Spring Framework 3.1 Tutorial

Simply Easy Learning

tutorialspoint.com

SPRING FRAMEWORK 3.1 TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

Table of Contents

Spring Framework Overview

This chapter gives a basic idea about Spring Framework starting with its origin and its advantages and core technologies associated with the framework.

S pring is the most popular application development framework for enterprise Java.

Millions of developers around the world use Spring Framework to create high performing, easily testable, reusable code.

Spring framework is an open source Java platform and it was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.

Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 2MB.

The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make J2EE development easier to use and promote good programming practice by enabling a POJO-based programming model.

Benefits of Using 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 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, other view technologies.

- Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBean-style 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).

Dependency Injection (DI)

The technology that Spring is most identified with is the **Dependency Injection (DI)** flavor of Inversion of Control. The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways and Dependency Injection is merely one concrete example of Inversion of Control.

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 helps in gluing these classes together and same time keeping them independent.

What is dependency injection exactly? Let's look at these two words separately. Here the dependency part translates into an association between two classes. For example, class A is dependent on class B. Now, let's look at the second part, injection. All this means is that class B will get injected into class A by the IoC.

Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods. As Dependency Injection is the heart of Spring Framework, so I will explain this concept in a separate chapter with a nice example.

Aspect Oriented Programming (AOP)

One of the key components of Spring is the **Aspect oriented programming (AOP)** framework. 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 including logging, 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. Whereas DI helps you decouple your application objects from each other, AOP helps you decouple cross-cutting concerns from the objects that they affect.

The AOP module of Spring Framework provides aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. I will discuss more about Spring AOP concepts in a separate chapter.

Spring Framework Architecture

This section describes the basic architecture of the framework and its main building blocks which are called "modules" in software terminology.

S pring could potentially be a one-stop shop for all your enterprise applications, however,

Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest. Following section gives detail about all the modules available in Spring Framework.

The Spring Framework provides about 20 modules which can be used based on an application requirement.
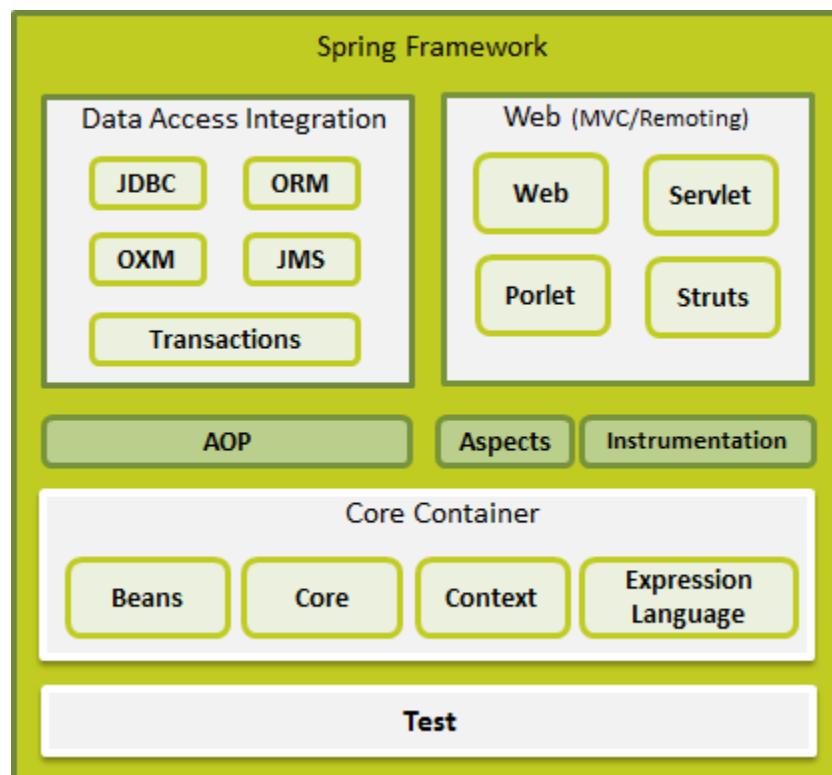
Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules whose detail is 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 Expression Language 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 to do 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-Servlet, Web-Struts, and Web-Portlet modules whose detail is 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-Servlet module contains Spring's model-view-controller (MVC) implementation for web applications.

- The Web-Struts module contains the support classes for integrating a classic Struts web tier within a Spring application.

- 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 whose detail is as follows:

- The AOP module provides 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 aspect oriented programming (AOP) framework.

- The Instrumentation module provides class instrumentation support and class loader implementations to be used in certain application servers.

- The Test module supports the testing of Spring components with JUnit or TestNG frameworks.

Spring Environment Setup

T his tutorial will guide you on how to prepare a development environment to start your

work with Spring Framework. This tutorial will also teach you how to setup JDK, Tomcat and Eclipse on your machine before you setup Spring Framework:

Step 1 - Setup Java Development Kit (JDK)

You can download the latest version of SDK from Oracle's Java site: Java SE Downloads. You will find instructions for installing JDK in downloaded files, follow the given instructions to install and configure the setup. Finally set PATH and JAVA_HOME environment variables to refer to the directory that contains java and javac, typically java_install_dir/bin and java_install_dir respectively.

If you are running Windows and installed the JDK in C:\jdk1.6.0_15, you would have to put the following line in your C:\autoexec.bat file.

```
set PATH=C:\jdk1.6.0_15\bin;%PATH%
set JAVA_HOME=C:\jdk1.6.0_15
```

Alternatively, on Windows NT/2000/XP, you could also right-click on My Computer, select Properties, then Advanced, then Environment Variables. Then, you would update the PATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if the SDK is installed in /usr/local/jdk1.6.0_15 and you use the C shell, you would put the following into your .cshrc file.

```
setenv PATH /usr/local/jdk1.6.0_15/bin:$PATH
setenv JAVA_HOME /usr/local/jdk1.6.0_15
```

Alternatively, if you use an Integrated Development Environment (IDE) like Borland JBuilder, Eclipse, IntelliJ IDEA, or Sun ONE Studio, compile and run a simple program to confirm that the IDE knows where you installed Java, otherwise do proper setup as given document of the IDE.

Step 2 - Install Apache Common Logging API

You can download the latest version of Apache Commons Logging API from http://commons.apache.org/logging/. Once you downloaded the installation, unpack the binary distribution into a convenient location. For example in C:\commons-logging-1.1.1 on windows, or /usr/local/commons-logging-1.1.1 on Linux/Unix. This directory will have following jar files and other supporting documents etc.

Make sure you set your CLASSPATH variable on this directory properly otherwise you will face problem while running your application.

Step 3 - Setup Eclipse IDE

All the examples in this tutorial have been written using Eclipse IDE. So I would suggest you should have latest version of Eclipse installed on your machine.

To install Eclipse IDE, download the latest Eclipse binaries from http://www.eclipse.org/downloads/. Once you downloaded the installation, unpack the binary distribution into a convenient location. For example in C:\eclipse on windows, or /usr/local/eclipse on Linux/Unix and finally set PATH variable appropriately.

Eclipse can be started by executing the following commands on windows machine, or you can simply double click on eclipse.exe

```
%C:\eclipse\eclipse.exe
```

Eclipse can be started by executing the following commands on Unix (Solaris, Linux, etc.) machine:

```
$/usr/local/eclipse/eclipse
```

After a successful startup, if everything is fine then it should display following result:
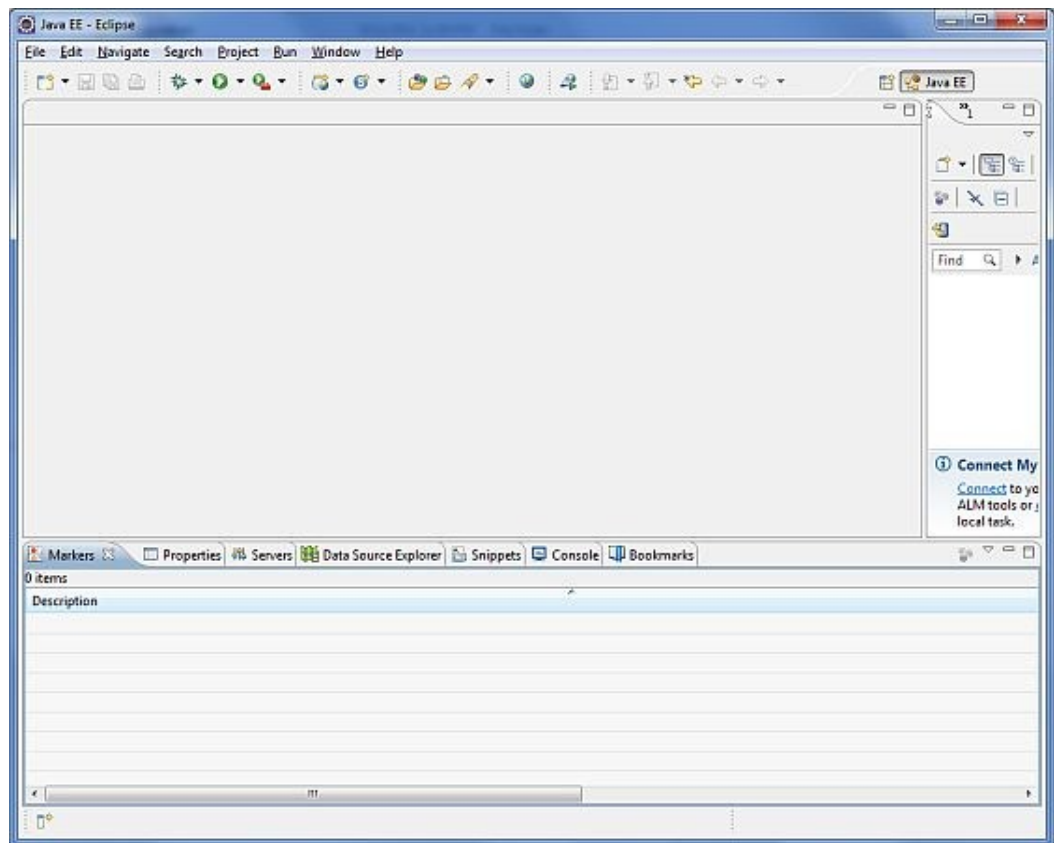
Step 4 - Setup Spring Framework Libraries

Now if everything is fine, then you can proceed to setup your Spring framework. Following are the simple steps to download and install the framework on your machine.

- Make a choice whether you want to install Spring on Windows, or Unix and then proceed to the next step to download .zip file for windows and .tz file for Unix.

- Download the latest version of Spring framework binaries from http://www.springsource.org/download.

- At the time of writing this tutorial, I downloaded spring-framework-3.1.0.M2.zip on my Windows machine and when you unzip the downloaded file it will give you directory structure inside C:\spring-framework-3.1.0.M2 as follows.

Name	Date modified	Type	Size
dist	6/8/2011 4:38 AM	File folder	
projects	6/8/2011 4:45 AM	File folder	
src	6/8/2011 4:38 AM	File folder	
changelog	6/8/2011 4:45 AM	Text Document	50 KB
license	6/8/2011 4:45 AM	Text Document	15 KB
notice	6/8/2011 4:45 AM	Text Document	1 KB
readme	6/8/2011 4:45 AM	Text Document	1 KB

You will find all the Spring libraries in the directory **C:\spring-framework-3.1.0.M2\dist**. Make sure you set your CLASSPATH variable on this directory properly otherwise you will face problem while running your application. If you are using Eclipse then it is not required to set CLASSPATH because all the setting will be done through Eclipse.

Once you are done with this last step, you are ready to proceed for your first Spring Example which you will see in the next chapter.

Spring Hello World Example

L̲et us start actual programming with Spring Framework. Before you start writing your first

example using Spring framework, you have make sure that you have setup your Spring environment properly as explained in Spring - Environment Setup tutorial. I also assume that you have a little bit working knowledge with Eclipse IDE.

So let us proceed to write a simple Spring Application which will print "Hello World!" or any other message based on the configuration done in Spring Beans Configuration file.

Step 1 - Create Java Project

The first step is to create a simple Java Project using Eclipse IDE. Follow the option **File → New → Project** and finally select **Java Project** wizard from the wizard list. Now name your project as **HelloSpring** using the wizard window as follows:
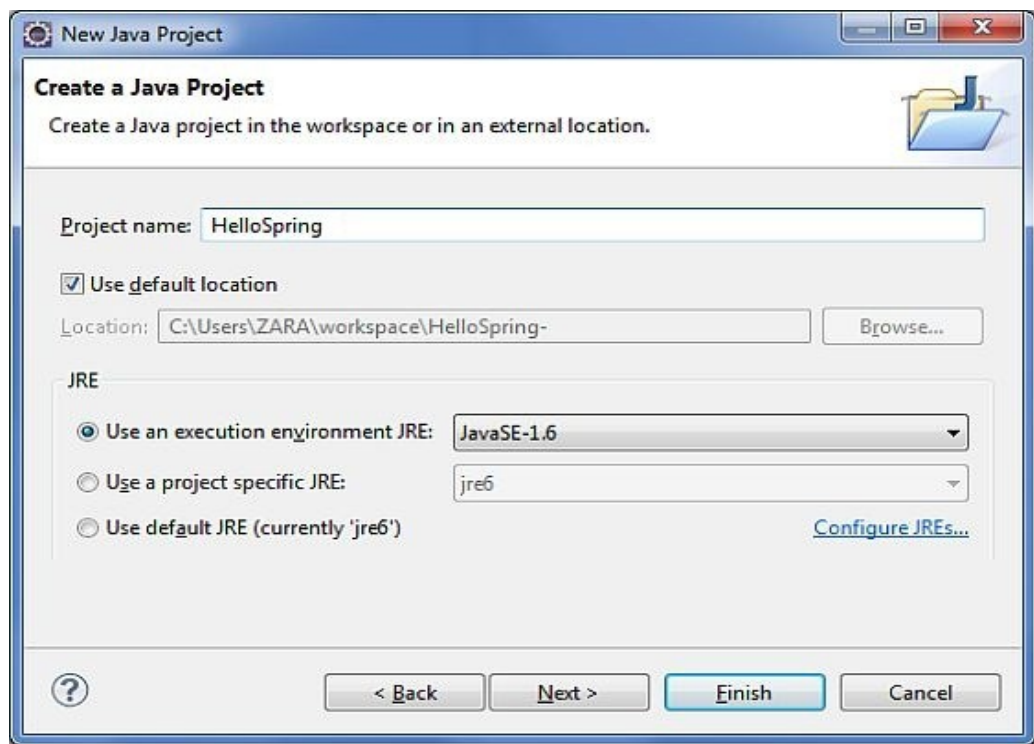
Once your project is created successfully, you will have following content in your **Project Explorer:**
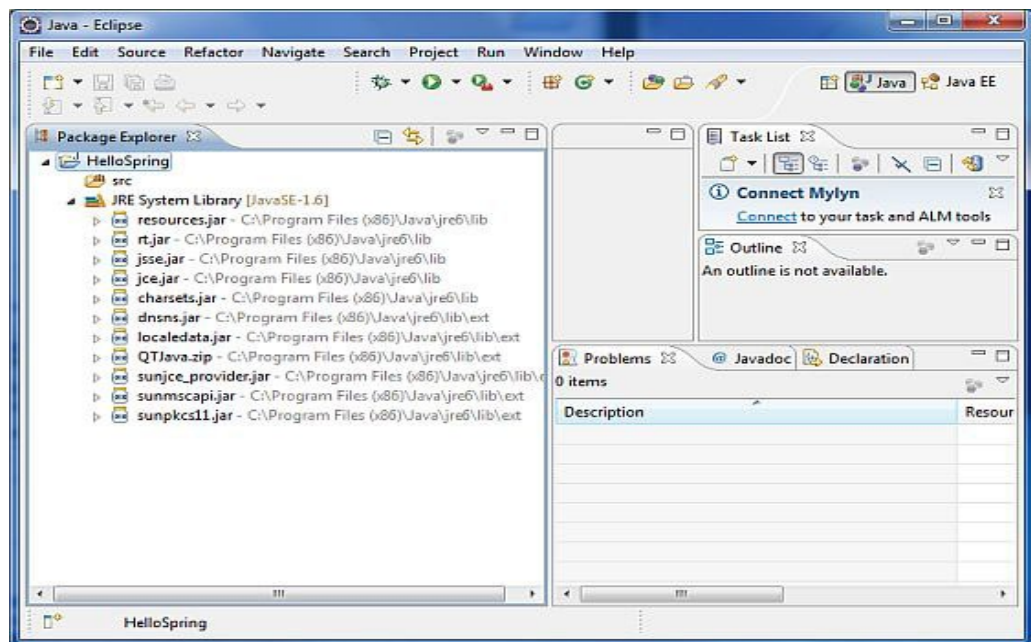
Step 2 - Add Required Libraries

