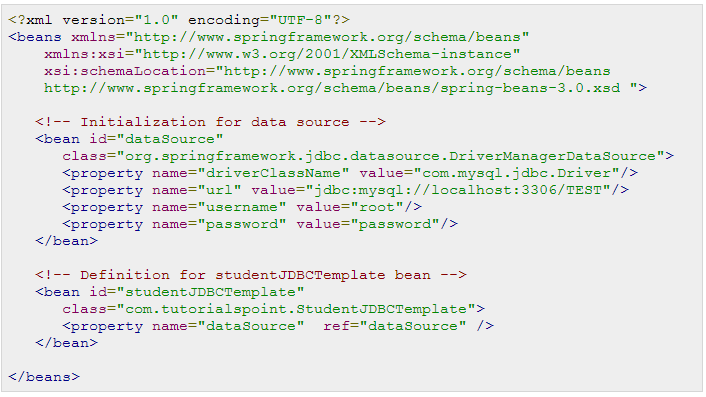
Following is the implementation class file **StudentJDBCTemplate.java** for the defined DAO interface StudentDAO:

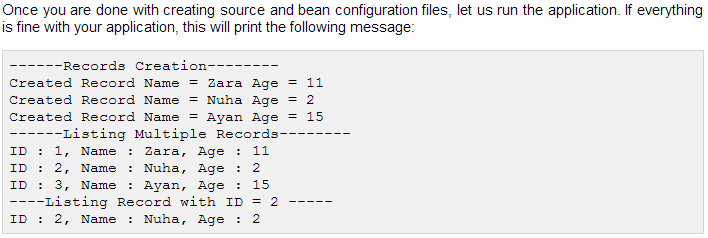


Few words about above program: The code you write for the execution of the call involves creating an*SqlParameterSource* containing the IN parameter. It's important to match the name provided for the input value with that of the parameter name declared in the stored procedure. The *execute* method takes the IN parameters and returns a Map containing any out parameters keyed by the name as specified in the stored procedure. Now let us move with the main application file **MainApp.java**, which is as follows:

****

Following is the configuration file **Beans.xml**:

****

****

**Spring Transaction Management**

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of and RDBMS oriented enterprise applications to ensure data integrity and consistency. The concept of transactions can be described with following four key properties described as **ACID**:

**Atomicity:** A transaction should be treated as a single unit of operation which means either the entire sequence of operations is successful or unsuccessful.

**Consistency:** This represents the consistency of the referential integrity of the database, unique primary keys in tables etc.

**Isolation:** There may be many transactions processing with the same data set at the same time, each transaction should be isolated from others to prevent data corruption.

**Durability:** Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

A real RDBMS database system will guarantee all the four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows:

* Begin the transaction using *begin transaction* command.
* Perform various deleted, update or insert operations using SQL queries.
* If all the operation are successful then perform *commit* otherwise *rollback* all the operations.

Spring framework provides an abstract layer on top of different underlying transaction management APIs. The Spring's transaction support aims to provide an alternative to EJB transactions by adding transaction capabilities to POJOs. Spring supports both programmatic and declarative transaction management. EJBs requires an application server, but Spring transaction management can be implemented without a need of application server.

## Local vs. Global Transactions:

Local transactions are specific to a single transactional resource like a JDBC connection, whereas global transactions can span multiple transactional resources like transaction in a distributed system.

Local transaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine. Local transactions are easier to be implemented.

Global transaction management is required in a distributed computing environment where all the resources are distributed across multiple systems. In such a case transaction management needs to be done both at local and global levels. A distributed or a global transaction is executed across multiple systems, and its execution requires coordination between the global transaction management system and all the local data managers of all the involved systems.

## Programmatic vs. Declarative:

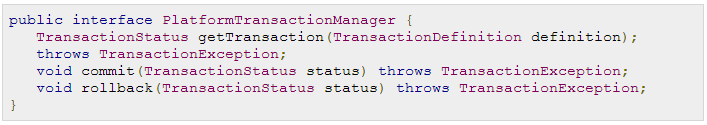
Spring supports two types of transaction management:

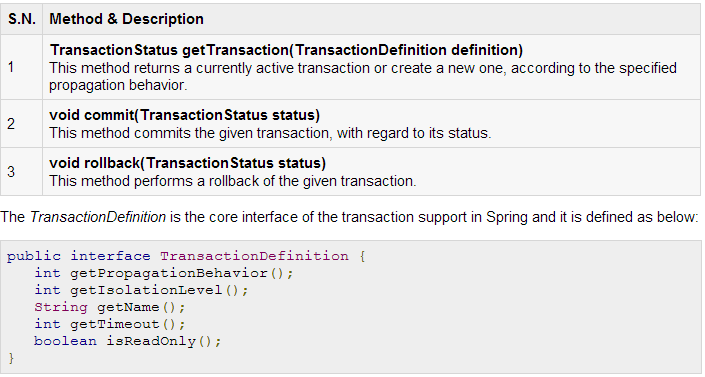
* [Programmatic transaction management:](http://www.tutorialspoint.com/spring/programmatic_management.htm) This means that you have manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
* [Declarative transaction management:](http://www.tutorialspoint.com/spring/declarative_management.htm) This means you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions.

