**.REFERENCE :- SPRING DURGA SOFT (naveen), JAVATPOINT**

**Spring**

 It was **developed by Rod Johnson in 2003**. Spring framework makes the easy development of JavaEE application.

## Spring Framework

Spring is **a lightweight framework**. It can be thought of as a **framework of frameworks** because it provides support to various frameworks such as [Struts](https://www.javatpoint.com/struts-2-tutorial), [Hibernate](https://www.javatpoint.com/hibernate-tutorial), Tapestry, [EJB](https://www.javatpoint.com/ejb-tutorial), [JSF](https://www.javatpoint.com/jsf-tutorial), etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises several modules such as **IOC, AOP, DAO, Context, ORM, WEB MVC** etc.

It is used for business development purpose.

### Advantages of Spring Framework

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

#### 1) Predefined Templates

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

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

#### 2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

#### 3) Easy to test

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

#### 4) Lightweight

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

#### 5) Fast Development

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

#### 6) Powerful abstraction

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

#### 7) Declarative support

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

### 1.Inversion Of Control (IOC) and Dependency Injection

**Dependency injection = passing required parameters to POJO classes from xml File**

These are the design patterns that are used to remove dependency from the programming code. They make the code easier to test and maintain.

Let's understand this with the following code:

**class Employee**

**{**

**Address address;**

**Employee()**

**{**

**address=new Address();**

**}**

**}**

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

**class Employee**

**{**

**Address address;**

**Employee(Address address)**

**{**

**this.address=address;**

**}**

**}**

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

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

#### Advantage of Dependency Injection

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

**IoC Container**

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

* to instantiate the application class
* to configure the object
* to assemble the dependencies between the objects

There are two types of IoC containers. They are:

1. **BeanFactory (core)**
2. **ApplicationContext (j2ee)**

**By default scope of bean is singleton. Beanfactory create an object at user request while Application Context create object at the time of loading**

### Difference between BeanFactory and the ApplicationContext

The **org.springframework.beans.factory.BeanFactory**

 and the **org.springframework.context.ApplicationContext**

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

### Using BeanFactory

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

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

**BeanFactory factory=new XmlBeanFactory(resource);**

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

#### Using ApplicationContext

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

// at the time of loading the object is created i.e. below statement

**ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");**

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

If the scope in the xml file is not singleton (means it is prototype) than application context will behave as bean factory i.e. object is created at user request.

Example

**package** com.gd. springapp. beans;

**public** **class** Employee {

//Identify Fields /Instance Variables

**private** **long** id;

**private** String name;

**private** **float** salary;

//Default Constructor

**public** Employee () {

**super** ();

}

//Getter and setter methods

**public** **long** getId () {

**return** id;

}

**public** **void** setId (**long** id) {

**this**.id = id;

}

**public** String getName () {

**return** name;

}

**public** **void** setName (String name) {

**this**.name = name;

}

**public** **float** getSalary () {

**return** salary;

}

**public** **void** setSalary (**float** salary) {

**this**. salary = salary;

}

}

**//(xml file)By default singleton**

**/\*schema for using various functionality like collections etc\*/**

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

<beans xmlns=[*http://www.springframework.org/schema/beans*](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.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"* >

<bean id=*"employee"*

class=*"com.gd.springapp.beans.Employee"* >

<!-- <bean name="" class=""> -->

<property name=*"id"* value=*"1"* />

<property name=*"name"* value=*"Sowmya"* />

<property name=*"salary"* value=*"10000.7"* />

</bean>

</beans>

### Using BeanFactory

**package** com.gd.springapp.main;

**import** org.springframework.beans.factory.BeanFactory;

**import** org.springframework.beans.factory.xml.~~XmlBeanFactory~~;

**import** org.springframework.core.io.ClassPathResource;

**import** org.springframework.core.io.Resource;

**import** com.gd.springapp.beans.Employee;

@SuppressWarnings("deprecation")

**public** **class** TestMain1

{

**public** **static** **void** main(String[] args)

{

// Load the bean config

// to configure path

Resource resource = **new** ClassPathResource("beans.xml");

// Get the bean factory

// load xml file to container

BeanFactory factory = **new** ~~XmlBeanFactory~~(resource);