As a second step let us add Spring Framework and common logging API libraries in our project. To do this, right click on your project name **HelloSpring** and then follow the following option available in context menu: **Build Path → Configure Build Path** to display the Java Build Path window as follows:
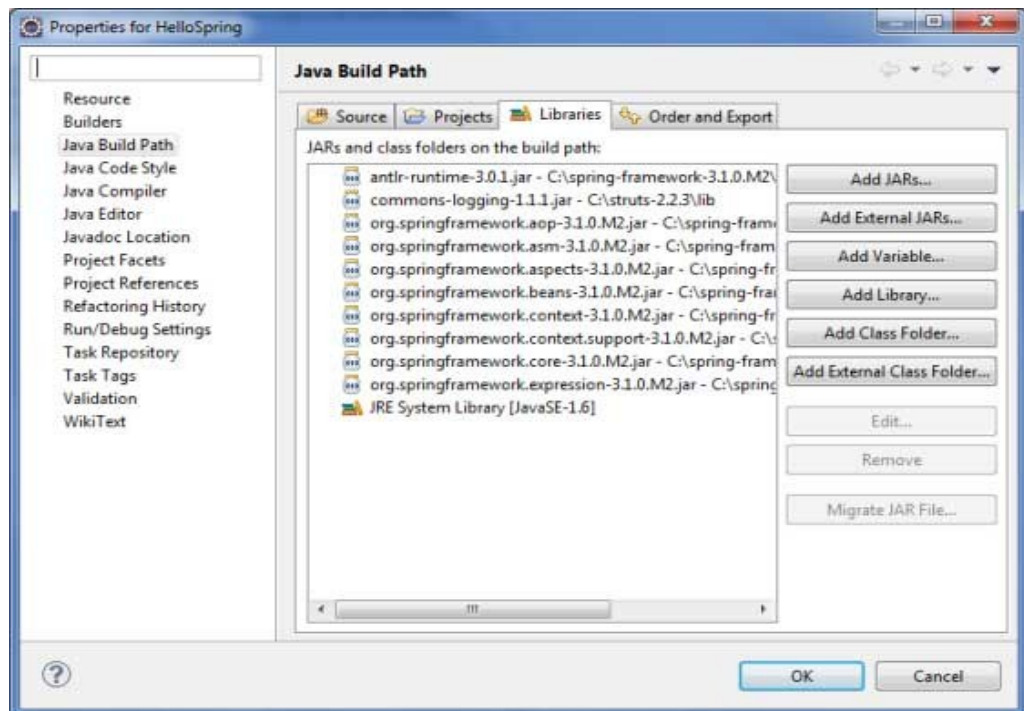
Now use **Add External JARs** button available under **Libraries** tab to add the following core JARs from Spring Framework and Common Logging installation directories:

- antlr-runtime-3.0.1

- org.springframework.aop-3.1.0.M2

- org.springframework.asm-3.1.0.M2

- org.springframework.aspects-3.1.0.M2

- org.springframework.beans-3.1.0.M2

- org.springframework.context.support-3.1.0.M2

- org.springframework.context-3.1.0.M2

- org.springframework.core-3.1.0.M2

- org.springframework.expression-3.1.0.M2

- commons-logging-1.1.1

Step 3 - Create Source Files

Now let us create actual source files under the **HelloSpring** project. First we need to create a package called **com.tutorialspoint**. To do this, right click on **src** in package explorer section and follow the option : **New → Package**.

Next we will create **HelloWorld.java** and **MainApp.java** files under the com.tutorialspoint package.

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

```java
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }

   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

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

```java
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
            new ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

      obj.getMessage();
   }
}
```

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

1. First step is to create application context where we used framework APIClassPathXmlApplicationContext(). This API loads beans configuration file and eventually based on the provided API, it takes care of creating and initializing all the objects ie. beans mentioned in the 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.

Step 4 - Create Bean Configuration File

You need to create a Bean Configuration file which is an XML file and acts as cement that glues the beans ie. classes together. This file needs to be created under the **src** directory as shown below:
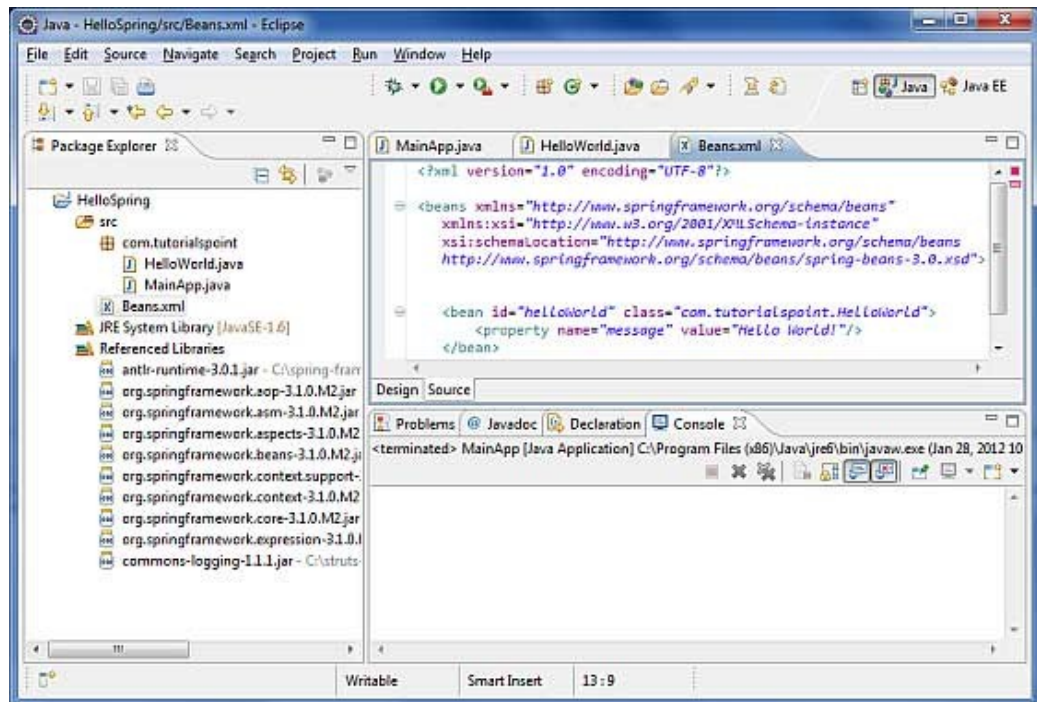
Usually developers keep this file name as **Beans.xml**, but you are independent to choose any name you like. You have to make sure that this file is available in CLASSPATH and use the same name in main application while creating application context as shown in MainApp.java file.

The Beans.xml is used to assign unique IDs to different beans and to control the creation of objects with different values without impacting any of the Spring source files. For example, using below file you can pass any value for "message" variable and so you can print different values of message without impacting HelloWorld.java and MainApp.java files. Let us see how it works:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd">

   <bean id="helloWorld" class="com.tutorialspoint.HelloWorld">
       <property name="message" value="Hello World!"/>
   </bean>

</beans>
```

When Spring application gets loaded into the memory, Framework makes use of the above configuration file to create all the beans defined and assign them a unique ID as defined in **<bean>**tag. You can use **<property>** tag to pass the values of different variables used at the time of object creation.

Step 5 - Running the Program

Once you are done with creating source and beans configuration files, you are ready for this step which is compiling and running your program. To do this, Keep MainApp.Java file tab active and

use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **MainApp** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```
Your Message : Hello World!
```

Congratulations, you have created your first Spring Application successfully. You can see the flexibility of above Spring application by changing the value of "message" property and keeping both the source files unchanged. Further, let us start doing something more interesting in next few chapters.

Spring IoC Containers

T he 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. These objects are called Spring Beans which we will discuss in next chapter.

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.
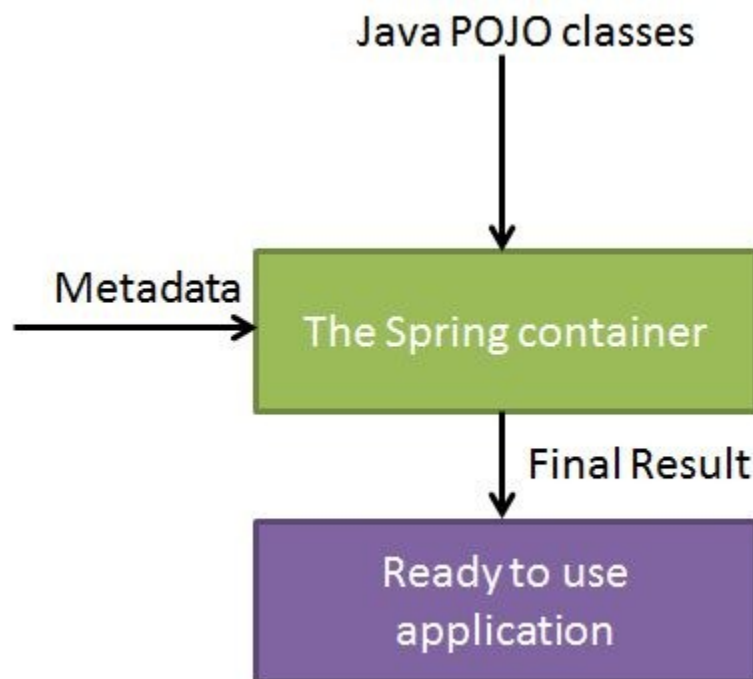
Spring provides following two distinct types of containers.

S.N.	Container & Description
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.
2	**Spring ApplicationContext Container** This container 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* container includes all functionality of the *BeanFactory* container, so it is generally recommended over the *BeanFactory*. BeanFactory can still be used for light weight applications like mobile devices or applet based applications where data volume and speed is significant.

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:

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* 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:

```
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }

   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

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

```
package com.tutorialspoint;

import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class MainApp {
   public static void main(String[] args) {
      XmlBeanFactory factory = new XmlBeanFactory
                              (new ClassPathResource("Beans.xml"));

      HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");
      obj.getMessage();
   }
}
```

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 XmlBeanFactory() to create the factory bean and ClassPathResource() API to load the bean configuration file available in CLASSPATH. The XmlBeanFactory() API takes care of creating and initializing all the objects ie. beans mentioned in the configuration file.

2. Second step is used to get required bean using getBean() method of the created bean factory object. 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

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

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id="helloWorld" class="com.tutorialspoint.HelloWorld">
      <property name="message" value="Hello World!"/>
   </bean>
```

```
</beans>
```

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:

```
Your Message : Hello World!
```

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.

Example

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* 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:

```java
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }

   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

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

```java
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {

      ApplicationContext context = new FileSystemXmlApplicationContext
            ("C:/Users/ZARA/workspace/HelloSpring/src/Beans.xml");

      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
      obj.getMessage();
   }
}
```

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

1. First step is to create factory object where we used framework APIFileSystemXmlApplicationContext 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

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="helloWorld" class="com.tutorialspoint.HelloWorld">
        <property name="message" value="Hello World!"/>
    </bean>

</beans>
```

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:

```
Your Message : Hello World!
```

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.

Properties	Description
class	This attribute is mandatory and specify the bean class to be used to create the bean.
name	This attribute specifies the bean identifier uniquely. In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).
scope	This attribute specifies the scope of the objects created from a particular bean definition and it will be discussed in bean scopes chapter.
constructor-arg	This is used to inject the dependencies and will be discussed in next chapters.
properties	This is used to inject the dependencies and will be discussed in next chapters.
autowiring mode	This is used to inject the dependencies and will be discussed in next chapters.

lazy-initialization mode	A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.
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.
destruction method	A callback to be used when the container containing the bean is destroyed. It will be discussed in bean life cycle chapter.

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

You already have seen how XML based configuration metadata provided to the container, but let us see another sample of XML based configuration file with different bean definitions including lazy initialization, initialization method and destruction method:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd">

  <!-- A simple bean definition -->
   <bean id="..." class="...">
  <!-- collaborators and configuration for this bean go here -->
   </bean>

  <!-- A bean definition with lazy init set on -->
   <bean id="..." class="..." lazy-init="true">
  <!-- collaborators and configuration for this bean go here -->
   </bean>

   <!-- A bean definition with initialization method -->
   <bean id="..." class="..." init-method="...">
  <!-- collaborators and configuration for this bean go here -->
   </bean>

  <!-- A bean definition with destruction method -->
   <bean id="..." class="..." destroy-method="...">
  <!-- collaborators and configuration for this bean go here -->
   </bean>

   <!-- more bean definitions go here -->
```

```
</beans>
```

You can check <u>Spring Hello World Example</u> to understand how to define, configure and create Spring Beans.

I will discuss about Annotation Based Configuration in a separate chapter. I kept it intentionally in a separate chapter because I want you to grasp few other important Spring concepts before you start programming width Spring Dependency Injection with Annotations.

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.

Scope	Description
singleton	This scopes the bean definition to a single instance per Spring IoC container (default).
prototype	This scopes a single bean definition to have any number of object instances.
request	This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
session	This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
global-session	This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

This chapter will discuss about first two scopes and remaining three will be discussed when we will discuss about web-aware Spring 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:

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

```
<bean id="..." class="..." scope="singleton">
    <!-- collaborators and configuration for this bean go here -->
</bean>
```

Example

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* 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:

```
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }

   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

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

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
            new ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

      objA.setMessage("I'm object A");
```

```
        objA.getMessage();

        HelloWorld objB = (HelloWorld) context.getBean("helloWorld");
        objB.getMessage();
    }
}
```

Following is the configuration file **Beans.xml** required for singleton scope:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd">

    <bean id="helloWorld" class="com.tutorialspoint.HelloWorld"
        scope="singleton">
    </bean>

</beans>
```

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:

```
Your Message : I'm object A

Your Message : I'm object A
```

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:

```xml
<!-- A bean definition with singleton scope -->
<bean id="..." class="..." scope="prototype">
    <!-- collaborators and configuration for this bean go here -->
</bean>
```

Example

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* 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:

```
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }

   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

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

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
             new ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

      objA.setMessage("I'm object A");
      objA.getMessage();

      HelloWorld objB = (HelloWorld) context.getBean("helloWorld");
      objB.getMessage();
   }
}
```

Following is the configuration file **Beans.xml** required for prototype scope:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
   xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd">

   <bean id="helloWorld" class="com.tutorialspoint.HelloWorld"
      scope="prototype">
   </bean>

</beans>
```

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:

```
Your Message : I'm object A

Your Message : null
```

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 is 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:

```
void afterPropertiesSet() throws Exception;
```

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

```
public class ExampleBean implements InitializingBean {
    public void afterPropertiesSet() {
        // do some initialization work
    }
}
```

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

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

```
void destroy() throws Exception;
```

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

```
public class ExampleBean implements DisposableBean {
    public void destroy() {
        // do some destruction work
    }
}
```

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
    }
}
```

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

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* 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:

```java
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
   public void init(){
      System.out.println("Bean is going through init.");
   }
   public void destroy(){
      System.out.println("Bean will destroy now.");
   }
}
```

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.