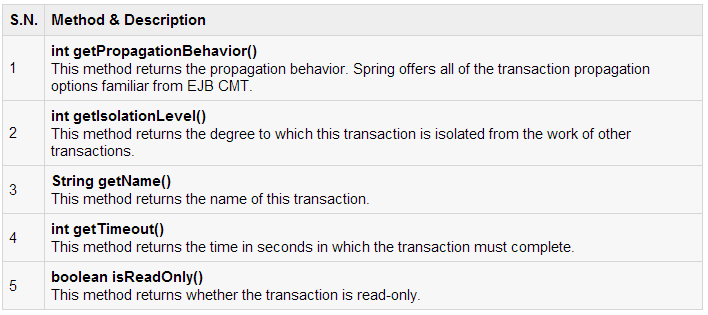
Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code. But as a kind of crosscutting concern, declarative transaction management can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework.

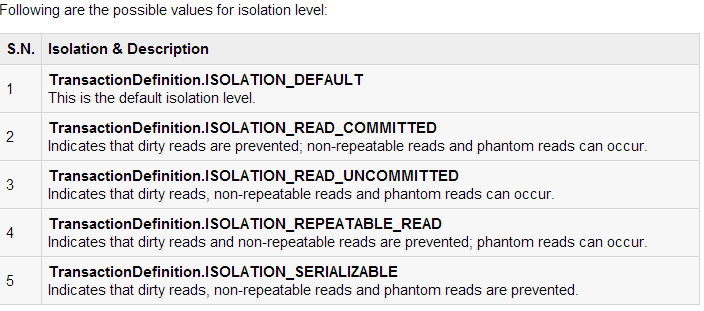
## Spring Transaction Abstractions:

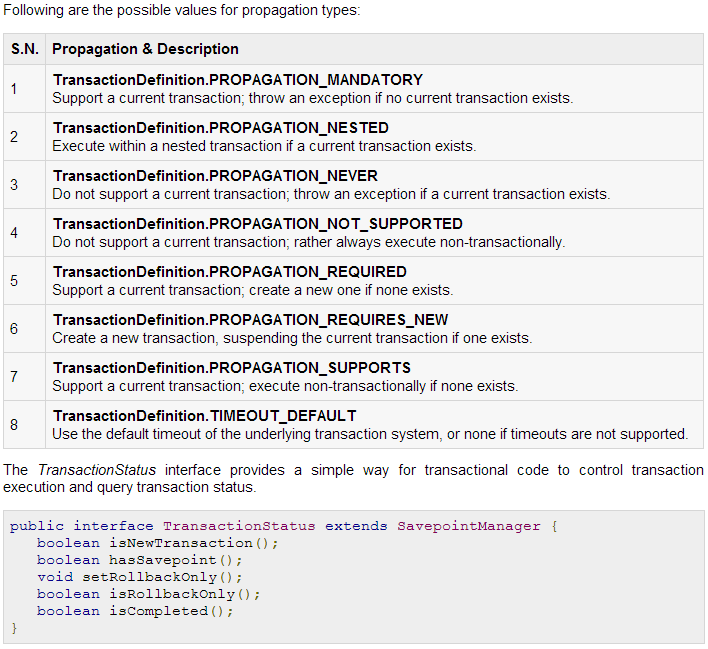
The key to the Spring transaction abstraction is defined by the*org.springframework.transaction.PlatformTransactionManager* interface, which is as follows:

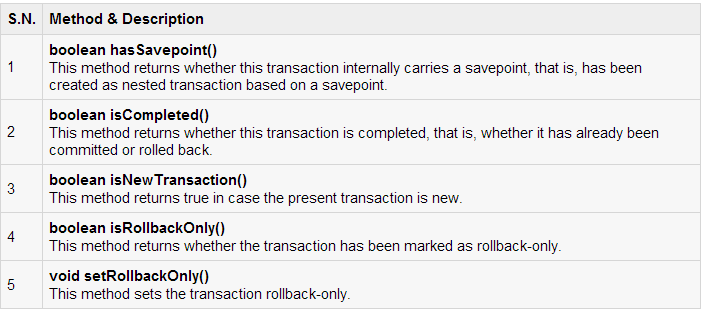
****

****

****

****

****

****

**Spring Programmatic Transaction Management**

Programmatic transaction management approach allows you to manage the transaction with the help of programming in your source code. That gives you extreme flexibility, but it is difficult to maintain.

Before we begin, it is important to have at least two database tables on which we can perform various CRUD operations with the help of transactions. Let us take **Student** table, which can be created in MySQL TEST database with the following DDL:

CREATE TABLE Student(

ID INT NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(20) NOT NULL,

AGE INT NOT NULL,

PRIMARY KEY (ID)

);

Second table is **Marks** in which we will maintain marks for students based on years. Here **SID** is the foreign key for Student table.