// Get the bean object i.e. for creating object

Employee emp = (Employee)factory.getBean("employee");

**Employee emp1 = (Employee)factory.getBean("employee");**

**//to check whether it is singleton or not**

**System.*out*.println(emp==emp1);**

**/\***

**It return true which means by default scope is singleton**

**\*/**

// Display bean object state

System.***out***.println("Id : " + emp.getId());

System.***out***.println("Name : " + emp.getName());

System.***out***.println("Salary : " + emp.getSalary());

}

}

#### Using ApplicationContext

**package** com.gd.springapp.main;

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

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

**import** com.gd.springapp.beans.Employee;

**public** **class** TestMain2

{

**public** **static** **void** main (String [] args)

{

// Load the bean context & its object is created

ApplicationContext context=**new** ClassPathXmlApplicationContext("beans.xml");

// Get the bean object

Employee emp = (Employee)context.getBean("employee");

**Employee emp1 = (Employee)context.getBean("employee");**

**//to check whether it is singleton or not**

**System.*out*.println(emp==emp1);**

**/\***

**It return true which means by default scope is singleton**

**\*/**

// Display bean object state

System.***out***.println("Id: " + emp. getId ());

System.***out***.println("Name: " + emp. getName ());

System.***out***.println("Salary: " + emp. getSalary ());

}

}

**NOTE:-**

**We can override particular property in same bean**

**<bean id=*"employee"* class=*"com.gd.springapp.beans.Employee"* >**

**<!-- <bean name="" class=""> -->**

**<property name=*"id"* value=*"1"* />**

**<property name=*"id"* value=*"2"* /> //INVALID**

**<property name=*"name"* value=*"Sowmya"* />**

**<property name=*"salary"* value=*"10000.7"* />**

**</bean>**

**</beans>**

**For parameterized constructor it is possible to override**

**1st way:-**

**<bean id=*"employee"* class=*"di.annotation.example\_4.Employee"*>**

**<constructor-arg index=*"0"* value=*"1"*/>**

**<constructor-arg index=*"1"* value=*"Chandra"*/>**

**<constructor-arg index=*"2"* value=*"7500"*/>**

**<constructor-arg index=*"2"* value=*"8500"*/> //VALID**

**</bean>**

**2nd way:-**

**<bean id=*"employee"* class=*"di.annotation.example\_4.Employee"*>**

**<constructor-arg value=*"1" type=”int”*/>**

**<constructor-arg value=*"Chandra" type=”String”*/>**

**</bean>**

EXAMPLE 2

If there is HAS-A relationship between the classes, we create the instance of dependent object (contained object) first then pass it as an argument of the main class constructor. Here, our scenario is Employee HAS-A Address. The Address class object will be termed as the dependent object.

**HERE WE ARE IMPLEMENTING HAS A RELATIONSHIP USING XML FILE**

**package** di.annotation.example\_3;

**public** **class Address** {

// Instance variables

**private** **int** doorNo;

**private** String streetName;

**private** String city;

// Getter and setter methods

**public** **int** getDoorNo() {

**return** doorNo;

}

**public** **void** setDoorNo(**int** doorNo) {

**this**.doorNo = doorNo;

}

**public** String getStreetName() {

**return** streetName;

}

**public** **void** setStreetName(String streetName) {

**this**.streetName = streetName;

}

**public** String getCity() {

**return** city;

}

**public** **void** setCity(String city) {

**this**.city = city;

}

@Override

**public** String toString() {

**return** "Address [doorNo=" + doorNo + ", streetName=" + streetName

+ ", city=" + city + "]";

}

}

package di.annotation.example\_3;

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

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

public **class Employee**

{

// Instance variables

private int employeeId;

private String employeeName;

private double salary;

//Use autowired and understand the difference

@Qualifier(value="empHome")

private Address homeAddress;

//Use autowired and understand the difference

@Qualifier(value="empOffice")

private Address officeAddress;

// Parameterized Constructor

public Employee(int employeeId, String employeeName, double salary)

{

this.employeeId = employeeId;

this.employeeName = employeeName;

this.salary = salary;

}

// Getter and setter methods

public int getEmployeeId ()