```java
package com.tutorialspoint;

import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {

      AbstractApplicationContext context =
                         new
ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
      obj.getMessage();
      context.registerShutdownHook();
   }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

```
    <bean id="helloWorld"
        class="com.tutorialspoint.HelloWorld"
        init-method="init" destroy-method="destroy">
        <property name="message" value="Hello World!"/>
    </bean>

</beans>
```

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:

```
Bean is going through init.

Your Message : Hello World!

Bean will destroy now.
```

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:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
    default-init-method="init"
    default-destroy-method="destroy">

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

</beans>
```

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.

Example

The following examples show how to write, register, and use BeanPostProcessors in the context of an ApplicationContext.

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*, *InitHelloWorld* 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:

```
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }

   public void getMessage(){
      System.out.println("Your Message : " + message);
   }

   public void init(){
      System.out.println("Bean is going through init.");
   }

   public void destroy(){
      System.out.println("Bean will destroy now.");
   }
}
```

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:

```
package com.tutorialspoint;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;

public class InitHelloWorld implements BeanPostProcessor {

   public Object postProcessBeforeInitialization(Object bean,
               String beanName) throws BeansException {
      System.out.println("BeforeInitialization : " + beanName);
      return bean;  // you can return any other object as well
   }

   public Object postProcessAfterInitialization(Object bean,
               String beanName) throws BeansException {
      System.out.println("AfterInitialization : " + beanName);
      return bean;  // you can return any other object as well
   }

}
```

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.

```
package com.tutorialspoint;

import org.springframework.context.support.AbstractApplicationContext;
```

```
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {

      AbstractApplicationContext context =
                  new ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
      obj.getMessage();
      context.registerShutdownHook();
   }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id="helloWorld" class="com.tutorialspoint.HelloWorld"
       init-method="init" destroy-method="destroy">
       <property name="message" value="Hello World!"/>
   </bean>

   <bean class="com.tutorialspoint.InitHelloWorld" />

</beans>
```

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:

```
BeforeInitialization : helloWorld

Bean is going through init.

AfterInitialization : helloWorld

Your Message : Hello World!

Bean will destroy now.
```

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

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*, *HelloIndia* 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.

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*.

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd">

   <bean id="helloWorld" class="com.tutorialspoint.HelloWorld">
       <property name="message1" value="Hello World!"/>
       <property name="message2" value="Hello Second World!"/>
   </bean>

   <bean id="helloIndia" class="com.tutorialspoint.HelloIndia"
       parent="helloWorld">
       <property name="message1" value="Hello India!"/>
       <property name="message3" value="Namaste India!"/>
   </bean>

</beans>
```

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

```java
package com.tutorialspoint;

public class HelloWorld {
   private String message1;
   private String message2;

   public void setMessage1(String message){
      this.message1  = message;
   }

   public void setMessage2(String message){
      this.message2  = message;
   }

   public void getMessage1(){
      System.out.println("World Message1 : " + message1);
   }

   public void getMessage2(){
      System.out.println("World Message2 : " + message2);
   }
}
```

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

```java
package com.tutorialspoint;

public class HelloIndia {
   private String message1;
   private String message2;
   private String message3;
```

```
   public void setMessage1(String message){
      this.message1  = message;
   }

   public void setMessage2(String message){
      this.message2  = message;
   }

   public void setMessage3(String message){
      this.message3  = message;
   }

   public void getMessage1(){
      System.out.println("India Message1 : " + message1);
   }

   public void getMessage2(){
      System.out.println("India Message2 : " + message2);
   }

   public void getMessage3(){
      System.out.println("India Message3 : " + message3);
   }
}
```

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

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
            new ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

      objA.getMessage1();
      objA.getMessage2();

      HelloIndia objB = (HelloIndia) context.getBean("helloIndia");
      objB.getMessage1();
      objB.getMessage2();
      objB.getMessage3();
   }
}
```

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:

```
World Message1 : Hello World!

World Message2 : Hello Second World!
```

```
India Message1 : Hello India!

India Message2 : Hello Second World!

India Message3 : Namaste India!
```

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:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd">

   <bean id="beanTeamplate" abstract="true">
       <property name="message1" value="Hello World!"/>
       <property name="message2" value="Hello Second World!"/>
       <property name="message3" value="Namaste India!"/>
   </bean>

   <bean id="helloIndia" class="com.tutorialspoint.HelloIndia"
       parent="beanTeamplate">
       <property name="message1" value="Hello India!"/>
       <property name="message3" value="Namaste India!"/>
   </bean>

</beans>
```

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 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:

```java
public class TextEditor {
   private SpellChecker spellChecker;
   public TextEditor() {
      spellChecker = new SpellChecker();
   }
}
```

What we've done here is create a dependency between the TextEditor and the SpellChecker. In an inversion of control scenario we would instead do something like this:

```java
public class TextEditor {
   private SpellChecker spellChecker;
   public TextEditor(SpellChecker spellChecker) {
      this.spellChecker = spellChecker;
   }
}
```

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:

S.N.	Dependency Injection Type & Description
1	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.
2	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.

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

The following example shows a class *TextEditor* that can only be dependency-injected with constructor injection.

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 *TextEditor*, *SpellChecker* 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 **TextEditor.java** file:

```
package com.tutorialspoint;
```

```
public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor(SpellChecker spellChecker) {
      System.out.println("Inside TextEditor constructor." );
      this.spellChecker = spellChecker;
   }
   public void spellCheck() {
      spellChecker.checkSpelling();
   }
}
```

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

```
package com.tutorialspoint;

public class SpellChecker {
   public SpellChecker(){
      System.out.println("Inside SpellChecker constructor." );
   }

   public void checkSpelling() {
      System.out.println("Inside checkSpelling." );
   }

}
```

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

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
            new ClassPathXmlApplicationContext("Beans.xml");

      TextEditor te = (TextEditor) context.getBean("textEditor");

      te.spellCheck();
   }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

```
   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.tutorialspoint.TextEditor">
      <constructor-arg ref="spellChecker"/>
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.tutorialspoint.SpellChecker">
   </bean>

</beans>
```

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:

```
Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.
```

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:

```
package x.y;

public class Foo {
   public Foo(Bar bar, Baz baz) {
      // ...
   }
}
```

The following configuration works fine:

```
<beans>
   <bean id="foo" class="x.y.Foo">
      <constructor-arg ref="bar"/>
      <constructor-arg ref="baz"/>
   </bean>

   <bean id="bar" class="x.y.Bar"/>
   <bean id="baz" class="x.y.Baz"/>
</beans>
```

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

```
package x.y;

public class Foo {
   public Foo(int year, String name) {
```

```
        // ...
    }
}
```

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:

```
<beans>

   <bean id="exampleBean" class="examples.ExampleBean">
      <constructor-arg type="int" value="2001"/>
      <constructor-arg type="java.lang.String" value="Zara"/>
   </bean>

</beans>
```

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:

```
<beans>

   <bean id="exampleBean" class="examples.ExampleBean">
      <constructor-arg index="0" value="2001"/>
      <constructor-arg index="1" value="Zara"/>
   </bean>

</beans>
```

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

The following example shows a class *TextEditor* that can only be dependency-injected using pure setter-based injection.

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 *TextEditor*, *SpellChecker* 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 **TextEditor.java** file:

```java
package com.tutorialspoint;

public class TextEditor {
   private SpellChecker spellChecker;

   // a setter method to inject the dependency.
   public void setSpellChecker(SpellChecker spellChecker) {
      System.out.println("Inside setSpellChecker." );
      this.spellChecker = spellChecker;
   }
   // a getter method to return spellChecker
   public SpellChecker getSpellChecker() {
      return spellChecker;
   }

   public void spellCheck() {
      spellChecker.checkSpelling();
   }
}
```

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**:

```java
package com.tutorialspoint;

public class SpellChecker {
   public SpellChecker(){
      System.out.println("Inside SpellChecker constructor." );
   }

   public void checkSpelling() {
      System.out.println("Inside checkSpelling." );
   }

}
```

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

```java
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
```

```
            new ClassPathXmlApplicationContext("Beans.xml");

      TextEditor te = (TextEditor) context.getBean("textEditor");

      te.spellCheck();
   }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.tutorialspoint.TextEditor">
      <property name="spellChecker" ref="spellChecker"/>
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.tutorialspoint.SpellChecker">
   </bean>

</beans>
```

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:

```
Inside SpellChecker constructor.

Inside setSpellChecker.

Inside checkSpelling.
```

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:

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

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

```
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id="john-classic" class="com.example.Person">
      <property name="name" value="John Doe"/>
      <property name="spouse" ref="jane"/>
   </bean>

   <bean name="jane" class="com.example.Person">
      <property name="name" value="John Doe"/>
   </bean>

</beans>
```

Above XML configuration can be re-written in a cleaner way using **p-namespace** as follows:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id="john-classic" class="com.example.Person"
      p:name="John Doe"
      p:spouse-ref="jane"/>
   </bean>

   <bean name="jane" class="com.example.Person"
      p:name="John Doe"/>
   </bean>

</beans>
```

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.

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="outerBean" class="...">
        <property name="target">
            <bean id="innerBean" class="..."/>
        </property>
    </bean>

</beans>
```

Example

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 *TextEditor*, *SpellChecker* 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 **TextEditor.java** file:

```java
package com.tutorialspoint;

public class TextEditor {
   private SpellChecker spellChecker;

   // a setter method to inject the dependency.
   public void setSpellChecker(SpellChecker spellChecker) {
      System.out.println("Inside setSpellChecker." );
      this.spellChecker = spellChecker;
   }
   // a getter method to return spellChecker
   public SpellChecker getSpellChecker() {
      return spellChecker;
   }

   public void spellCheck() {
      spellChecker.checkSpelling();
   }
}
```

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

```java
package com.tutorialspoint;

public class SpellChecker {
   public SpellChecker(){
      System.out.println("Inside SpellChecker constructor." );
   }

   public void checkSpelling(){
      System.out.println("Inside checkSpelling." );
   }

}
```

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

```java
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
             new ClassPathXmlApplicationContext("Beans.xml");

      TextEditor te = (TextEditor) context.getBean("textEditor");

      te.spellCheck();
   }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <!-- Definition for textEditor bean using inner bean -->
   <bean id="textEditor" class="com.tutorialspoint.TextEditor">
      <property name="spellChecker">
      <bean id="spellChecker" class="com.tutorialspoint.SpellChecker"/>
      </property>
   </bean>

</beans>
```

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:

```
Inside SpellChecker constructor.

Inside setSpellChecker.

Inside checkSpelling.
```

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:

Element	Description
<list>	This helps in wiring ie injecting a list of values, allowing duplicates.
<set>	This helps in wiring a set of values but without any duplicates.
<map>	This can be used to inject a collection of name-value pairs where name and value can be of any type.
<props>	This can be used to inject a collection of name-value pairs where the name and value are both Strings.

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:

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 *JavaCollection*, 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 **JavaCollection.java** file:

```java
package com.tutorialspoint;
import java.util.*;

public class JavaCollection {
   List addressList;
   Set  addressSet;
   Map  addressMap;
   Properties addressProp;

   // a setter method to set List
   public void setAddressList(List addressList) {
      this.addressList = addressList;
   }
   // prints and returns all the elements of the list.
   public List getAddressList() {
      System.out.println("List Elements :"  + addressList);
      return addressList;
   }

   // a setter method to set Set
   public void setAddressSet(Set addressSet) {
      this.addressSet = addressSet;
   }

   // prints and returns all the elements of the Set.
   public Set getAddressSet() {
      System.out.println("Set Elements :"  + addressSet);
      return addressSet;
   }

   // a setter method to set Map
   public void setAddressMap(Map addressMap) {
      this.addressMap = addressMap;
   }
   // prints and returns all the elements of the Map.
   public Map getAddressMap() {
      System.out.println("Map Elements :"  + addressMap);
      return addressMap;
   }

   // a setter method to set Property
   public void setAddressProp(Properties addressProp) {
      this.addressProp = addressProp;
   }
   // prints and returns all the elements of the Property.
   public Properties getAddressProp() {
      System.out.println("Property Elements :"  + addressProp);
      return addressProp;
   }
}
```

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

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
              new ClassPathXmlApplicationContext("Beans.xml");

      JavaCollection
jc=(JavaCollection)context.getBean("javaCollection");

      jc.getAddressList();
      jc.getAddressSet();
      jc.getAddressMap();
      jc.getAddressProp();
   }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <!-- Definition for javaCollection -->
   <bean id="javaCollection"
class="com.tutorialspoint.JavaCollection">

      <!-- results in a setAddressList(java.util.List) call -->
      <property name="addressList">
        <list>
           <value>INDIA</value>
           <value>Pakistan</value>
           <value>USA</value>
           <value>USA</value>
        </list>
      </property>

     <!-- results in a setAddressSet(java.util.Set) call -->
     <property name="addressSet">
        <set>
           <value>INDIA</value>
           <value>Pakistan</value>
           <value>USA</value>
           <value>USA</value>
        </set>
      </property>

     <!-- results in a setAddressMap(java.util.Map) call -->
```

```
        <property name="addressMap">
           <map>
              <entry key="1" value="NDIA"/>
              <entry key="2" value="Pakistan"/>
              <entry key="3" value="USA"/>
              <entry key="4" value="USA"/>
           </map>
        </property>

        <!-- results in a setAddressProp(java.util.Properties) call -->
        <property name="addressProp">
           <props>
              <prop key="one">INDIA</prop>
              <prop key="two">Pakistan</prop>
              <prop key="three">USA</prop>
              <prop key="four">USA</prop>
           </props>
        </property>

    </bean>

</beans>
```

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:

```
List Elements :[INDIA, Pakistan, USA, USA]

Set Elements :[INDIA, Pakistan, USA]

Map Elements :{1=NDIA, 2=Pakistan, 3=USA, 4=USA}

Property Elements :{two=Pakistan, one=INDIA, three=USA, four=USA}
```

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:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <!-- Bean Definition to handle references and values -->
   <bean id="..." class="...">

      <!-- Passing bean reference  for java.util.List -->
      <property name="addressList">
        <list>
           <ref bean="address1"/>
           <ref bean="address2"/>
           <value>Pakistan</value>
```

```
            </list>
        </property>

    <!-- Passing bean reference  for java.util.Set -->
    <property name="addressSet">
        <set>
            <ref bean="address1"/>
            <ref bean="address2"/>
            <value>Pakistan</value>
        </set>
     </property>

    <!-- Passing bean reference  for java.util.Map -->
    <property name="addressMap">
        <map>
            <entry key="one" value="NDIA"/>
            <entry key ="two" value-ref="address1"/>
            <entry key ="three" value-ref="address2"/>
        </map>
    </property>

    </bean>

</beans>
```

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:

```
<bean id="..." class="exampleBean">
    <property name="email" value=""/>
</bean>
```

The preceding example is equivalent to the Java code: exampleBean.setEmail(""). If you need to pass an NULL value then you can pass it as follows:

```
<bean id="..." class="exampleBean">
    <property name="email"><null/></property>
</bean>
```

The preceding example is equivalent to the Java code: exampleBean.setEmail(null)

Spring Beans Auto-Wiring

Y ou 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.