CREATE TABLE Marks(

SID INT NOT NULL,

MARKS INT NOT NULL,

YEAR INT NOT NULL

);

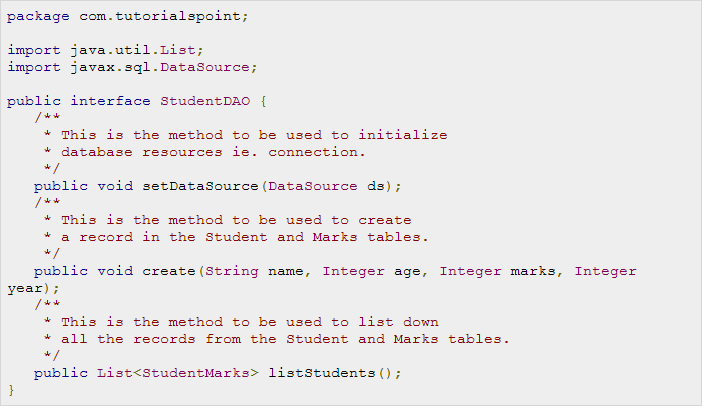
Let us use *PlatformTransactionManager* directly to implement programmatic approach to implement transactions. To start a new transaction you need to have a instance of *TransactionDefinition* with the appropriate transaction attributes. For this example we will simply create an instance of*DefaultTransactionDefinition* to use the default transaction attributes.

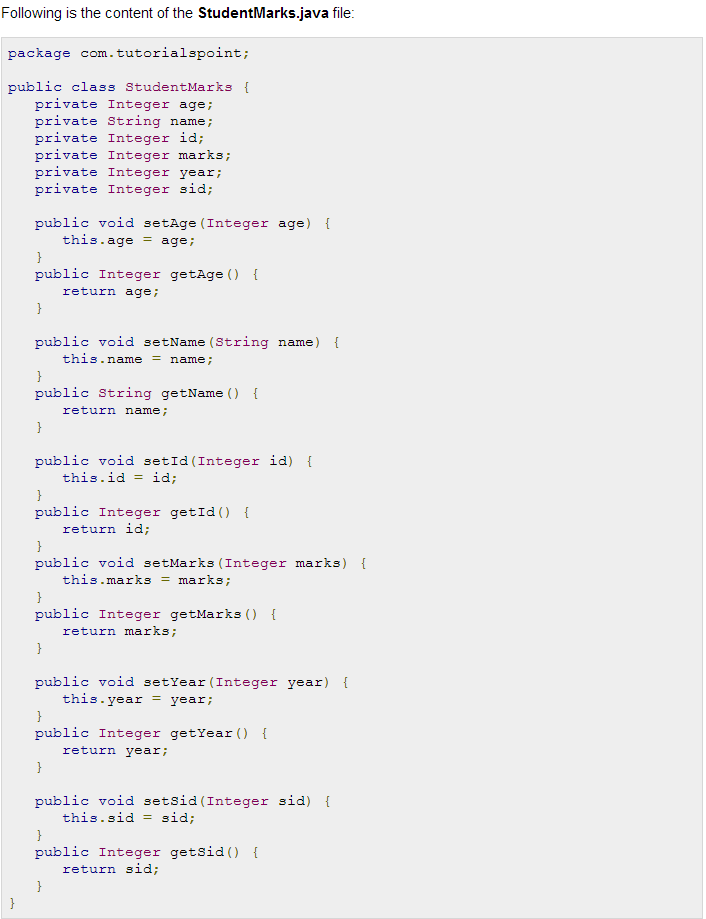
Once the TransactionDefinition is created, you can start your transaction by calling *getTransaction()* method, which returns an instance of *TransactionStatus*. The *TransactionStatus* objects helps in tracking the current status of the transaction and finally, if everything goes fine, you can use *commit()* method of*PlatformTransactionManager* to commit the transaction, otherwise you can use *rollback()* to rollback the complete operation.

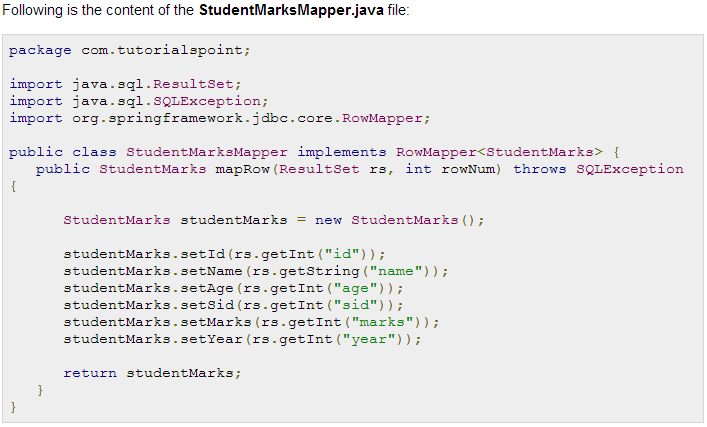
Now let us write our Spring JDBC application which will implement simple operations on Student and Marks tables. Let us have working Eclipse IDE in place and follow the following steps to create a Spring application:

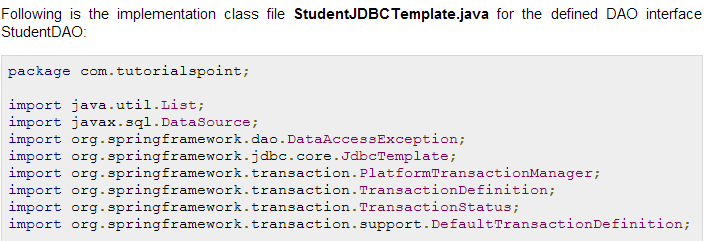
|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the**src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Add Spring JDBC specific latest libraries **mysql-connector-java.jar**,**org.springframework.jdbc.jar** and **org.springframework.transaction.jar** in the project. You can download required libraries if you do not have them already. |
| 4 | Create DAO interface *StudentDAO* and list down all the required methods. Though it is not required and you can directly write *StudentJDBCTemplate* class, but as a good practice, let's do it. |
| 5 | Create other required Java classes *StudentMarks*, *StudentMarksMapper*, *StudentJDBCTemplate*and *MainApp* under the *com.tutorialspoint* package. You can create rest of the POJO classes if required. |
| 6 | Make sure you already created **Student** and **Marks** tables in TEST database. Also make sure your MySQL server is working fine and you have read/write access on the database using the give username and password. |
| 7 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 8 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

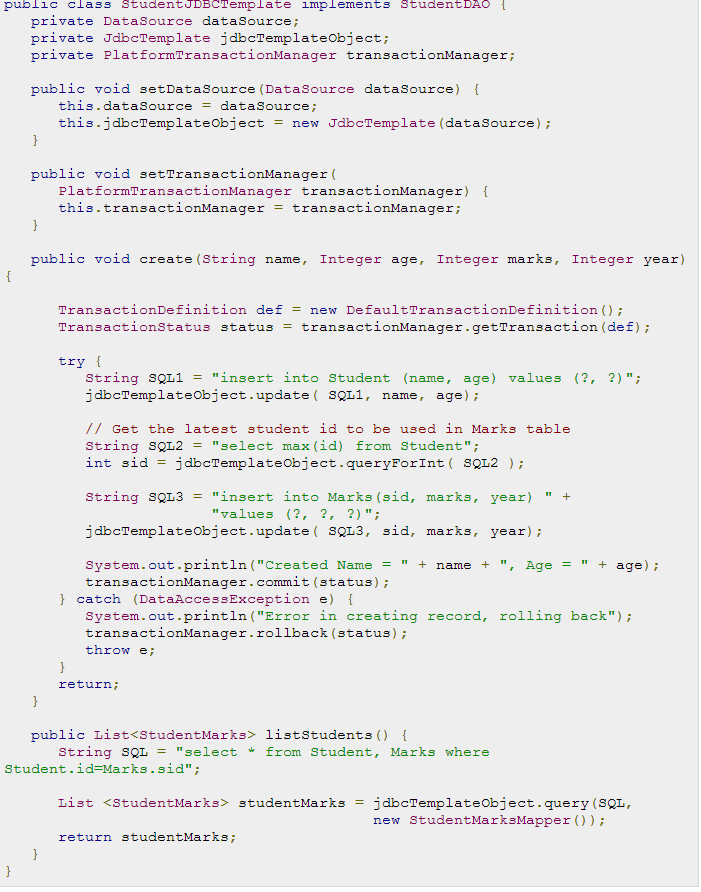
Following is the content of the Data Access Object interface file **StudentDAO.java**:



****

****

****

****

Now let us move with the main application file **MainApp.java**, which is as follows:

****

****

**Spring Declarative Transaction Management**

Declarative transaction management approach allows you to manage the transaction with the help of configuration instead of hard coding in your source code. This means that you can separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions. The bean configuration will specify the methods to be transactional. Here are the steps associated with declarative transaction:

* We use <tx:advice /> tag, which creates a transaction-handling advice and same time we define a pointcut that matches all methods we wish to make transactional and reference the transactional advice.
* If a method name has been included in the transactional configuration then created advice will begin the transaction before calling the method.
* Target method will be executed in a *try / catch* block.
* If the method finishes normally, the AOP advice commits the transaction successfully otherwise it performs a rollback.

Let us see how above mentioned steps work but before we begin, it is important to have at least two database tables on which we can perform various CRUD operations with the help of transactions. Let us take **Student** table, which can be created in MySQL TEST database with the following DDL:

CREATE TABLE Student(

ID INT NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(20) NOT NULL,

AGE INT NOT NULL,

PRIMARY KEY (ID)

);

Second table is **Marks** in which we will maintain marks for students based on years. Here **SID** is the foreign key for Student table.