{

return employeeId;

}

public void setEmployeeId (int employeeId) {

this.employeeId = employeeId;

}

public String getEmployeeName() {

return employeeName;

}

public void setEmployeeName(String employeeName) {

this.employeeName = employeeName;

}

public double getSalary () {

return salary;

}

public void setSalary (double salary) {

this.salary = salary;

}

@Override

public String toString() {

return "Employee [employeeId=" + employeeId + ", employeeName="

+ employeeName + ", salary=" + salary + "] \n"

+ "Home Address : " + getHomeAddress() + "\n"

+ "Office Address : " + getOfficeAddress() + "\n";

}

public Address getHomeAddress() {

return homeAddress;

}

public void setHomeAddress(Address homeAddress) {

this.homeAddress = homeAddress;

}

public Address getOfficeAddress() {

return officeAddress;

}

public void setOfficeAddress(Address officeAddress) {

this.officeAddress = officeAddress;

}

}

package di.annotation.example\_3;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestMain {

public static void main(String[] args) {

// Get bean object

ApplicationContext context = new ClassPathXmlApplicationContext("spring\_3.xml");

//Get the Employee

Employee emp= (Employee)context.getBean("employee");

//Display Employee

System.out.println(emp.toString());

}

}

XML FILE

<?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*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.0.xsd"*

xmlns:context=*"http://www.springframework"* >

<bean id=*"employee"* class=*"di.annotation.example\_3.Employee"*>

<constructor-arg index=*"0"* value=*"1"*/>

<constructor-arg index=*"1"* value=*"Chandra"*/>

<constructor-arg index=*"2"* value=*"7500"*/>

**/\* HERE WE IMPLEMENTED HAS A RELATIONSHIP using pass by reference\*/**

**<property name = *"homeAddress"* ref = *"home"* />**

**<property name = *"officeAddress"* ref = *"office"* />**

</bean>

<bean id = *"home"* class = *"di.annotation.example\_3.Address"*>

<!-- <qualifier value="empHome"></qualifier> -->

<property name = *"doorNo"* value = *"55"* />

<property name = *"streetName"* value = *"Vijayanagar"* />

<property name = *"city"* value = *"Bangalore"* />

</bean>

<bean id = *"office"* class = *"di.annotation.example\_3.Address"*>

<property name = *"doorNo"* value = *"184"* />

<property name = *"streetName"* value = *"Whitefield"* />

<property name = *"city"* value = *"Bangalore"* />

</bean>

<bean class=*"org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"*></bean>

</beans>

**OR WE CAN CREATE LIKE THIS ALSO USING** INNER CLASS

<?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*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.0.xsd"*

xmlns:context=*"http://www.springframework"* >

<bean id=*"employee"* class=*"di.annotation.example\_3.Employee"*>

<constructor-arg index=*"0"* value=*"1"*/>

<constructor-arg index=*"1"* value=*"Chandra"*/>

<constructor-arg index=*"2"* value=*"7500"*/>

**/\* HERE WE IMPLEMENTED HAS A RELATIONSHIP using pass by object\*/**

**<property name = *"homeAddress"* >**

< bean class = *"di.annotation.example\_3.Address"*>

<!-- <qualifier value="empHome"></qualifier> -->

<property name = *"doorNo"* value = *"55"* />

<property name = *"streetName"* value = *"Vijayanagar"* />

<property name = *"city"* value = *"Bangalore"* />

</bean>

**</property>**

**<property name = *"officeAddress"* >**

<bean class = *"di.annotation.example\_3.Address"*>

<property name = *"doorNo"* value = *"184"* />

<property name = *"streetName"* value = *"Whitefield"* />

<property name = *"city"* value = *"Bangalore"* />

</bean>

**</property>**

</bean>

EXAMPLE FOR ARRAYS

public **class Employee**

{

private int employeeId [];