Mode	Description
no	This is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring. This is what you already have seen in Dependency Injection chapter.
byName	Autowiring by property name. Spring container looks at the properties of the beans on which *autowire* 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.
byType	Autowiring by property datatype. Spring container looks at the properties of 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 more than one such beans exists, a fatal exception is thrown.
constructor	Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.
autodetect	Spring first tries to wire using autowire by *constructor*, if it does not work, Spring tries to autowire by *byType*.

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.

Limitations	Description
Overriding possibility	You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring.
Primitive data types	You cannot autowire so-called simple properties such as primitives, Strings, and Classes.
Confusing nature	Autowiring is less exact than explicit wiring, so if possible prefer using explicit wiring.

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:

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 *TextEditor*, *SpellChecker* 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 **TextEditor.java** file:

```
package com.tutorialspoint;

public class TextEditor {
   private SpellChecker spellChecker;
   private String name;
```

```java
    public void setSpellChecker( SpellChecker spellChecker ){
        this.spellChecker = spellChecker;
    }
    public SpellChecker getSpellChecker() {
        return spellChecker;
    }

    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }

    public void spellCheck() {
        spellChecker.checkSpelling();
    }
}
```

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

```java
package com.tutorialspoint;

public class SpellChecker {
    public SpellChecker() {
        System.out.println("Inside SpellChecker constructor." );
    }

    public void checkSpelling() {
        System.out.println("Inside checkSpelling." );
    }

}
```

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

```java
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context =
                new ClassPathXmlApplicationContext("Beans.xml");

        TextEditor te = (TextEditor) context.getBean("textEditor");

        te.spellCheck();
    }
}
```

Following is the configuration file **Beans.xml** in normal condition:

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

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

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <!-- Definition for textEditor bean -->
    <bean id="textEditor" class="com.tutorialspoint.TextEditor">
        <property name="spellChecker" ref="spellChecker" />
        <property name="name" value="Generic Text Editor" />
    </bean>

    <!-- Definition for spellChecker bean -->
    <bean id="spellChecker" class="com.tutorialspoint.SpellChecker">
    </bean>

</beans>
```

But if you are going to use autowiring 'byName', then your XML configuration file will become as follows:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <!-- Definition for textEditor bean -->
    <bean id="textEditor" class="com.tutorialspoint.TextEditor"
        autowire="byName">
        <property name="name" value="Generic Text Editor" />
    </bean>

    <!-- Definition for spellChecker bean -->
    <bean id="spellChecker" class="com.tutorialspoint.SpellChecker">
    </bean>

</beans>
```

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:

```
Inside SpellChecker constructor.

Inside checkSpelling.
```

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:

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 *TextEditor*, *SpellChecker* 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 **TextEditor.java** file:

```java
package com.tutorialspoint;

public class TextEditor {
   private SpellChecker spellChecker;
   private String name;

   public void setSpellChecker( SpellChecker spellChecker ) {
      this.spellChecker = spellChecker;
   }
   public SpellChecker getSpellChecker() {
      return spellChecker;
   }

   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }

   public void spellCheck() {
      spellChecker.checkSpelling();
   }
}
```

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

```java
package com.tutorialspoint;

public class SpellChecker {
   public SpellChecker(){
      System.out.println("Inside SpellChecker constructor." );
   }

   public void checkSpelling() {
      System.out.println("Inside checkSpelling." );
   }
```

```
}
```

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

```java
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
            new ClassPathXmlApplicationContext("Beans.xml");

      TextEditor te = (TextEditor) context.getBean("textEditor");

      te.spellCheck();
   }
}
```

Following is the configuration file **Beans.xml** in normal condition:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.tutorialspoint.TextEditor">
      <property name="spellChecker" ref="spellChecker" />
      <property name="name" value="Generic Text Editor" />
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.tutorialspoint.SpellChecker">
   </bean>

</beans>
```

But if you are going to use autowiring 'byType', then your XML configuration file will become as follows:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.tutorialspoint.TextEditor"
      autowire="byType">
      <property name="name" value="Generic Text Editor" />
   </bean>
```

```
    <!-- Definition for spellChecker bean -->
    <bean id="SpellChecker" class="com.tutorialspoint.SpellChecker">
    </bean>

</beans>
```

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:

```
Inside SpellChecker constructor.

Inside checkSpelling.
```

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:

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 *TextEditor*, *SpellChecker* 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 **TextEditor.java** file:

```
package com.tutorialspoint;

public class TextEditor {
   private SpellChecker spellChecker;
   private String name;

   public TextEditor( SpellChecker spellChecker, String name ) {
      this.spellChecker = spellChecker;
      this.name = name;
   }
   public SpellChecker getSpellChecker() {
      return spellChecker;
```

```
    }
    public String getName() {
        return name;
    }

    public void spellCheck() {
        spellChecker.checkSpelling();
    }
}
```

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

```
package com.tutorialspoint;

public class SpellChecker {
    public SpellChecker(){
        System.out.println("Inside SpellChecker constructor." );
    }

    public void checkSpelling()
    {
        System.out.println("Inside checkSpelling." );
    }

}
```

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

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context =
                new ClassPathXmlApplicationContext("Beans.xml");

        TextEditor te = (TextEditor) context.getBean("textEditor");

        te.spellCheck();
    }
}
```

Following is the configuration file **Beans.xml** in normal condition:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd">

    <!-- Definition for textEditor bean -->
```

```
   <bean id="textEditor" class="com.tutorialspoint.TextEditor">
      <constructor-arg  ref="spellChecker" />
      <constructor-arg  value="Generic Text Editor"/>
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker"
class="com.tutorialspoint.SpellChecker">
   </bean>

</beans>
```

But if you are going to use autowiring 'by constructor', then your XML configuration file will become as follows:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd">

   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.tutorialspoint.TextEditor"
      autowire="constructor">
      <constructor-arg value="Generic Text Editor"/>
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="SpellChecker"
class="com.tutorialspoint.SpellChecker">
   </bean>

</beans>
```

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:

```
Inside SpellChecker constructor.

Inside checkSpelling.
```

Spring Annotation Based Configuration

Starting from Spring 2.5 it became possible to configure the dependency injection using **annotations**. So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches.

Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file. So consider to have following configuration file in case you want to use any annotation in your Spring application.

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

<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:annotation-config/>
   <!-- bean definitions go here -->

</beans>
```

Once **<context:annotation-config/>** is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors. Let us see few important annotations to understand how they work:

S.N.	Annotation & Description
1	@Required The @Required annotation applies to bean property setter methods.
2	@Autowired The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties.

3	@Qualifier The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wired.
4	JSR-250 Annotations Spring supports JSR-250 based annotations which include @Resource, @PostConstruct and @PreDestroy annotations.

Spring @Required Annotation

The **@Required** annotation applies to bean property setter methods and it indicates that the affected bean property must be populated in XML configuration file at configuration time otherwise the container throws a BeanInitializationException exception. Below is an example to show the use of @Required annotation.

Example

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 *Student* 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 **Student.java** file:

```java
package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Required;

public class Student {
   private Integer age;
   private String name;

   @Required
   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
      return age;
   }

   @Required
   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
```

```
        return name;
    }
}
```

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

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context =
                new ClassPathXmlApplicationContext("Beans.xml");

        Student student = (Student) context.getBean("student");
        System.out.println("Name : " + student.getName() );
        System.out.println("Age : " + student.getAge() );
    }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:annotation-config/>

   <!-- Definition for student bean -->
   <bean id="student" class="com.tutorialspoint.Student">
      <property name="name"  value="Zara" />

      <!-- try without passing age and check the result -->
      <!-- property name="age"  value="11"-->
   </bean>

</beans>
```

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 raise *BeanInitializationException* exception and print the following error along with other log messages:

```
Property 'age' is required for bean 'student'
```

Next, you can try above example after removing comment from 'age' property as follows:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-
3.0.xsd">

    <context:annotation-config/>

    <!-- Definition for student bean -->
    <bean id="student" class="com.tutorialspoint.Student">
        <property name="name"  value="Zara" />
        <property name="age"  value="11"/>
    </bean>

</beans>
```

Now above example will produce following result:

```
Name : Zara

Age : 11
```

Spring @Autowired Annotation

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.

Example

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 *TextEditor*, *SpellChecker* 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 **TextEditor.java** file:

```java
package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

public class TextEditor {
   private SpellChecker spellChecker;

   @Autowired
   public void setSpellChecker( SpellChecker spellChecker ){
      this.spellChecker = spellChecker;
   }
   public SpellChecker getSpellChecker( ) {
      return spellChecker;
   }
   public void spellCheck() {
      spellChecker.checkSpelling();
   }
}
```

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

```java
package com.tutorialspoint;

public class SpellChecker {
   public SpellChecker(){
      System.out.println("Inside SpellChecker constructor." );
   }

   public void checkSpelling(){
      System.out.println("Inside checkSpelling." );
   }

}
```

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

```java
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
             new ClassPathXmlApplicationContext("Beans.xml");

      TextEditor te = (TextEditor) context.getBean("textEditor");

      te.spellCheck();
   }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
3.0.xsd">

   <context:annotation-config/>

   <!-- Definition for textEditor bean without constructor-arg  -->
   <bean id="textEditor" class="com.tutorialspoint.TextEditor">
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker"class="com.tutorialspoint.SpellChecker">
   </bean>

</beans>
```

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:

```
Inside SpellChecker constructor.

Inside checkSpelling.
```

@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 your **TextEditor.java**file will become as follows:

```java
package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

public class TextEditor {
   @Autowired
   private SpellChecker spellChecker;

   public TextEditor() {
      System.out.println("Inside TextEditor constructor." );
   }
   public SpellChecker getSpellChecker( ){
      return spellChecker;
   }
   public void spellCheck(){
      spellChecker.checkSpelling();
   }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
3.0.xsd">

  <context:annotation-config/>

  <!-- Definition for textEditor bean -->
  <bean id="textEditor" class="com.tutorialspoint.TextEditor">
  </bean>

  <!-- Definition for spellChecker bean -->
  <bean id="spellChecker" class="com.tutorialspoint.SpellChecker">
  </bean>

</beans>
```

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

```
Inside TextEditor constructor.

Inside SpellChecker constructor.

Inside checkSpelling.
```

@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.

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

```java
package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

public class TextEditor {
   private SpellChecker spellChecker;

   @Autowired
   public TextEditor(SpellChecker spellChecker){
      System.out.println("Inside TextEditor constructor." );
      this.spellChecker = spellChecker;
   }
```

```
    public void spellCheck(){
        spellChecker.checkSpelling();
    }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
3.0.xsd">

  <context:annotation-config/>

  <!-- Definition for textEditor bean without constructor-arg  -->
  <bean id="textEditor" class="com.tutorialspoint.TextEditor">
  </bean>

  <!-- Definition for spellChecker bean -->
  <bean id="spellChecker" class="com.tutorialspoint.SpellChecker">
  </bean>

</beans>
```

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

```
Inside TextEditor constructor.

Inside SpellChecker constructor.

Inside checkSpelling.
```

@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.

The following example will work even if you do not pass any value for age property but still it will demand for name property. You can try this example yourself because this is similar to @Required annotation example except that only **Student.java** file has been changed.

```java
package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

public class Student {
    private Integer age;
    private String name;
```

```
   @Autowired(required=false)
   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
      return age;
   }

   @Autowired
   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }
}
```

Spring @Qualifier Annotation

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 case you can use **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired. Below is an example to show the use of @Qualifier annotation.

Example

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 *Student*, *Profile* 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 **Student.java** file:

```
package com.tutorialspoint;

public class Student {
   private Integer age;
   private String name;

   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
      return age;
   }

   public void setName(String name) {
```

```
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

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

```java
package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class Profile {
    @Autowired
    @Qualifier("student1")
    private Student student;

    public Profile(){
        System.out.println("Inside Profile constructor." );
    }

    public void printAge() {
        System.out.println("Age : " + student.getAge() );
    }

    public void printName() {
        System.out.println("Name : " + student.getName() );
    }
}
```

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

```java
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context =
                new ClassPathXmlApplicationContext("Beans.xml");

        Profile profile = (Profile) context.getBean("profile");

        profile.printAge();
        profile.printName();
    }
}
```

Consider the example of following configuration file **Beans.xml**:

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

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

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:annotation-config/>

   <!-- Definition for profile bean -->
   <bean id="profile" class="com.tutorialspoint.Profile">
   </bean>

   <!-- Definition for student1 bean -->
   <bean id="student1" class="com.tutorialspoint.Student">
      <property name="name"  value="Zara" />
      <property name="age"  value="11"/>
   </bean>

   <!-- Definition for student2 bean -->
   <bean id="student2" class="com.tutorialspoint.Student">
      <property name="name"  value="Nuha" />
      <property name="age"  value="2"/>
   </bean>

</beans>
```

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:

```
Inside Profile constructor.

Age : 11

Name : Zara
```

Spring JSR-250 Annotations

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

@PostConstruct and @PreDestroy Annotations

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.

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.

Example

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* 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:

```
package com.tutorialspoint;
import javax.annotation.*;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }

   public String getMessage(){
      System.out.println("Your Message : " + message);
      return message;
   }
   @PostConstruct
   public void init(){
      System.out.println("Bean is going through init.");
   }
   @PreDestroy
   public void destroy(){
      System.out.println("Bean will destroy now.");
   }
}
```

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.

```
package com.tutorialspoint;

import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {

      AbstractApplicationContext context =
                        new ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
```

```
      obj.getMessage();
      context.registerShutdownHook();
   }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-
3.0.xsd">

   <context:annotation-config/>

   <bean id="helloWorld"
       class="com.tutorialspoint.HelloWorld"
       init-method="init" destroy-method="destroy">
       <property name="message" value="Hello World!"/>
   </bean>

</beans>
```

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:

```
Bean is going through init.
Your Message : Hello World!
Bean will destroy now.
```

@Resource Annotation

You can use **@Resource** annotation on fields or setter methods and it works the same as in Java EE 5. The @Resource annotation takes a 'name' attribute which will be interpreted as the bean name to be injected. You can say, it follows **by-name** autowiring semantics as demonstrated in the below example:

```java
package com.tutorialspoint;

import javax.annotation.Resource;

public class TextEditor {
   private SpellChecker spellChecker;

   @Resource(name= "spellChecker")
   public void setSpellChecker( SpellChecker spellChecker ){
      this.spellChecker = spellChecker;
   }
   public SpellChecker getSpellChecker(){
      return spellChecker;
   }
   public void spellCheck(){
      spellChecker.checkSpelling();
```

```
   }
}
```

If no 'name' is specified explicitly, the default name is derived from the field name or setter method. In case of a field, it takes the field name; in case of a setter method, it takes the bean property name.

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 I will say it is really not required to learn how to proceed with Java based configuration because 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 but with the help of few Java-based annotations explained below.