CREATE TABLE Marks(

SID INT NOT NULL,

MARKS INT NOT NULL,

YEAR INT NOT NULL

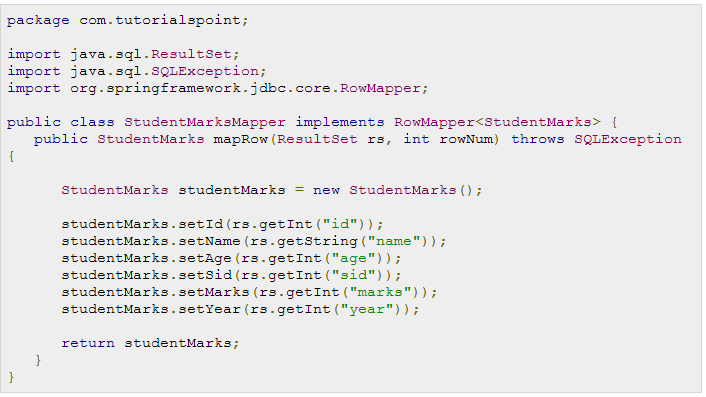
);

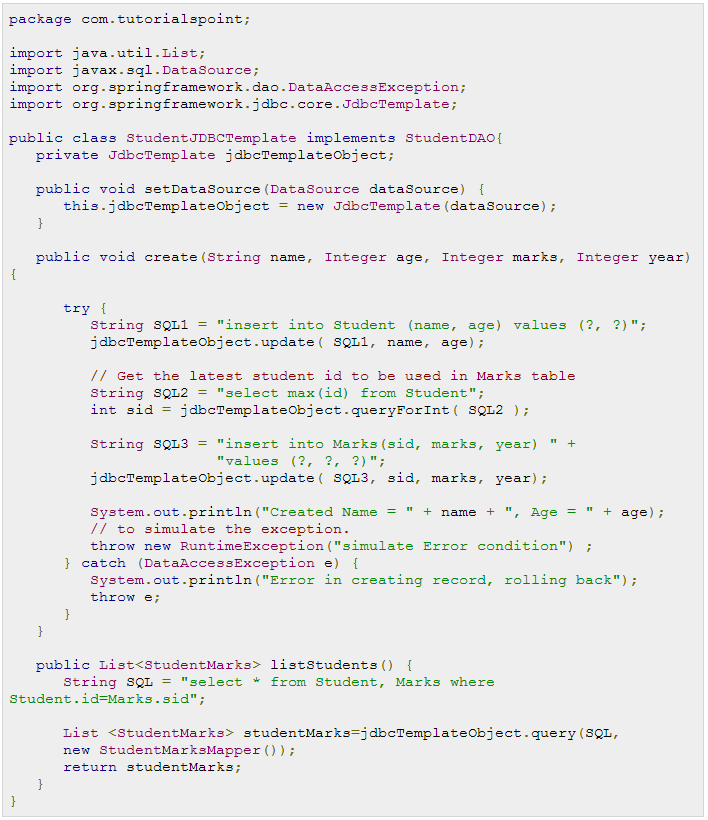
Following is the content of the Data Access Object interface file **StudentDAO.java**:





Following is the content of the **StudentMarksMapper.java** file:

Following is the implementation class file **StudentJDBCTemplate.java** for the defined DAO interface StudentDAO:

****

Now let us move with the main application file **MainApp.java**, which is as follows:

****

Following is the configuration file **Beans.xml**:

****

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following exception will be raised. In this case transaction will be rolled back and no record will be created in the database table.

------Records creation--------

Created Name = Zara, Age = 11

Exception in thread "main" java.lang.RuntimeException: simulate Error condition

You can try above example after removing exception, and in this case it should commit the transaction and you should see a record in the database.

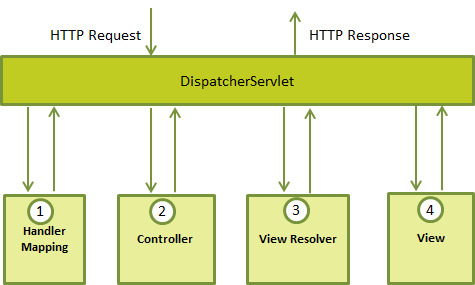
**Spring Web MVC Framework**

The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

* The **Model** encapsulates the application data and in general they will consist of POJO.
* The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The **Controller** is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

## The DispatcherServlet:

The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC*DispatcherServlet* is illustrated in the following diagram:



Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet*:

1. After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
2. The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
3. The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
4. Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

All the above mentioned components ie. HandlerMapping, Controller and ViewResolver are parts of*WebApplicationContext* which is an extension of the plain *ApplicationContext* with some extra features necessary for web applications.

## Required Configuration:

You need to map requests that you want the *DispatcherServlet* to handle, by using a URL mapping in the**web.xml** file. The following is an example to show declaration and mapping for **HelloWeb** *DispatcherServlet*example:

****

The **web.xml** file will be kept *WebContent/WEB-INF* directory of your web application. OK, upon initialization of **HelloWeb** *DispatcherServlet*, the framework will try to load the application context from a file named**[servlet-name]-servlet.xml** located in the application's *WebContent/WEB-INF* directory. In this case our file will be **HelloWeb-servlet.xml**.