**public** **void** setEmployeeId(**int**[] employeeId) {

**this**.employeeId = employeeId;

}

**public** **int []** getEmployeeId() {

**return** employeeId ;

}

}

<?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*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.0.xsd"*

xmlns:context=*"http://www.springframework"* >

<bean id=*"employee"* class=*"di.annotation.example\_3.Employee"*>

<property name =”employeeId”>

<list>

<value>1</value>

<value>2</value>

<value>3</value>

</list>

</property>

</bean>

</beans>

**EXAMPLE FOR COLLECTIONS**

1. **package** com.javatpoint;
3. **import** java.util.Iterator;
4. **import** java.util.List;
6. **public** **class** Question {
7. **private** **int** id;
8. **private** String name;
9. **private** List<Answer> answers;
11. //setters and getters
13. **public** **void** displayInfo(){
14. System.out.println(id+" "+name);
15. System.out.println("answers are:");
16. Iterator<Answer> itr=answers.iterator();
17. **while**(itr.hasNext()){
18. System.out.println(itr.next());
19. }
20. }
22. }

**Answer.java**

1. **package** com.javatpoint;
3. **public** **class** Answer {
4. **private** **int** id;
5. **private** String name;
6. **private** String by;
8. //setters and getters
10. **public** String toString(){
11. **return** id+" "+name+" "+by;
12. }
13. }

**applicationContext.xml**

The **ref** element is used to define the reference of another bean. Here, we are using **bean** attribute of **ref** element to specify the reference of another bean.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="answer1" **class**="com.javatpoint.Answer">
10. <property name="id" value="1"></property>
11. <property name="name" value="Java is a programming language"></property>
12. <property name="by" value="Ravi Malik"></property>
13. </bean>
14. <bean id="answer2" **class**="com.javatpoint.Answer">
15. <property name="id" value="2"></property>
16. <property name="name" value="Java is a platform"></property>
17. <property name="by" value="Sachin"></property>
18. </bean>
20. <bean id="q" **class**="com.javatpoint.Question">
21. <property name="id" value="1"></property>
22. <property name="name" value="What is Java?"></property>
23. **<property name="answers">**
24. **<list>**
25. **<ref bean="answer1"/>**
26. **<ref bean="answer2"/>**
27. **</list>**
28. </property>
29. </bean>
31. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the displayInfo method.

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** Test {
9. **public** **static** **void** main(String[] args) {
10. Resource r=**new** ClassPathResource("applicationContext.xml");
11. BeanFactory factory=**new** XmlBeanFactory(r);
13. Question q=(Question)factory.getBean("q");
14. q.displayInfo();
16. }  }

**WITH MAP**

**Question.java**

This class contains three properties, getters & setters and displayInfo() method to display the information.

1. **package** com.javatpoint;
2. **import** java.util.Iterator;
3. **import** java.util.Map;
4. **import** java.util.Set;
5. **import** java.util.Map.Entry;
7. **public** **class** Question {
8. **private** **int** id;
9. **private** String name;
10. **private** Map<Answer,User> answers;
12. //getters and setters

15. **public** **void** displayInfo(){
16. System.out.println("question id:"+id);
17. System.out.println("question name:"+name);
18. System.out.println("Answers....");
19. Set<Entry<Answer, User>> set=answers.entrySet();
20. Iterator<Entry<Answer, User>> itr=set.iterator();
21. **while**(itr.hasNext()){
22. Entry<Answer, User> entry=itr.next();
23. Answer ans=entry.getKey();
24. User user=entry.getValue();
25. System.out.println("Answer Information:");
26. System.out.println(ans);
27. System.out.println("Posted By:");
28. System.out.println(user);
29. }
30. }
31. }

**Answer.java**

1. **package** com.javatpoint;
3. **import** java.util.Date;
5. **public** **class** Answer {
6. **private** **int** id;
7. **private** String answer;
8. **private** Date postedDate;
9. **public** Answer() {}
10. **public** Answer(**int** id, String answer, Date postedDate) {
11. **super**();
12. **this**.id = id;
13. **this**.answer = answer;
14. **this**.postedDate = postedDate;
15. }
17. **public** String toString(){
18. **return** "Id:"+id+" Answer:"+answer+" Posted Date:"+postedDate;
19. }
20. }