@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. The simplest possible @Configuration class would be as follows:

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;

@Configuration
public class HelloWorldConfig {

   @Bean
   public HelloWorld helloWorld(){
      return new HelloWorld();
   }
}
```

Above code will be equivalent to the following XML configuration:

```
<beans>
   <bean id="helloWorld" class="com.tutorialspoint.HelloWorld" />
</beans>
```

Here the method name annotated with @Bean works as bean ID and it creates and returns actual bean. Your configuration class can have declaration for more than one @Bean. Once your configuration classes are defined, you can load & provide them to Spring container using*AnnotationConfigApplicationContext* as follows:

```
public static void main(String[] args) {
   ApplicationContext ctx =
```

```
   new AnnotationConfigApplicationContext(HelloWorldConfig.class);

   HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

   helloWorld.setMessage("Hello World!");
   helloWorld.getMessage();
}
```

You can load various configuration classes as follows:

```
public static void main(String[] args) {
   AnnotationConfigApplicationContext ctx =
   new AnnotationConfigApplicationContext();

   ctx.register(AppConfig.class, OtherConfig.class);
   ctx.register(AdditionalConfig.class);
   ctx.refresh();

   MyService myService = ctx.getBean(MyService.class);
   myService.doStuff();
}
```

Example

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	Because you are using Java-based annotations, so you also need to add *CGLIB.jar* from your Java installation directory and *ASM.jar* library which can be downloaded from *asm.ow2.org*.
4	Create Java classes *HelloWorldConfig*, *HelloWorld* and *MainApp* under the *com.tutorialspoint*package.
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 **HelloWorldConfig.java** file:

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;

@Configuration
public class HelloWorldConfig {

   @Bean
   public HelloWorld helloWorld(){
      return new HelloWorld();
   }
}
```

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

```
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }

   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

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

```
package com.tutorialspoint;

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

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext ctx =
      new AnnotationConfigApplicationContext(HelloWorldConfig.class);

      HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

      helloWorld.setMessage("Hello World!");
      helloWorld.getMessage();
   }
}
```

Once you are done with creating all the source filesand adding required additional libraries, let us run the application. You should note that there is no configuration file required. If everything is fine with your application, this will print the following message:

```
Your Message : Hello World!
```

Injecting Bean Dependencies

When @Beans have dependencies on one another, expressing that dependency is as simple as having one bean method calling another as follows:

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;

@Configuration
public class AppConfig {
   @Bean
   public Foo foo() {
      return new Foo(bar());
   }
   @Bean
   public Bar bar() {
      return new Bar();
```

```
   }
}
```

Here, the foo bean receives a reference to bar via constructor injection. Now let us see one working example:

Example

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	Because you are using Java-based annotations, so you also need to add *CGLIB.jar* from your Java installation directory and *ASM.jar* library which can be downloaded from *asm.ow2.org*.
4	Create Java classes *TextEditorConfig*, *TextEditor*, *SpellChecker* and *MainApp* under the*com.tutorialspoint* package.
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 **TextEditorConfig.java** file:

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;

@Configuration
public class TextEditorConfig {

   @Bean
   public TextEditor textEditor(){
      return new TextEditor( spellChecker() );
   }

   @Bean
   public SpellChecker spellChecker(){
      return new SpellChecker( );
   }
}
```

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

```
package com.tutorialspoint;

public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor(SpellChecker spellChecker){
      System.out.println("Inside TextEditor constructor." );
```

```
        this.spellChecker = spellChecker;
    }
    public void spellCheck(){
        spellChecker.checkSpelling();
    }
}
```

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

```
package com.tutorialspoint;

public class SpellChecker {
    public SpellChecker(){
        System.out.println("Inside SpellChecker constructor." );
    }

    public void checkSpelling(){
        System.out.println("Inside checkSpelling." );
    }

}
```

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

```
package com.tutorialspoint;

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

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext ctx =
        new AnnotationConfigApplicationContext(TextEditorConfig.class);

        TextEditor te = ctx.getBean(TextEditor.class);

        te.spellCheck();
    }
}
```

Once you are done with creating all the source files and adding required additional libraries, let us run the application. You should note that there is no configuration file required. If everything is fine with your application, this will print the following message:

```
Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.
```

The @Import Annotation

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 a() {
      return new A();
   }
}
```

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();
    }
}
```

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:

S.N.	Spring Built-in Events & Description
1	**ContextRefreshedEvent** This event is published when the *ApplicationContext* is either initialized or refreshed. This can also be raised using the refresh() method on the *ConfigurableApplicationContext* interface.
2	**ContextStartedEvent** This event is published when the *ApplicationContext* is started using the start() method on the *ConfigurableApplicationContext* interface. You can poll your database or you can re/start any stopped application after receiving this event.
3	**ContextStoppedEvent** This event is published when the *ApplicationContext* is stopped using the stop() method on the *ConfigurableApplicationContext* interface. You can do required housekeep work after receiving this event.
4	**ContextClosedEvent** This event is published when the *ApplicationContext* is closed using the close() method on the *ConfigurableApplicationContext* interface. A closed context reaches its end of life; it cannot be refreshed or restarted.
5	**RequestHandledEvent** This is a web-specific event telling all beans that an HTTP request has been serviced.

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:

```java
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }

   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

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

```java
package com.tutorialspoint;

import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextStartedEvent;

public class CStartEventHandler
   implements ApplicationListener<ContextStartedEvent>{

   public void onApplicationEvent(ContextStartedEvent event) {
      System.out.println("ContextStartedEvent Received");
   }
}
```

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

```
package com.tutorialspoint;

import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextStoppedEvent;

public class CStopEventHandler
   implements ApplicationListener<ContextStoppedEvent>{

   public void onApplicationEvent(ContextStoppedEvent event) {
      System.out.println("ContextStoppedEvent Received");
   }
}
```

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

```
package com.tutorialspoint;

import org.springframework.context.ConfigurableApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ConfigurableApplicationContext context =
      new ClassPathXmlApplicationContext("Beans.xml");

      // Let us raise a start event.
      context.start();

      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

      obj.getMessage();

      // Let us raise a stop event.
      context.stop();
   }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id="helloWorld" class="com.tutorialspoint.HelloWorld">
      <property name="message" value="Hello World!"/>
   </bean>

   <bean id="cStartEventHandler"
         class="com.tutorialspoint.CStartEventHandler"/>

   <bean id="cStopEventHandler"
         class="com.tutorialspoint.CStopEventHandler"/>
```

```
</beans>
```

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:

```
ContextStartedEvent Received

Your Message : Hello World!

ContextStoppedEvent Received
```

If you like, you can publish your own custom events and later you can capture the same to take any action against those custom events.

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:

```java
package com.tutorialspoint;

import org.springframework.context.ApplicationEvent;

public class CustomEvent extends ApplicationEvent{

   public CustomEvent(Object source) {
      super(source);
   }
}
```

```
    public String toString(){
        return "My Custom Event";
    }
}
```

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

```
package com.tutorialspoint;

import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;

public class CustomEventPublisher
    implements ApplicationEventPublisherAware {

    private ApplicationEventPublisher publisher;

    public void setApplicationEventPublisher
                (ApplicationEventPublisher publisher){
        this.publisher = publisher;
    }

    public void publish() {
        CustomEvent ce = new CustomEvent(this);
        publisher.publishEvent(ce);
    }
}
```

Following is the content of the **CustomEventHandler.java** file.

```
package com.tutorialspoint;

import org.springframework.context.ApplicationListener;

public class CustomEventHandler
    implements ApplicationListener<CustomEvent>{

    public void onApplicationEvent(CustomEvent event) {
        System.out.println(event.toString());
    }

}
```

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

```
package com.tutorialspoint;

import org.springframework.context.ConfigurableApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
    public static void main(String[] args) {
        ConfigurableApplicationContext context =
        new ClassPathXmlApplicationContext("Beans.xml");

        CustomEventPublisher cvp =
        (CustomEventPublisher) context.getBean("customEventPublisher");
```

```
        cvp.publish();
        cvp.publish();
    }
}
```

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

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="customEventHandler"
        class="com.tutorialspoint.CustomEventHandler"/>

    <bean id="customEventPublisher"
        class="com.tutorialspoint.CustomEventPublisher"/>

</beans>
```

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:

```
My Custom Event

My Custom Event
```

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.

Terms	Description
Aspect	A module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement.
Join point	This represents a point in your application where you can plug-in AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.
Advice	This is the actual action to be taken either before or after the method execution. This is actual piece of code that is invoked during program execution by Spring AOP framework.
Pointcut	This is a set of one or more joinpoints where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples.
Introduction	An introduction allows you to add new methods or attributes to existing classes.

Target object	The object being advised by one or more aspects, this object will always be a proxied object. Also referred to as the advised object.
Weaving	Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime.

Types of Advice

Spring aspects can work with five kinds of advice mentioned below:

Advice	Description
before	Run advice before the a method execution.
after	Run advice after the a method execution regardless of its outcome.
after-returning	Run advice after the a method execution only if method completes successfully.
after-throwing	Run advice after the a method execution only if method exits by throwing an exception.
around	Run advice before and after the advised method is invoked.

Custom Aspects Implementation

Spring supports the **@AspectJ annotation style** approach and the **schema-based** approach to implement custom aspects. These two approaches have been explained in detail in the following two sub chapters

Approach	Description
XML Schema based	Aspects are implemented using regular classes along with XML based configuration.
@AspectJ based	@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations.

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:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

   <!-- bean definition & AOP specific configuration -->

</beans>
```

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:

```
<aop:config>
   <aop:aspect id="myAspect" ref="aBean">
   ...
   </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
...
</bean>
```

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:

```
<aop:config>
   <aop:aspect id="myAspect" ref="aBean">

   <aop:pointcut id="businessService"
      expression="execution(* com.xyz.myapp.service.*.*(..))"/>
   ...
   </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
...
</bean>
```

The following example defines a pointcut named 'businessService' that will match the execution of getName() method available in Student class under the package com.tutorialspoint:

```
<aop:config>
   <aop:aspect id="myAspect" ref="aBean">

   <aop:pointcut id="businessService"
      expression="execution(*
com.tutorialspoint.Student.getName(..))"/>
   ...
   </aop:aspect>
```

```
</aop:config>

<bean id="aBean" class="...">
...
</bean>
```

Declaring advices

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

```
<aop:config>
   <aop:aspect id="myAspect" ref="aBean">
      <aop:pointcut id="businessService"
         expression="execution(* com.xyz.myapp.service.*.*(..))"/>

      <!-- a before advice definition -->
      <aop:before pointcut-ref="businessService"
         method="doRequiredTask"/>

      <!-- an after advice definition -->
      <aop:after pointcut-ref="businessService"
         method="doRequiredTask"/>

      <!-- an after-returning advice definition -->
      <!--The doRequiredTask method must have parameter named retVal --
>
      <aop:after-returning pointcut-ref="businessService"
         returning="retVal"
         method="doRequiredTask"/>

      <!-- an after-throwing advice definition -->
      <!--The doRequiredTask method must have parameter named ex -->
      <aop:after-throwing pointcut-ref="businessService"
         throwing="ex"
         method="doRequiredTask"/>

      <!-- an around advice definition -->
      <aop:around pointcut-ref="businessService"
         method="doRequiredTask"/>
   ...
   </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
...
</bean>
```

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

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:

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 AOP specific libraries **aspectjrt.jar, aspectjweaver.jar** and **aspectj.jar** in the project.
4	Create Java classes **Logging**, *Student* and *MainApp* under the *com.tutorialspoint* package.
5	Create Beans configuration file *Beans.xml* under the **src** folder.
6	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 **Logging.java** file. This is actually a sample of aspect module which defines methods to be called at various points.

```java
package com.tutorialspoint;

public class Logging {

   /**
    * This is the method which I would like to execute
    * before a selected method execution.
    */
   public void beforeAdvice(){
      System.out.println("Going to setup student profile.");
   }

   /**
    * This is the method which I would like to execute
    * after a selected method execution.
    */
   public void afterAdvice(){
      System.out.println("Student profile has been setup.");
   }

   /**
    * This is the method which I would like to execute
    * when any method returns.
    */
   public void afterReturningAdvice(Object retVal){
      System.out.println("Returning:" + retVal.toString() );
   }

   /**
    * This is the method which I would like to execute
    * if there is an exception raised.
    */
   public void AfterThrowingAdvice(IllegalArgumentException ex){
      System.out.println("There has been an exception: " +
ex.toString());
   }

}
```

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

```java
package com.tutorialspoint;

public class Student {
   private Integer age;
   private String name;

   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
         System.out.println("Age : " + age );
      return age;
   }

   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      System.out.println("Name : " + name );
      return name;
   }

   public void printThrowException(){
         System.out.println("Exception raised");
       throw new IllegalArgumentException();
   }
}
```

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

```java
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
            new ClassPathXmlApplicationContext("Beans.xml");

      Student student = (Student) context.getBean("student");

      student.getName();
      student.getAge();

      student.printThrowException();
   }
}
```

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

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
```

```
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

<aop:config>
    <aop:aspect id="log" ref="logging">
        <aop:pointcut id="selectAll"
        expression="execution(* com.tutorialspoint.*.*(..))"/>
        <aop:before pointcut-ref="selectAll" method="beforeAdvice"/>
        <aop:after pointcut-ref="selectAll" method="afterAdvice"/>
        <aop:after-returning pointcut-ref="selectAll"
                                returning="retVal"
                                method="afterReturningAdvice"/>
        <aop:after-throwing pointcut-ref="selectAll"
                                throwing="ex"
                                method="AfterThrowingAdvice"/>
    </aop:aspect>
</aop:config>

<!-- Definition for student bean -->
<bean id="student" class="com.tutorialspoint.Student">
    <property name="name"  value="Zara" />
    <property name="age"   value="11"/>
</bean>

<!-- Definition for logging aspect -->
<bean id="logging" class="com.tutorialspoint.Logging"/>

</beans>
```

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:

```
Going to setup student profile.

Name : Zara

Student profile has been setup.

Returning:Zara

Going to setup student profile.

Age : 11

Student profile has been setup.

Returning:11

Going to setup student profile.

Exception raised

Student profile has been setup.