Next, <servlet-mapping> tag indicates what URLs will be handled by the which DispatcherServlet. Here all the HTTP requests ending with **.jsp** will be handled by the **HelloWeb** DispatcherServlet.

If you do not want to go with default filename as *[servlet-name]-servlet.xml* and default location as*WebContent/WEB-INF*, you can customize this file name and location by adding the servlet listener*ContextLoaderListener* in your web.xml file as follows:

****

Following are the important points about **HelloWeb-servlet.xml** file:

* The *[servlet-name]-servlet.xml* file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.
* The *<context:component-scan...>* tag will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like @Controller and @RequestMapping etc.
* The *InternalResourceViewResolver* will have rules defined to resolve the view names. As per the above defined rule, a logical view named **hello** is delegated to a view implementation located at*/WEB-INF/jsp/hello.jsp* .

Next section will show you how to create your actual components ie. Controller, Model and View.

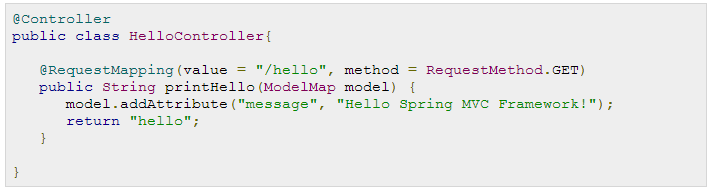
## Defining a Controller

DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The**@Controller** annotation indicates that a particular class serves the role of a controller. The**@RequestMapping** annotation is used to map a URL to either an entire class or a particular handler method.

****

The **@Controller** annotation defines the class as a Spring MVC controller. Here, the first usage of**@RequestMapping** indicates that all handling methods on this controller are relative to the **/hello** path. Next annotation **@RequestMapping(method = RequestMethod.GET)** is used to declare the *printHello()*method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.

You can write above controller in another form where you can add additional attributes in *@RequestMapping*as follows:

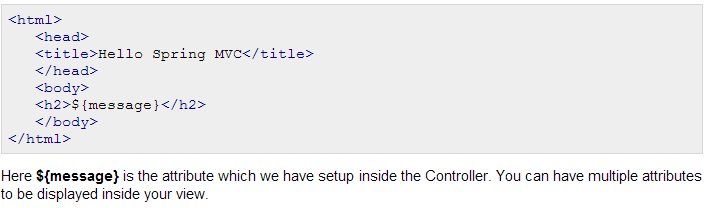
****

The **value** attribute indicates the URL to which the handler method is mapped and the **method** attribute defines the service method to handle HTTP GET request. There are following important points to be noted about the controller defined above:

* You will defined required business logic inside a service method. You can call another methods inside this method as per requirement.
* Based on the business logic defined, you will create a **model** within this method. You can setter different model attributes and these attributes will be accessed by the view to present the final result. This example creates a model with its attribute "message".
* A defined service method can return a String which contains the name of the **view** to be used to render the model. This example returns "hello" as logical view name.

## Creating JSP Views

Spring MVC supports many types of views for different presentation technologies. These include - JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports etc. But most commonly we use JSP templates written with JSTL. So let us write a simple **hello** view in /WEB-INF/hello/hello.jsp:

****

## Spring Web MVC Framework Examples:

Based on the above concepts, let us check few important examples which will help you in building your Spring Web Applications:

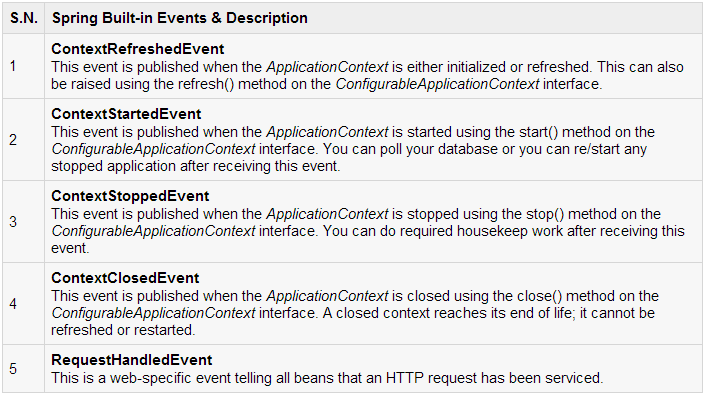
|  |  |
| --- | --- |
| **S.N.** | **Example & Description** |
| 1 | [Spring MVC Hello World Example](http://www.tutorialspoint.com/spring/spring_mvc_hello_world_example.htm) This example will explain how to write a simple Spring Web Hello World application. |
| 2 | [Spring MVC Form Handling Example](http://www.tutorialspoint.com/spring/spring_mvc_form_handling_example.htm) This example will explain how to write a Spring Web application using HTML forms to submit the data to the controller and display back a processed result. |
| 3 | [Spring Page Redirection Example](http://www.tutorialspoint.com/spring/spring_page_redirection_example.htm) Learn how to use page redirection functionality in Spring MVC Framework. |
| 4 | [Spring Static Pages Example](http://www.tutorialspoint.com/spring/spring_static_pages_example.htm) Learn how to access static pages along with dynamic pages in Spring MVC Framework. |
| 5 | [Spring Exception Handling Example](http://www.tutorialspoint.com/spring/spring_exception_handling_example.htm) Learn how to handle exceptions in Spring MVC Framework. |