**User.java**

1. **package** com.javatpoint;
3. **public** **class** User {
4. **private** **int** id;
5. **private** String name,email;
6. **public** User() {}
7. **public** User(**int** id, String name, String email) {
8. **super**();
9. **this**.id = id;
10. **this**.name = name;
11. **this**.email = email;
12. }
14. **public** String toString(){
15. **return** "Id:"+id+" Name:"+name+" Email Id:"+email;
16. }
17. }

**applicationContext.xml**

The **key-ref** and **value-ref** attributes of entry **element** is used to define the reference of bean in the map.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="answer1" **class**="com.javatpoint.Answer">
10. <property name="id" value="1"></property>
11. <property name="answer" value="Java is a Programming Language"></property>
12. <property name="postedDate" value="12/12/2001"></property>
13. </bean>
14. <bean id="answer2" **class**="com.javatpoint.Answer">
15. <property name="id" value="2"></property>
16. <property name="answer" value="Java is a Platform"></property>
17. <property name="postedDate" value="12/12/2003"></property>
18. </bean>
20. <bean id="user1" **class**="com.javatpoint.User">
21. <property name="id" value="1"></property>
22. <property name="name" value="Arun Kumar"></property>
23. <property name="email" value="arun@gmail.com"></property>
24. </bean>
25. <bean id="user2" **class**="com.javatpoint.User">
26. <property name="id" value="2"></property>
27. <property name="name" value="Varun Kumar"></property>
28. <property name="email" value="Varun@gmail.com"></property>
29. </bean>
31. <bean id="q" **class**="com.javatpoint.Question">
32. <property name="id" value="1"></property>
33. <property name="name" value="What is Java?"></property>
34. **<property name="answers">**
35. **<map>**
36. **<entry key-ref="answer1" value-ref="user1"></entry>**
37. **<entry key-ref="answer2" value-ref="user2"></entry>**
38. **</map>**
39. </property>
40. </bean>
42. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the displayInfo() method to display the information.

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** Test {
9. **public** **static** **void** main(String[] args) {
10. Resource r=**new** ClassPathResource("applicationContext.xml");
11. BeanFactory factory=**new** XmlBeanFactory(r);
13. Question q=(Question)factory.getBean("q");
14. q.displayInfo();
16. }
17. }

**FOR SET**

<property name = "addressSet">

<set>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</set>

</property>

### 2.2 Spring Beans Configuration

**See the different code snippets to declare the required type collection in bean configuration file.**

#### 2.2.1 Vector type

To inject the values for Vector, we have <util:list list-class=*"java.util.Vector"*/> child tag of <property/> element.

<property name="empName">

<util:list list-class="java.util.Vector">

<value>Atul</value>

<value>Sandeep</value>

</util:list>

</property>

#### 2.2.2 TreeSet type

For TreeSet we can use the <util:set set-class=*"java.util.TreeSet"*/>. It will also not accept the duplicate values.

<property name="empId">

<util:set set-class="java.util.TreeSet">

<value>WSO1101</value>

<value>WSO1201</value>

</util:set>

</property>

#### 2.2.3 Hashtable type

And for Hashtable we have <util:map map-class=*"java.util.Hashtable"*/> tag and its child tag <entry/> to pass the values in the pair of key and value. Hashtable will also not accept the duplicate values.

<property name="empIdName">

<util:map map-class="java.util.Hashtable">

<entry key="WSO1" value="Vipul" />

<entry key="WSO2" value="Mukul" />

<entry key="WSO3" value="Ankita" />

</util:map>

</property>

# Spring Dependency Checking

## ****Spring Dependency Checking modes****

1. **none** – No dependency checking will be performed; any property can be left as blank.
2. **simple** – If any properties of primitive type **(int, long, …)** and collection types **(map,list..) then compulsory(mandatory) we have to call primitive setter method otherwise error we get** , **UnsatisfiedDependencyException**  will be thrown.
3. **objects** – If any properties of object type **(other than the simple types) then compulsory(mandatory) we have to call secondary setter method** **otherwise error we get**, **UnsatisfiedDependencyException** will be thrown.
4. **all** –then **then compulsory(mandatory) we have to call primitive & secondary both setter method** If any properties of any type have not been set, an **UnsatisfiedDependencyException**will be thrown.