There has been an exception: java.lang.IllegalArgumentException

.....

other exception content
```

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:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

   <aop:config>
   <aop:aspect id="log" ref="logging">
     <aop:pointcut id="selectAll"
     expression="execution(* com.tutorialspoint.Student.getName(..))"/>
     <aop:before pointcut-ref="selectAll" method="beforeAdvice"/>
     <aop:after pointcut-ref="selectAll" method="afterAdvice"/>
   </aop:aspect>
   </aop:config>

   <!-- Definition for student bean -->
   <bean id="student" class="com.tutorialspoint.Student">
      <property name="name"  value="Zara" />
      <property name="age"  value="11"/>
   </bean>

   <!-- Definition for logging aspect -->
   <bean id="logging" class="com.tutorialspoint.Logging"/>

</beans>
```

If you will execute sample application with these configuration changes, this will print the following message:

```
Going to setup student profile.

Name : Zara

Student profile has been setup.

Age : 11

Exception raised

.....

other exception content
```

@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:

```
package org.xyz;

import org.aspectj.lang.annotation.Aspect;

@Aspect
public class AspectModule {

}
```

They will be configured in XML like any other bean as follows:

```
<bean id="myAspect" class="org.xyz.AspectModule">
   <!-- configure properties of aspect here as normal -->
</bean>
```

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:

```
import org.aspectj.lang.annotation.Pointcut;

@Pointcut("execution(* com.xyz.myapp.service.*.*(..))") // expression
private void businessService() {}  // signature
```

The following example defines a pointcut named 'getname' that will match the execution of getName() method available in Student class under the package com.tutorialspoint:

```
import org.aspectj.lang.annotation.Pointcut;
```

```
@Pointcut("execution(* com.tutorialspoint.Student.getName(..))")
private void getname() {}
```

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():

```
@Before("businessService()")
public void doBeforeTask(){
 ...
}

@After("businessService()")
public void doAfterTask(){
 ...
}

@AfterReturning(pointcut = "businessService()", returning="retVal")
public void doAfterReturnningTask(Object retVal){
  // you can intercept retVal here.
 ...
}

@AfterThrowing(pointcut = "businessService()", throwing="ex")
public void doAfterThrowingTask(Exception ex){
  // you can intercept thrown exception here.
 ...
}

@Around("businessService()")
public void doAroundTask(){
 ...
}
```

You can define you pointcut inline for any of the advices. Below is an example to define inline
pointcut for before advice:

```
@Before("execution(* com.xyz.myapp.service.*.*(..))")
public doBeforeTask(){
 ...
}
```

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:

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 AOP specific libraries **aspectjrt.jar, aspectjweaver.jar** and **aspectj.jar** in the project.
4	Create Java classes **Logging**, *Student* and *MainApp* under the *com.tutorialspoint* package.
5	Create Beans configuration file *Beans.xml* under the **src** folder.
6	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 **Logging.java** file. This is actually a sample of aspect module which defines methods to be called at various points.

```java
package com.tutorialspoint;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Around;

@Aspect
public class Logging {

   /** Following is the definition for a pointcut to select
    *  all the methods available. So advice will be called
    *  for all the methods.
    */
   @Pointcut("execution(* com.tutorialspoint.*.*(..))")
   private void selectAll(){}

   /**
    * This is the method which I would like to execute
    * before a selected method execution.
    */
   @Before("selectAll()")
   public void beforeAdvice(){
      System.out.println("Going to setup student profile.");
   }

   /**
    * This is the method which I would like to execute
    * after a selected method execution.
    */
   @After("selectAll()")
   public void afterAdvice(){
      System.out.println("Student profile has been setup.");
   }

   /**
    * This is the method which I would like to execute
    * when any method returns.
    */
   @AfterReturning(pointcut = "selectAll()", returning="retVal")
   public void afterReturningAdvice(Object retVal){
      System.out.println("Returning:" + retVal.toString() );
   }
```

```
    /**
     * This is the method which I would like to execute
     * if there is an exception raised by any method.
     */
    @AfterThrowing(pointcut = "selectAll()", throwing = "ex")
    public void AfterThrowingAdvice(IllegalArgumentException ex){
        System.out.println("There has been an exception: " +
ex.toString());
    }

}
```

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

```
package com.tutorialspoint;

public class Student {
   private Integer age;
   private String name;

   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
         System.out.println("Age : " + age );
      return age;
   }

   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      System.out.println("Name : " + name );
      return name;
   }
   public void printThrowException(){
      System.out.println("Exception raised");
      throw new IllegalArgumentException();
   }
}
```

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

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
            new ClassPathXmlApplicationContext("Beans.xml");

      Student student = (Student) context.getBean("student");

      student.getName();
```

```
      student.getAge();

      student.printThrowException();
   }
}
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

   <aop:aspectj-autoproxy/>

   <!-- Definition for student bean -->
   <bean id="student" class="com.tutorialspoint.Student">
      <property name="name"  value="Zara" />
      <property name="age"  value="11"/>
   </bean>

   <!-- Definition for logging aspect -->
   <bean id="logging" class="com.tutorialspoint.Logging"/>

</beans>
```

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:

```
Going to setup student profile.

Name : Zara

Student profile has been setup.

Returning:Zara

Going to setup student profile.

Age : 11

Student profile has been setup.

Returning:11

Going to setup student profile.

Exception raised

Student profile has been setup.

There has been an exception: java.lang.IllegalArgumentException

.....
```

```
other exception content
```

Spring JDBC Framework

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.

```
CREATE TABLE Student(
   ID   INT NOT NULL AUTO_INCREMENT,
   NAME VARCHAR(20) NOT NULL,
   AGE  INT NOT NULL,
   PRIMARY KEY (ID)
);
```

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:

```xml
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSour
ce">
   <property name="driverClassName"
value="com.mysql.jdbc.Driver"/>
   <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
   <property name="username" value="root"/>
   <property name="password" value="password"/>
</bean>
```

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:

```java
String SQL = "select count(*) from Student";
int rowCount = jdbcTemplateObject.queryForInt( SQL );
```

Querying for a long:

```java
String SQL = "select count(*) from Student";
long rowCount = jdbcTemplateObject.queryForLong( SQL );
```

A simple query using a bind variable:

```java
String SQL = "select age from Student where id = ?";
int age = jdbcTemplateObject.queryForInt(SQL, new Object[]{10});
```

Querying for a String:

```java
String SQL = "select name from Student where id = ?";
String name = jdbcTemplateObject.queryForObject(SQL, new Object[]{10},
String.class);
```

Querying and returning an object:

```java
String SQL = "select * from Student where id = ?";
```

```
Student student = jdbcTemplateObject.queryForObject(SQL,
                    new Object[]{10}, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException
{
      Student student = new Student();
      student.setID(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

Querying and returning multiple objects:

```
String SQL = "select * from Student";
List<Student> students = jdbcTemplateObject.query(SQL,
                         new StudentMapper());

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException
{
      Student student = new Student();
      student.setID(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

Inserting a row into the table:

```
String SQL = "insert into Student (name, age) values (?, ?)";
jdbcTemplateObject.update( SQL, new Object[]{"Zara", 11} );
```

Updating a row into the table:

```
String SQL = "update Student set name = ? where id = ?";
jdbcTemplateObject.update( SQL, new Object[]{"Zara", 10} );
```

Deletng a row from the table:

```
String SQL = "delete Student where id = ?";
jdbcTemplateObject.update( SQL, new Object[]{20} );
```

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:

```
String SQL = "CREATE TABLE Student( " +
   "ID   INT NOT NULL AUTO_INCREMENT, " +
   "NAME VARCHAR(20) NOT NULL, " +
   "AGE  INT NOT NULL, " +
   "PRIMARY KEY (ID));"
```

```
jdbcTemplateObject.execute( SQL );
```

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.

```
CREATE TABLE Student(
   ID   INT NOT NULL AUTO_INCREMENT,
   NAME VARCHAR(20) NOT NULL,
   AGE  INT NOT NULL,
   PRIMARY KEY (ID)
);
```

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**:

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;

public interface StudentDAO {
   /**
    * This is the method to be used to initialize
    * database resources ie. connection.
```

```
   */
   public void setDataSource(DataSource ds);
   /**
    * This is the method to be used to create
    * a record in the Student table.
    */
   public void create(String name, Integer age);
   /**
    * This is the method to be used to list down
    * a record from the Student table corresponding
    * to a passed student id.
    */
   public Student getStudent(Integer id);
   /**
    * This is the method to be used to list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();
   /**
    * This is the method to be used to delete
    * a record from the Student table corresponding
    * to a passed student id.
    */
   public void delete(Integer id);
   /**
    * This is the method to be used to update
    * a record into the Student table.
    */
   public void update(Integer id, Integer age);
}
```

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

```
package com.tutorialspoint;

public class Student {
   private Integer age;
   private String name;
   private Integer id;

   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
      return age;
   }

   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }

   public void setId(Integer id) {
      this.id = id;
   }
   public Integer getId() {
      return id;
   }
```

```
}
```

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

```java
package com.tutorialspoint;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException
{
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```java
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJDBCTemplate implements StudentDAO {
   private DataSource dataSource;
   private JdbcTemplate jdbcTemplateObject;

   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      this.jdbcTemplateObject = new JdbcTemplate(dataSource);
   }

   public void create(String name, Integer age) {
      String SQL = "insert into Student (name, age) values (?, ?)";

      jdbcTemplateObject.update( SQL, name, age);
      System.out.println("Created Record Name=" + name + " Age=" +
age);
      return;
   }

   public Student getStudent(Integer id) {
      String SQL = "select * from Student where id = ?";
      Student student = jdbcTemplateObject.queryForObject(SQL,
                     new Object[]{id}, new StudentMapper());
      return student;
   }

   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL,
                           new StudentMapper());
```

```
        return students;
    }

    public void delete(Integer id){
        String SQL = "delete from Student where id = ?";
        jdbcTemplateObject.update(SQL, id);
        System.out.println("Deleted Record with ID = " + id );
        return;
    }

    public void update(Integer id, Integer age){
        String SQL = "update Student set age = ? where id = ?";
        jdbcTemplateObject.update(SQL, age, id);
        System.out.println("Updated Record with ID = " + id );
        return;
    }

}
```

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

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.StudentJDBCTemplate;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context =
                new ClassPathXmlApplicationContext("Beans.xml");

        StudentJDBCTemplate studentJDBCTemplate =
        (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

        System.out.println("------Records Creation--------" );
        studentJDBCTemplate.create("Zara", 11);
        studentJDBCTemplate.create("Nuha", 2);
        studentJDBCTemplate.create("Ayan", 15);

        System.out.println("------Listing Multiple Records--------" );
        List<Student> students = studentJDBCTemplate.listStudents();
        for (Student record : students) {
            System.out.print("ID : " + record.getId() );
            System.out.print(", Name : " + record.getName() );
            System.out.println(", Age : " + record.getAge());
        }

        System.out.println("----Updating Record with ID = 2 -----" );
        studentJDBCTemplate.update(2, 20);

        System.out.println("----Listing Record with ID = 2 -----" );
        Student student = studentJDBCTemplate.getStudent(2);
        System.out.print("ID : " + student.getId() );
        System.out.print(", Name : " + student.getName() );
        System.out.println(", Age : " + student.getAge());
```

```
    }
}
```

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

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

    <!-- Initialization for data source -->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
        <property name="username" value="root"/>
        <property name="password" value="password"/>
    </bean>

    <!-- Definition for studentJDBCTemplate bean -->
    <bean id="studentJDBCTemplate"
        class="com.tutorialspoint.StudentJDBCTemplate">
        <property name="dataSource"  ref="dataSource" />
    </bean>

</beans>
```

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:

```
DELIMITER $$


DROP PROCEDURE IF EXISTS `TEST`.`getRecord` $$

CREATE PROCEDURE `TEST`.`getRecord` (

IN in_id INTEGER,

OUT out_name VARCHAR(20),

OUT out_age  INTEGER)

BEGIN

   SELECT name, age

   INTO out_name, out_age

   FROM Student where id = in_id;

END $$


DELIMITER ;
```

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:

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 given 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**:

```java
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;

public interface StudentDAO {
   /**
    * This is the method to be used to initialize
    * database resources ie. connection.
    */
   public void setDataSource(DataSource ds);
   /**
    * This is the method to be used to create
    * a record in the Student table.
    */
   public void create(String name, Integer age);
   /**
    * This is the method to be used to list down
    * a record from the Student table corresponding
    * to a passed student id.
    */
   public Student getStudent(Integer id);
   /**
    * This is the method to be used to list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();

}
```

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

```java
package com.tutorialspoint;

public class Student {
   private Integer age;
   private String name;
```

```
   private Integer id;

   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
      return age;
   }

   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }

   public void setId(Integer id) {
      this.id = id;
   }
   public Integer getId() {
      return id;
   }
}
```

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

```
package com.tutorialspoint;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException
{
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.Map;

import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;

public class StudentJDBCTemplate implements StudentDAO {
   private DataSource dataSource;
   private SimpleJdbcCall jdbcCall;
```

```
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
        this.jdbcCall =  new SimpleJdbcCall(dataSource).
                          withProcedureName("getRecord");
    }

    public void create(String name, Integer age) {
        JdbcTemplate jdbcTemplateObject = new JdbcTemplate(dataSource);
        String SQL = "insert into Student (name, age) values (?, ?)";

        jdbcTemplateObject.update( SQL, name, age);
        System.out.println("Created Record Name = " + name + " Age = " +
age);
        return;
    }

    public Student getStudent(Integer id) {
        SqlParameterSource in = new MapSqlParameterSource().
                                addValue("in_id", id);
        Map<String, Object> out = jdbcCall.execute(in);

        Student student = new Student();
        student.setId(id);
        student.setName((String) out.get("out_name"));
        student.setAge((Integer) out.get("out_age"));

        return student;
    }

    public List<Student> listStudents() {
        String SQL = "select * from Student";

        List <Student> students = jdbcTemplateObject.query(SQL,
                                      new StudentMapper());
        return students;
    }

}
```

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:

```
package com.tutorialspoint;
import java.util.List;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.StudentJDBCTemplate;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
            new ClassPathXmlApplicationContext("Beans.xml");

      StudentJDBCTemplate studentJDBCTemplate =
```

```
        (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

        System.out.println("------Records Creation-------" );
        studentJDBCTemplate.create("Zara", 11);
        studentJDBCTemplate.create("Nuha", 2);
        studentJDBCTemplate.create("Ayan", 15);

        System.out.println("------Listing Multiple Records-------" );
        List<Student> students = studentJDBCTemplate.listStudents();
        for (Student record : students) {
            System.out.print("ID : " + record.getId() );
            System.out.print(", Name : " + record.getName() );
            System.out.println(", Age : " + record.getAge());
        }

        System.out.println("----Listing Record with ID = 2 -----" );
        Student student = studentJDBCTemplate.getStudent(2);
        System.out.print("ID : " + student.getId() );
        System.out.print(", Name : " + student.getName() );
        System.out.println(", Age : " + student.getAge());

    }
}
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

    <!-- Initialization for data source -->
    <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
        <property name="username" value="root"/>
        <property name="password" value="password"/>
    </bean>

    <!-- Definition for studentJDBCTemplate bean -->
    <bean id="studentJDBCTemplate"
        class="com.tutorialspoint.StudentJDBCTemplate">
        <property name="dataSource"  ref="dataSource" />
    </bean>

</beans>
```

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

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

ID : 2, Name : Nuha, Age : 2
```

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**:

1. **Atomicity:** A transaction should be treated as a single unit of operation which means either the entire sequence of operations is successful or unsuccessful.
2. **Consistency:** This represents the consistency of the referential integrity of the database, unique primary keys in tables etc.
3. **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.
4. **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:

1. Programmatic transaction management: This means that you have manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.

2. Declarative transaction management: 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:

```
public interface PlatformTransactionManager {
   TransactionStatus getTransaction(TransactionDefinition definition);
   throws TransactionException;
   void commit(TransactionStatus status) throws TransactionException;
   void rollback(TransactionStatus status) throws TransactionException;
}
```

S.N.	Method & Description
1	**TransactionStatus getTransaction(TransactionDefinition definition)** This method returns a currently active transaction or create a new one, according to the specified propagation behavior.
2	**void commit(TransactionStatus status)** This method commits the given transaction, with regard to its status.
3	**void rollback(TransactionStatus status)** This method performs a rollback of the given transaction.

The *TransactionDefinition* is the core interface of the transaction support in Spring and it is defined as below:

```
public interface TransactionDefinition {
   int getPropagationBehavior();
   int getIsolationLevel();
   String getName();
   int getTimeout();
   boolean isReadOnly();
}
```

S.N.	Method & Description
1	**int getPropagationBehavior()** This method returns the propagation behavior. Spring offers all of the transaction propagation options familiar from EJB CMT.
2	**int getIsolationLevel()** This method returns the degree to which this transaction is isolated from the work of other transactions.
3	**String getName()** This method returns the name of this transaction.
4	**int getTimeout()** This method returns the time in seconds in which the transaction must complete.
5	**boolean isReadOnly()** This method returns whether the transaction is read-only.

Following are the possible values for isolation level:

S.N.	Isolation & Description
1	**TransactionDefinition.ISOLATION_DEFAULT** This is the default isolation level.
2	**TransactionDefinition.ISOLATION_READ_COMMITTED** Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
3	**TransactionDefinition.ISOLATION_READ_UNCOMMITTED** Indicates that dirty reads, non-repeatable reads and phantom reads can occur.
4	**TransactionDefinition.ISOLATION_REPEATABLE_READ** Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
5	**TransactionDefinition.ISOLATION_SERIALIZABLE** Indicates that dirty reads, non-repeatable reads and phantom reads are prevented.

Following are the possible values for propagation types:

S.N.	Propagation & Description
1	**TransactionDefinition.PROPAGATION_MANDATORY** Support a current transaction; throw an exception if no current transaction exists.
2	**TransactionDefinition.PROPAGATION_NESTED** Execute within a nested transaction if a current transaction exists.
3	**TransactionDefinition.PROPAGATION_NEVER** Do not support a current transaction; throw an exception if a current transaction exists.
4	**TransactionDefinition.PROPAGATION_NOT_SUPPORTED** Do not support a current transaction; rather always execute non-transactionally.
5	**TransactionDefinition.PROPAGATION_REQUIRED** Support a current transaction; create a new one if none exists.

6	**TransactionDefinition.PROPAGATION_REQUIRES_NEW** Create a new transaction, suspending the current transaction if one exists.
7	**TransactionDefinition.PROPAGATION_SUPPORTS** Support a current transaction; execute non-transactionally if none exists.
8	**TransactionDefinition.TIMEOUT_DEFAULT** Use the default timeout of the underlying transaction system, or none if timeouts are not supported.

The *TransactionStatus* interface provides a simple way for transactional code to control transaction execution and query transaction status.