**Event Handling in Spring**

You have seen in all the chapters that core of Spring is the **ApplicationContext**, which manages complete life cycle of the beans. The ApplicationContext publishes certain types of events when loading the beans. For example, a *ContextStartedEvent* is published when the context is started and *ContextStoppedEvent* is published when the context is stopped.

Event handling in the *ApplicationContext* is provided through the *ApplicationEvent* class and*ApplicationListener* interface. So if a bean implements the *ApplicationListener*, then every time an*ApplicationEvent* gets published to the ApplicationContext, that bean is notified.

Spring provides the following standard events:

****

Spring's event handling is single-threaded so if an event is published, until and unless all the receivers get the message, the processes are blocked and the flow will not continue. Hence, care should be taken when designing your application if event handling is to be used.

## Listening to Context Events:

To listen a context event, a bean should implement the *ApplicationListener* interface which has just one method **onApplicationEvent()**. So let us write an example to see how the events propagates and how you can put your code to do required task based on certain events.

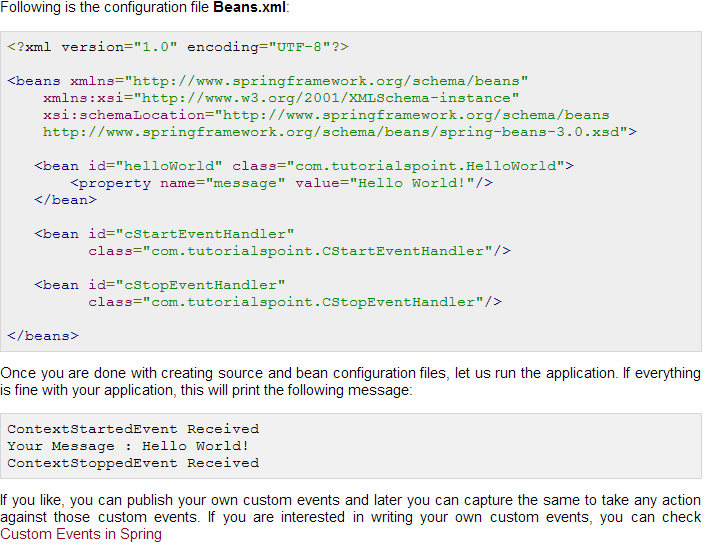
Let us have working Eclipse IDE in place and follow the following steps to create a Spring application:

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the**src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Create Java classes *HelloWorld*, *CStartEventHandler*, *CStopEventHandler* and *MainApp* under the*com.tutorialspoint* package. |
| 4 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **HelloWorld.java** file:

****

****

****

**Custom Events in Spring**

There are number of steps to be taken to write and publish your own custom events. Follow the instructions given in this chapter to write, publish and handle Custom Spring Events.

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the**src** folder in the created project. All the classes will be created under this package. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Create an event class, *CustomEvent* by extending **ApplicationEvent**. This class must define a default constructor which should inherit constructor from ApplicationEvent class. |
| 4 | Once your event class is defined, you can publish it from any class, let us say*EventClassPublisher* which implements *ApplicationEventPublisherAware*. You will also need to declare this class in XML configuration file as a bean so that the container can identify the bean as an event publisher because it implements the ApplicationEventPublisherAware interface. |
| 5 | A published event can be handled in a class, let us say *EventClassHandler* which implements*ApplicationListener* interface and implements *onApplicationEvent* method for the custom event. |
| 6 | Create beans configuration file *Beans.xml* under the **src** folder and a *MainApp* class which will work as Spring application. |
| 7 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **CustomEvent.java** file:

****

****

****

# Spring

### Requirements

Knowledge in Java, basic understanding of annotations and aspects will be very helpful, knowledge in (Eclipse, Intelij Idea, Netbeans)

### Description

The training represents an intensive introduction and thorough acquaintance with the Spring framework. It is built in a way that covers completely the basic concepts in Spring, by trying to develop the knowledge of participants about the technology. Each topic includes examples and explanations, as well as presentation of practical applications.