**Note :** The default mode is **none**. Spring dependency checking feature can only check if the properties has been **set**or**not**, but it cannot check if value set is**null**or**not.**

package com.javainterviewpoint;

public **class Employee**

{

public String name;

public int age;

public Employee()

{

super();

}

public Employee(String name, int age)

{

super();

this.name = name;

this.age = age;

}

public String getName ()

{

return name;

}

public void setName (String name)

{

this.name = name;

}

public int getAge ()

{

return age;

}

public void setAge (int age)

{

this.age = age;

}

@Override

public String toString ()

{

return "Employee [name=" + name + ", age=" + age + "]";

}

}

### ****Declaring Bean Configuration File (SpringConfig.xml)****

<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-2.5.xsd">

<bean id="emp" class="com.javainterviewpoint.Employee**" dependency-check="simple**">

<property name="age" value="100"></property>

</bean>

</beans>

package com.javainterviewpoint;

public **class Employee**

{

private PermanentEmployee pe;

int age;

public Employee()

{

super();

}

public int getAge()

{

return age;

}

public void setAge(int age)

{

this.age = age;

}

public Employee(PermanentEmployee pe)

{

super();

this.pe = pe;

}

public PermanentEmployee getPe()

{

return pe;

}

public void setPe(PermanentEmployee pe)

{

this.pe = pe;

}

}

### ****SpringConfig.xml****

In our configuration file we are not injecting reference to the property **“pe”**

 <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-2.5.xsd">

<bean id="emp" class="com.javainterviewpoint.Employee" **dependency-check="objects">**

<property name=”pe” ref=”si”>

</bean>

</beans>

**DEPENDS ON**

The depends-on attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized.

### Spring depends-on attribute example

Suppose there is a class **ClassA** with it’s own fields, methods and a [static block](https://www.netjstech.com/2016/02/static-block-in-java.html).

There is another class, **ClassB** with a [constructor](https://www.netjstech.com/2015/04/constructor-in-java.html), but it doesn’t refer to ClassA in anyway. Now you want to make sure that the **static block** should have executed before the initialization of **ClassB** bean, which means ClassA should be initialized before ClassB.

public class ClassA {

public ClassA(){

System.out.println("Initializing ClassA");

}

// static blank final variable

static final int i;

static int b;

//static block

static {

System.out.println("in static block");

i = 5;

b = i \* 5;

System.out.println("Values " + i + " " + b);

}

public static int getB() {

return b;

}

public static void setB(int b) {

ClassA.b = b;

}

public static int getI() {

return i;

}

}

public class ClassB {

public ClassB(){

System.out.println("Initializing ClassB");

}

}

If you define the beans in your XML configuration with out using depends-on attribute there is **no guarantee** that ClassA will be initialized first.

<?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"

xmlns:context="http://www.springframework.org/schema/context"

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

http://www.springframework.org/schema/beans/spring-beans-4.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<bean id="classB" class="org.netjs.prog.ClassB">

</bean>

<bean id="classA" class="org.netjs.prog.ClassA">

</bean>

</beans>

You can run it using this **test class**-

import org.netjs.prog.ClassB;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

public static void main(String[] args) {

ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext

("appcontext.xml");

ClassB cbObj = (ClassB)context.getBean("classB");

//cbObj.displayValue();

context.close();

}

}

**Output**

Initializing ClassB

in static block

Values 5 25

Initializing ClassA

Here you can see ClassB object is initialized first and then ClassA object is initialized and its static block is executed.  
But that is not what you want so you can use **depends-on** attribute to tell the spring framework that ClassA object should be ready before initializing ClassB object.

In that case your **XML configuration** would change -

<?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"

xmlns:context="http://www.springframework.org/schema/context"

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

http://www.springframework.org/schema/beans/spring-beans-4.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<bean id="classB" class="org.netjs.prog.ClassB" **depends-on="classA**">

</bean>

<bean id="classA" class="org.netjs.prog.ClassA">

</bean>

</beans>

Now running it using the **test class** yields-

in static block

Values 5 25

Initializing ClassA

Initializing ClassB

Now you can see that the ClassA is initialized first.