```
public interface TransactionStatus extends SavepointManager {
    boolean isNewTransaction();
    boolean hasSavepoint();
    void setRollbackOnly();
    boolean isRollbackOnly();
    boolean isCompleted();
}
```

S.N.	Method & Description
1	**boolean hasSavepoint()** This method returns whether this transaction internally carries a savepoint, that is, has been created as nested transaction based on a savepoint.
2	**boolean isCompleted()** This method returns whether this transaction is completed, that is, whether it has already been committed or rolled back.
3	**boolean isNewTransaction()** This method returns true in case the present transaction is new.
4	**boolean isRollbackOnly()** This method returns whether the transaction has been marked as rollback-only.
5	**void setRollbackOnly()** This method sets the transaction rollback-only.

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**:

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;

public interface StudentDAO {
```

```
   /**
    * This is the method to be used to initialize
    * database resources ie. connection.
    */
   public void setDataSource(DataSource ds);
   /**
    * This is the method to be used to create
    * a record in the Student and Marks tables.
    */
   public void create(String name, Integer age, Integer marks, Integer
year);
   /**
    * This is the method to be used to list down
    * all the records from the Student and Marks tables.
    */
   public List<StudentMarks> listStudents();
}
```

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

```
package com.tutorialspoint;

public class StudentMarks {
   private Integer age;
   private String name;
   private Integer id;
   private Integer marks;
   private Integer year;
   private Integer sid;

   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
      return age;
   }

   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }

   public void setId(Integer id) {
      this.id = id;
   }
   public Integer getId() {
      return id;
   }
   public void setMarks(Integer marks) {
      this.marks = marks;
   }
   public Integer getMarks() {
      return marks;
   }

   public void setYear(Integer year) {
      this.year = year;
   }
```

```java
    public Integer getYear() {
        return year;
    }

    public void setSid(Integer sid) {
        this.sid = sid;
    }
    public Integer getSid() {
        return sid;
    }
}
```

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

```java
package com.tutorialspoint;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class StudentMarksMapper implements RowMapper<StudentMarks> {
    public StudentMarks mapRow(ResultSet rs, int rowNum) throws
SQLException {

        StudentMarks studentMarks = new StudentMarks();

        studentMarks.setId(rs.getInt("id"));
        studentMarks.setName(rs.getString("name"));
        studentMarks.setAge(rs.getInt("age"));
        studentMarks.setSid(rs.getInt("sid"));
        studentMarks.setMarks(rs.getInt("marks"));
        studentMarks.setYear(rs.getInt("year"));

        return studentMarks;
    }
}
```

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

```java
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import
org.springframework.transaction.support.DefaultTransactionDefinition;

public class StudentJDBCTemplate implements StudentDAO {
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplateObject;
    private PlatformTransactionManager transactionManager;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
```

```
      this.jdbcTemplateObject = new JdbcTemplate(dataSource);
   }

   public void setTransactionManager(
      PlatformTransactionManager transactionManager) {
      this.transactionManager = transactionManager;
   }

   public void create(String name, Integer age, Integer marks, Integer
year){

      TransactionDefinition def = new DefaultTransactionDefinition();
      TransactionStatus status =
transactionManager.getTransaction(def);

      try {
         String SQL1 = "insert into Student (name, age) values (?, ?)";
         jdbcTemplateObject.update( SQL1, name, age);

         // Get the latest student id to be used in Marks table
         String SQL2 = "select max(id) from Student";
         int sid = jdbcTemplateObject.queryForInt( SQL2 );

         String SQL3 = "insert into Marks(sid, marks, year) " +
                        "values (?, ?, ?)";
         jdbcTemplateObject.update( SQL3, sid, marks, year);

         System.out.println("Created Name = " + name + ", Age = " +
age);
         transactionManager.commit(status);
      } catch (DataAccessException e) {
         System.out.println("Error in creating record, rolling back");
         transactionManager.rollback(status);
         throw e;
      }
      return;
   }

   public List<StudentMarks> listStudents() {
      String SQL = "select * from Student, Marks where
Student.id=Marks.sid";

      List <StudentMarks> studentMarks = jdbcTemplateObject.query(SQL,
                                    new StudentMarksMapper());
      return studentMarks;
   }
}
```

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

```
package com.tutorialspoint;
import java.util.List;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.StudentJDBCTemplate;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
```

```
            new ClassPathXmlApplicationContext("Beans.xml");

     StudentJDBCTemplate studentJDBCTemplate =
     (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

     System.out.println("------Records creation-------" );
     studentJDBCTemplate.create("Zara", 11, 99, 2010);
     studentJDBCTemplate.create("Nuha", 20, 97, 2010);
     studentJDBCTemplate.create("Ayan", 25, 100, 2011);

     System.out.println("------Listing all the records-------" );
     List<StudentMarks> studentMarks =
studentJDBCTemplate.listStudents();
     for (StudentMarks record : studentMarks) {
        System.out.print("ID : " + record.getId() );
        System.out.print(", Name : " + record.getName() );
        System.out.print(", Marks : " + record.getMarks());
        System.out.print(", Year : " + record.getYear());
        System.out.println(", Age : " + record.getAge());
     }
   }
}
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

   <!-- Initialization for data source -->
   <bean id="dataSource"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="password"/>
   </bean>

   <!-- Initialization for TransactionManager -->
   <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager
">
      <property name="dataSource"  ref="dataSource" />
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <property name="dataSource"  ref="dataSource" />
      <property name="transactionManager"  ref="transactionManager" />
   </bean>

</beans>
```

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 Name = Zara, Age = 11

Created Name = Nuha, Age = 20

Created Name = Ayan, Age = 25

------Listing all the records--------

ID : 1, Name : Zara, Marks : 99, Year : 2010, Age : 11

ID : 2, Name : Nuha, Marks : 97, Year : 2010, Age : 20

ID : 3, Name : Ayan, Marks : 100, Year : 2011, Age : 25
```

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
);
```

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 other required libraries *mysql-connector-java.jar*, *aopalliance-x.y.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 given 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**:

```java
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;

public interface StudentDAO {
   /**
    * This is the method to be used to initialize
    * database resources ie. connection.
    */
   public void setDataSource(DataSource ds);
   /**
    * This is the method to be used to create
    * a record in the Student and Marks tables.
    */
   public void create(String name, Integer age, Integer marks, Integer
year);
   /**
    * This is the method to be used to list down
    * all the records from the Student and Marks tables.
    */
   public List<StudentMarks> listStudents();
}
```

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

```
package com.tutorialspoint;

public class StudentMarks {
   private Integer age;
   private String name;
   private Integer id;
   private Integer marks;
   private Integer year;
   private Integer sid;

   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
      return age;
   }

   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }

   public void setId(Integer id) {
      this.id = id;
   }
   public Integer getId() {
      return id;
   }
   public void setMarks(Integer marks) {
      this.marks = marks;
   }
   public Integer getMarks() {
      return marks;
   }

   public void setYear(Integer year) {
      this.year = year;
   }
   public Integer getYear() {
      return year;
   }

   public void setSid(Integer sid) {
      this.sid = sid;
   }
   public Integer getSid() {
      return sid;
   }
}
```

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

```
package com.tutorialspoint;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
```

```
public class StudentMarksMapper implements RowMapper<StudentMarks> {
   public StudentMarks mapRow(ResultSet rs, int rowNum) throws
SQLException {

      StudentMarks studentMarks = new StudentMarks();

      studentMarks.setId(rs.getInt("id"));
      studentMarks.setName(rs.getString("name"));
      studentMarks.setAge(rs.getInt("age"));
      studentMarks.setSid(rs.getInt("sid"));
      studentMarks.setMarks(rs.getInt("marks"));
      studentMarks.setYear(rs.getInt("year"));

      return studentMarks;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJDBCTemplate implements StudentDAO{
   private JdbcTemplate jdbcTemplateObject;

   public void setDataSource(DataSource dataSource) {
      this.jdbcTemplateObject = new JdbcTemplate(dataSource);
   }

   public void create(String name, Integer age, Integer marks, Integer
year){

      try {
         String SQL1 = "insert into Student (name, age) values (?, ?)";
         jdbcTemplateObject.update( SQL1, name, age);

         // Get the latest student id to be used in Marks table
         String SQL2 = "select max(id) from Student";
         int sid = jdbcTemplateObject.queryForInt( SQL2 );

         String SQL3 = "insert into Marks(sid, marks, year) " +
                       "values (?, ?, ?)";
         jdbcTemplateObject.update( SQL3, sid, marks, year);

         System.out.println("Created Name = " + name + ", Age = " +
age);
         // to simulate the exception.
         throw new RuntimeException("simulate Error condition") ;
      } catch (DataAccessException e) {
         System.out.println("Error in creating record, rolling back");
         throw e;
      }
   }
```

```
   public List<StudentMarks> listStudents() {
      String SQL = "select * from Student, Marks where
Student.id=Marks.sid";

      List <StudentMarks> studentMarks=jdbcTemplateObject.query(SQL,
      new StudentMarksMapper());
      return studentMarks;
   }
}
```

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

```
package com.tutorialspoint;
import java.util.List;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
            new ClassPathXmlApplicationContext("Beans.xml");

      StudentDAO studentJDBCTemplate =
      (StudentDAO)context.getBean("studentJDBCTemplate");

      System.out.println("------Records creation--------" );
      studentJDBCTemplate.create("Zara", 11, 99, 2010);
      studentJDBCTemplate.create("Nuha", 20, 97, 2010);
      studentJDBCTemplate.create("Ayan", 25, 100, 2011);

      System.out.println("------Listing all the records--------" );
      List<StudentMarks> studentMarks =
studentJDBCTemplate.listStudents();
      for (StudentMarks record : studentMarks) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
         System.out.print(", Marks : " + record.getMarks());
         System.out.print(", Year : " + record.getYear());
         System.out.println(", Age : " + record.getAge());
      }
   }
}
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

   <!-- Initialization for data source -->
```

```xml
   <bean id="dataSource"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource">
       <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
       <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
       <property name="username" value="root"/>
       <property name="password" value="cohondob"/>
   </bean>

   <tx:advice id="txAdvice"  transaction-manager="transactionManager">
       <tx:attributes>
       <tx:method name="create"/>
       </tx:attributes>
   </tx:advice>

   <aop:config>
    <aop:pointcut id="createOperation"
    expression="execution(*
com.tutorialspoint.StudentJDBCTemplate.create(..))"/>
       <aop:advisor advice-ref="txAdvice" pointcut-
ref="createOperation"/>
   </aop:config>

   <!-- Initialization for TransactionManager -->
   <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager
">
       <property name="dataSource"  ref="dataSource" />
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
   class="com.tutorialspoint.StudentJDBCTemplate">
       <property name="dataSource"  ref="dataSource" />
   </bean>

</beans>
```

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.

```
ID : 2, Name : Nuha, Age : 2
```

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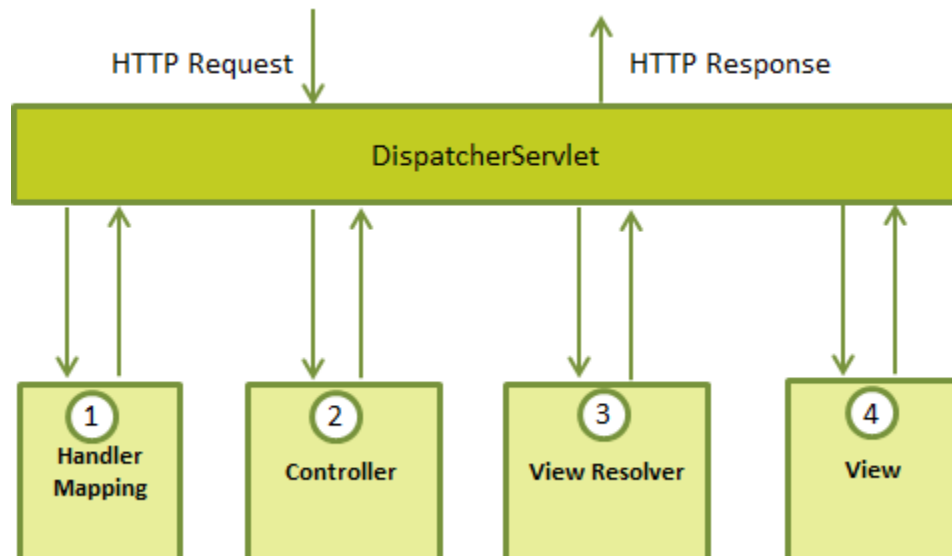
Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet*:

1. After recieving 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:

```xml
<web-app id="WebApp_ID" version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Spring MVC Application</display-name>

   <servlet>
      <servlet-name>HelloWeb</servlet-name>
      <servlet-class>
         org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>HelloWeb</servlet-name>
      <url-pattern>*.jsp</url-pattern>
   </servlet-mapping>

</web-app>
```

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:

```xml
<web-app...>

<!------- DispatcherServlet definition goes here----->
....
<context-param>
   <param-name>contextConfigLocation</param-name>
```

```
    <param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>
</context-param>

<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
</web-app>
```

Now, let us check the required configuration for **HelloWeb-servlet.xml** file, placed in your web application's *WebContent/WEB-INF* directory:

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="com.tutorialspoint" />

    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolve
r">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

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.

```
@Controller
@RequestMapping("/hello")
public class HelloController{
```

```
    @RequestMapping(method = RequestMethod.GET)
    public String printHello(ModelMap model) {
        model.addAttribute("message", "Hello Spring MVC Framework!");
        return "hello";
    }

}
```

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:

```
@Controller
public class HelloController{

    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    public String printHello(ModelMap model) {
        model.addAttribute("message", "Hello Spring MVC Framework!");
        return "hello";
    }

}
```

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:

```
<html>
    <head>
    <title>Hello Spring MVC</title>
    </head>
    <body>
    <h2>${message}</h2>
    </body>
</html>
```

Here **${message}** is the attribute which we have setup inside the Controller. You can have multiple attributes to be displayed inside your view.