### Programme

***Prerequisites:***

*Good understanding of the Java programming language.*

*A basic understanding of annotations and aspects will be very helpful.*

1.  Spring Framework Overview

a.   Benefits of Using Spring Framework

b.   Dependency Injection (DI)

c.    Aspect Oriented Programming (AOP)

2.  Spring Framework Architecture

a.   Core Container

b.   Data Access/Integration

c.    Web

d.   Miscellaneous

3.  Spring IoC Containers

a.   What is Spring Container

b.   Spring BeanFactory Container

c.    Example

d.   Spring ApplicationContext Container

e.   Example

4.  Spring Environment Setup Example

a.   Step 1

b.   Setup Java Development Kit (JDK)

c.    Step 2

d.   Install Apache Common Logging API

e.   Step 3

f.      Setup Eclipse IDE

g.   Step 4

h.   Setup Spring Framework Libraries

5.  Spring Hello World Example

a.   Step 1

b.   Create Java Project

c.    Step 2

d.   Add Required Libraries

e.   Step 3

f.      Create Source Files

g.   Step 4

h.   Create Bean Configuration File

i.      Step 5

j.      Running the Program

6.  Spring Bean Definition

a.   What are beans and how to configure them in spring

b.   Spring Configuration Metadata

7.  Spring Bean Scopes

a.   What are Spring beans scopes

b.   The singleton scope

c.    Example

d.   The prototype scope

e.   Example

8.  Spring Bean LifeCycle

a.   Intro to Spring Beans Lifecycle

b.   Initialization callbacks

c.    Destruction callbacks

d.   Example

e.   Default initialization and destroy methods

9.  Spring Bean Post Processors

a.   What are Spring Bean Post Processors

b.   Example configuration

10.                 Spring Bean Definition Inheritance

a.   How Spring Bean Definition Inheritance works

b.   Example Configuration

c.    Bean Definition Template

11.                 Spring Dependency Injection

a.   Intro (What is) Dependency Injection

b.   Constructor - based Dependency Injection

c.    Example

d.   Constructor arguments resolution

e.   Setter - based Dependency Injection

f.      Example

g.   XML Configuration using p-namespace

12.                 Spring Injecting Inner Beans

a.   Explain Inner Beans Injection with Example

13.                 Spring Injecting Collection

a.   Collection Injection with Example

b.   Injecting Bean References

c.    Injecting null and empty string values

14.                 Spring Beans Auto - Wiring

a.   What is Spring autowiring

b.   Autowiring Modes

c.    Limitations with autowiring

d.   Spring Autowiring 'byName'

e.   Spring Autowiring 'byType'

f.      Spring Autowiring by Constructor

15.                 Event Handling in Spring

a.   Intro into Spring Event Handling

b.   Listening to Context Events

16.                 Custom Events in Spring

a.  Intro to Custom events in Spring with example

17.                 Spring Annotation Based Configuration

a.   Intro to annotation based configuration

b.   Spring @Required Annotation

c.    Example

d.   Spring @Autowired Annotation

e.   @Autowired on Setter Methods

f.      Example

g.   @Autowired on Properties

h.   @Autowired on Constructors

i.      @Autowired with (required=false) option

j.      Spring @Qualifier Annotation

k.    Example

l.      Spring JSR-250 Annotations

m. @PostConstruct and @PreDestroy Annotations

n.   Example

o.   @Resource Annotation

18.                 Spring Java Based Configuration

a.   Intro to Java Based configuration

b.   @Configuration & @Bean Annotations

c.    Example

d.   Injecting Bean Dependencies

e.   Example

f.      The @Import Annotation

g.   Lifecycle Callbacks

h.   Specifying Bean Scope

19.                 AOP with Spring Framework

a.   Intro into AOP and Spring AOP

b.   AOP Terminologies

c.    Types of Advice

d.   Custom Aspects Implementation

e.   XML Schema Based AOP with Spring

                                                   i.     Declaring an aspect

                                                 ii.     Declaring a pointcut

                                               iii.     Declaring advices

                                               iv.     Example

f.      @AspectJ Based AOP with Spring

                                                   i.     Declaring an aspect

                                                 ii.     Declaring a pointcut

                                               iii.     Declaring advices

                                               iv.     Example

20.                 Spring JPA with Hibernate

a.    Introduction to ORM(Hibernate) with Spring

b.   Introduction to JPA

c.     Introduction to JPA with Spring

                                                   i.     including EntityManagerFactory & EntityManager,  @PersistenceContext

d.    Explaining 3 types of configurations with Spring

e.    Dealing with more than one persistence units

f.      Example DAOs based Implementation on plain JPA

21.                 Spring Transaction Management

a.   What are Transactions

b.   Local vs. Global Transactions

c.    Programmatic vs Declarative

d.   Spring Transaction Definitions and Abstractions

e.   Spring transactions and hibernate transactions

f.      @Transactional annotation

g.   Configuring transactions with AOP

h.   Potential Pitfalls

                                                   i.     Transactions and Proxies

                                                 ii.     Changing the Isolation level

                                               iii.     Read Only Transactions

### iv.     Transaction Logging

22.                 Spring Web MVC Framework

a.   Intro to MVC (Model View Controller pattern)

b.   The Spring DispatcherServlet

c.    Required Configuration

d.   Defining a Controller

e.   Creating JSP Views

f.      Spring Web MVC Framework Examples

g.   Spring MVC Hello World Example

h.   Spring MVC Form Handling Example

23.                   Spring Versions

a.   difference between spring 3 and 4

b.   configuration differences and migration