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 This example will explain how to write a simple Spring Web Hello World application.
2	Spring MVC Form Handling Example 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.

Spring MVC Hello World Example

The following example show how to write a simple web based Hello World application using Spring MVC framework. To start with it, let us have working Eclipse IDE in place and follow the following steps to develope a Dynamic Web Application using Spring Web Framework:

Step	Description
1	Create a *Dynamic Web Project* with a name *HelloWeb* and create a package*com.tutorialspoint* under the *src* folder in the created project.
2	Drag and drop below mentioned Spring and other libraries into the folder *WebContent/WEB-INF/lib*.
3	Create a Java class *HelloController* under the *com.tutorialspoint* package.
4	Create Spring configuration files *Web.xml* and *HelloWeb-servlet.xml* under the*WebContent/WEB-INF* folder.
5	Create a sub-folder with a name *jsp* under the *WebContent/WEB-INF* folder. Create a view file*hello.jsp* under this sub-folder.
6	The final step is to create the content of all the source and configuration files and export the application as explained below.

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

```java
package com.tutorialspoint;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.ui.ModelMap;

@Controller
@RequestMapping("/hello")
public class HelloController{

    @RequestMapping(method = RequestMethod.GET)
    public String printHello(ModelMap model) {
```

```
        model.addAttribute("message", "Hello Spring MVC Framework!");

        return "hello";
    }

}
```

Following is the content of Spring Web configuration file **web.xml**

```xml
<web-app id="WebApp_ID" version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Spring MVC Application</display-name>

    <servlet>
        <servlet-name>HelloWeb</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWeb</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

Following is the content of another Spring Web configuration file **HelloWeb-servlet.xml**

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-
3.0.xsd">

    <context:component-scan base-package="com.tutorialspoint" />

    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolve
r">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

Following is the content of Spring view file **hello.jsp**

```jsp
<%@ page contentType="text/html; charset=UTF-8" %>
```

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
    <h2>${message}</h2>
</body>
</html>
```

Finally, following is the list of Spring and other libraries to be included in your web application. You simply drag these files and drop them in **WebContent/WEB-INF/lib** folder.

- commons-logging-x.y.z.jar

- org.springframework.asm-x.y.z.jar

- org.springframework.beans-x.y.z.jar

- org.springframework.context-x.y.z.jar

- org.springframework.core-x.y.z.jar

- org.springframework.expression-x.y.z.jar

- org.springframework.web.servlet-x.y.z.jar

- org.springframework.web-x.y.z.jar

- spring-web.jar

Once you are done with creating source and configuration files, export your application. Right click on your application and use **Export > WAR File** option and save your **HelloWeb.war** file in Tomcat's*webapps* folder.

Now start your Tomcat server and make sure you are able to access other web pages from webapps folder using a standard browser. Now try to access the URL **http://localhost:8080/HelloWeb/hello** and if everything is fine with your Spring Web Application, you should see the following result:
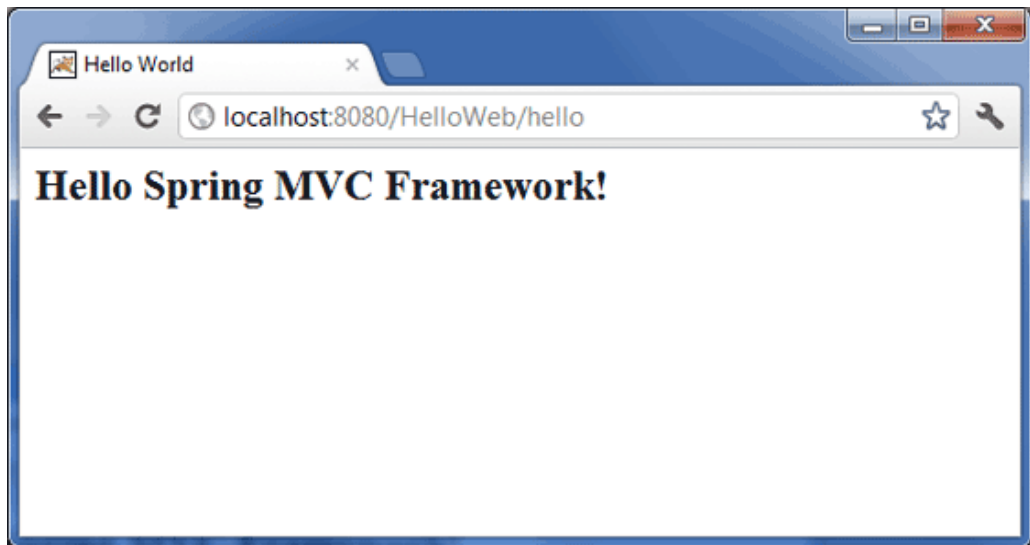
You should note that in the given URL, **HelloWeb** is the application name and **hello** is the virtual subfolder which we have mentioned in our controller using @RequestMapping("/hello"). You can use direct root while mapping your URL using **@RequestMapping("/")**, in this case you can access the same page using short URL **http://localhost:8080/HelloWeb/** but it is advised to have different functionalities under different folders.

Spring MVC Form Handling Example

The following example show how to write a simple web based application which makes use of HTML forms using Spring Web MVC framework. To start with it, let us have working Eclipse IDE in place and follow the following steps to develope a Dynamic Form based Web Application using Spring Web Framework:

Step	Description
1	Create a *Dynamic Web Project* with a name *HelloWeb* and create a package*com.tutorialspoint* under the *src* folder in the created project.
2	Drag and drop below mentioned Spring and other libraries into the folder *WebContent/WEB-INF/lib*.
3	Create a Java classes *Student* and *StudentController* under the *com.tutorialspoint* package.
4	Create Spring configuration files *Web.xml* and *HelloWeb-servlet.xml* under the*WebContent/WEB-INF* folder.
5	Create a sub-folder with a name *jsp* under the *WebContent/WEB-INF* folder. Create a view files *student.jsp* and *result.jsp* under this sub-folder.
6	The final step is to create the content of all the source and configuration files and export the application as explained below.

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

```
package com.tutorialspoint;

public class Student {
    private Integer age;
```

```
    private String name;
    private Integer id;

    public void setAge(Integer age) {
        this.age = age;
    }
    public Integer getAge() {
        return age;
    }

    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }

    public void setId(Integer id) {
        this.id = id;
    }
    public Integer getId() {
        return id;
    }
}
```

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

```
package com.tutorialspoint;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.ModelMap;

@Controller
public class StudentController {

   @RequestMapping(value = "/student", method = RequestMethod.GET)
   public ModelAndView student() {
      return new ModelAndView("student", "command", new Student());
   }

   @RequestMapping(value = "/addStudent", method = RequestMethod.POST)
   public String addStudent(@ModelAttribute("SpringWeb")Student
student,
   ModelMap model) {
      model.addAttribute("name", student.getName());
      model.addAttribute("age", student.getAge());
      model.addAttribute("id", student.getId());

      return "result";
   }
}
```

Here the first service method **student()**, we have passed a blank **Student** object in the
ModelAndView object with name "command" because the spring framework expects an object

with name "command" if you are using <form:form> tags in your JSP file. So when **student()** method is called it returns**student.jsp** view.

Second service method **addStudent()** will be called against a POST method on the**HelloWeb/addStudent** URL. You will prepare your model object based on the submitted information. Finally a "result" view will be returned from the service method, which will result in rendering result.jsp

Following is the content of Spring Web configuration file **web.xml**

```
<web-app id="WebApp_ID" version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Spring MVC Form Handling</display-name>

    <servlet>
        <servlet-name>HelloWeb</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWeb</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

Following is the content of another Spring Web configuration file **HelloWeb-servlet.xml**

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-
3.0.xsd">

    <context:component-scan base-package="com.tutorialspoint" />

    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolve
r">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

Following is the content of Spring view file **student.jsp**

```
<%@taglib uri="http://www.springframework.org/tags/form"
prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Student Information</h2>
<form:form method="POST" action="/HelloWeb/addStudent">
   <table>
    <tr>
        <td><form:label path="name">Name</form:label></td>
        <td><form:input path="name" /></td>
    </tr>
    <tr>
        <td><form:label path="age">Age</form:label></td>
        <td><form:input path="age" /></td>
    </tr>
    <tr>
        <td><form:label path="id">id</form:label></td>
        <td><form:input path="id" /></td>
    </tr>
    <tr>
        <td colspan="2">
            <input type="submit" value="Submit"/>
        </td>
    </tr>
</table>
</form:form>
</body>
</html>
```

Following is the content of Spring view file **result.jsp**

```
<%@taglib uri="http://www.springframework.org/tags/form"
prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Submitted Student Information</h2>
   <table>
    <tr>
        <td>Name</td>
        <td>${name}</td>
    </tr>
    <tr>
        <td>Age</td>
        <td>${age}</td>
    </tr>
    <tr>
        <td>ID</td>
        <td>${id}</td>
    </tr>
</table>
</body>
```
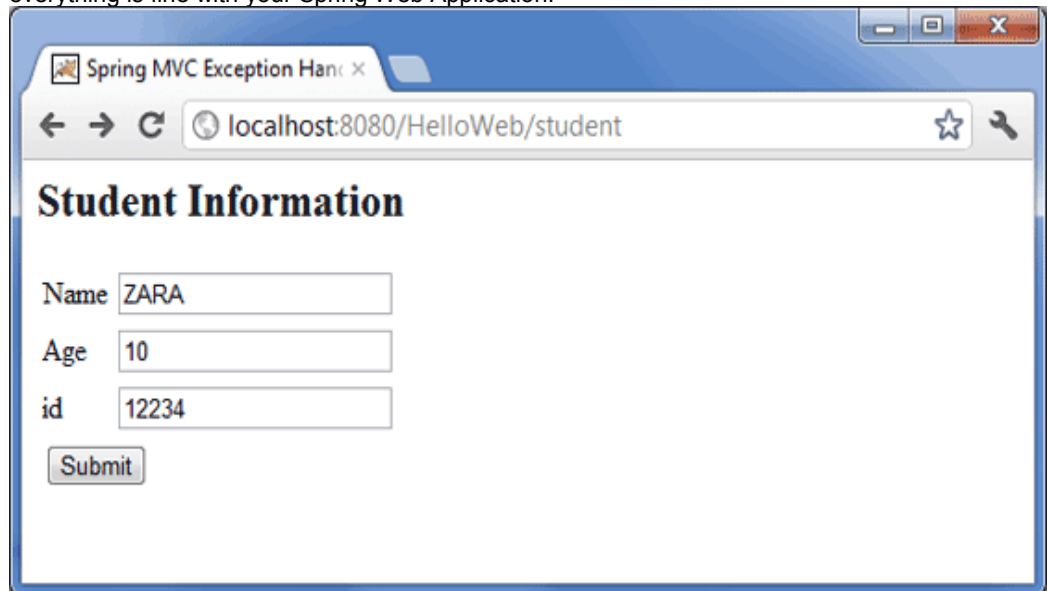
```
</html>
```

Finally, following is the list of Spring and other libraries to be included in your web application. You simply drag these files and drop them in **WebContent/WEB-INF/lib** folder.

- commons-logging-x.y.z.jar

- org.springframework.asm-x.y.z.jar

- org.springframework.beans-x.y.z.jar

- org.springframework.context-x.y.z.jar

- org.springframework.core-x.y.z.jar

- org.springframework.expression-x.y.z.jar

- org.springframework.web.servlet-x.y.z.jar

- org.springframework.web-x.y.z.jar

- spring-web.jar

Once you are done with creating source and configuration files, export your application. Right click on your application and use **Export > WAR File** option and save your **SpringWeb.war** file in Tomcat's *webapps* folder.

Now start your Tomcat server and make sure you are able to access other web pages from webapps folder using a standard browser. Now try a URL **http://localhost:8080/SpringWeb/student** and you should see the following result if everything is fine with your Spring Web Application:

After submitting required information click on submit button to submit the form. You should see the following result if everything is fine with your Spring Web Application:
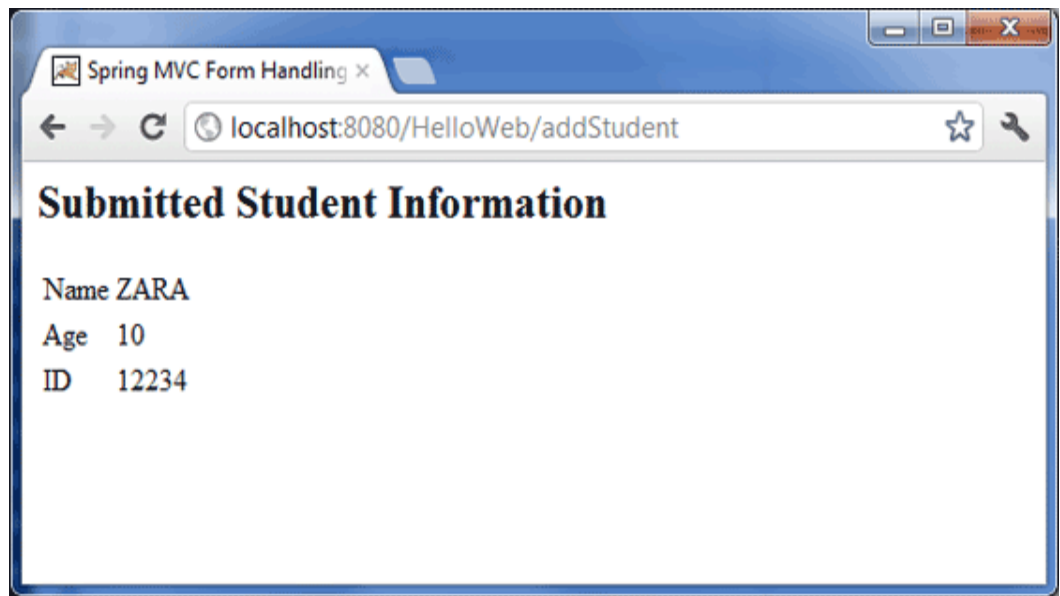
About tutorialspoint.com

Tutorials Point is not a commercial site, this site has been created just for educational purposes and to help the people who are enthusiastic to learn new technologies....

Tutorials Point is aiming to provide the Best Training Materials on highly demanding technical and managerial subjects like:

- Python
- Ruby
- HTML5
- CSS
- JavaScript and related frameworks
- Ruby on Rails
- JAVA and related technologies
- PMP Exams
- Earned Value Management
- Six Sigma
- Parrot
- AJAX
- PHP Programming
- HTML and XHTML
- CGI and Perl
- C Programming
- C++ Programming
- XML-RPC
- SOAP Communication
- HTTP Protocol
- Unix Makefile
- Web Services
- WSDL and UDDI
- Wi-Fi and WiMAX
- Many more...

TutorialsPoint is a FREE site and will remain FREE in future as well.....If you think it is worth to visit this website, kindly share it with your friends and colleagues.

Please send us your feedback at webmaster@tutorialspoint.com

Thanks for visiting us.