# Spring p-namespace

In the Spring framework, p-namespace is used to inject setter-based dependency. The p-namespace is XML shortcut and reduce the numbers of line in the configuration file. However, the p-namespace is not defined in an XSD file and exists only in the core of Spring.

Student.java

package beans;

public class Student {

// Setters and Getters....

private String name;

private String course;

}

Teacher.java

package beans;

public class Teacher {

// Generate setters....

private String name;

private String qualification;

private Student student;

public void setName(String name) {

this.name = name;

}

public void setQualification(String qualification) {

this.qualification = qualification;

}

public void setStudent(Student student) {

this.student = student;

}

// business logic

public void display() {

System.out.println("Teacher and Student details...........\n");

System.out.println("Teacher name: " + name);

System.out.println("Teacher qualification: " + qualification);

System.out.println("Student name: " + student.getName());

System.out.println("Student course: " + student.getCourse());

}

}

### Spring Beans Configuration

Add p-namespace schema into root tag beans.

spring.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:p="<http://www.springframework.org/schema/p>"

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

<http://www.springframework.org/schema/beans/spring-beans.xsd>">

<!-- setter-based dependency injection using p-namespace -->

<bean id="newTeacher" class="beans.Teacher" **p:name="Vijay Pandey" p:qualification="PhD" p:student-ref="newStudent" />**

<bean id="newStudent" class="beans.Student**" p:name="Atul" p:course="B.Sc"** />

</beans>

### Run it

Load the configuration file and run it.

Principal.xml

package clients;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import beans.Teacher;

public class Principal {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");

Teacher tchr = (Teacher) context.getBean("newTeacher");

tchr.display();

}

}

**Output:**

You will get the following result in your console log.

Teacher and Student details...........

Teacher name: Vijay Pandey

Teacher qualification: PhD

Student name: Atul

Student course: B.Sc

**Spring c-namespace**

Spring c-namespace is an XML shortcut and replacement of the <constructor-arg/> subelement of the <bean/> tag. To enable the c-namespace feature, we need to add the xmlns:c="http://www.springframework.org/schema/c" into the XML file. Note that this namespace does not have a separate XSD file; therefore, IDEs such as IntelliJ do not recognize it.

package com.zetcode.bean;

public class User {

private String name;

private String occupation;

public User (String name, String occupation) {

this.name = name;

this.occupation = occupation;

}

@Override

public String toString() {

final var sb = new StringBuilder("User {");

sb.append("name='").append(name).append('\'');

sb.append(", occupation='").append(occupation).append('\'');

sb.append('}');

return sb.toString();

}

}

<?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:c="http://www.springframework.org/schema/c"

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

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

<bean name="user1" class="com.zetcode.bean.User">

<constructor-arg name="name" value="John Doe"/>

<constructor-arg name="occupation" value="gardener"/>

</bean>

**<bean name="user2" class="com.zetcode.bean.User"**

**c:name="Peter Smith" c:occupation="teacher"/>**

</beans>

# Autowiring in Spring

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

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

## Autowiring Modes

|  |  |  |
| --- | --- | --- |
| **No.** | **Mode** | **Description** |
| 1) | no | It is the default autowiring mode. It means no autowiring bydefault. |
| 2) | byName | The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method. |
| 3) | byType | The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method. |
| 4) | constructor | The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters. |
| 5) | autodetect | It is deprecated since Spring 3. (For both setter & constructor) |

## 1) byName autowiring mode

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

**It internally uses setter injection.**

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

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

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

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

**EXAMPLE**

**B.java**

This class contains a constructor and method only.

1. **package** org.sssit;
2. **public** **class** B
3. {
4. B()
5. { System.out.println("b is created"); }
6. **void** print()
7. { System.out.println("hello b"); }
8. }

**A.java**

This class contains reference of B class and constructor and method.

1. **package** org.sssit;
2. **public** **class** A
3. {
4. **B b;**
5. A()
6. { System.out.println("a is created"); }
7. **public** B getB()
8. {
9. **return** b;
10. }
11. **public** **void** setB(B b) {
12. **this**.b = b;
13. }
14. **void** print()
15. {System.out.println("hello a");}
16. **void** display(){
17. print();
18. b.print();
19. }
20. }

**applicationContext.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. **<bean id="b" class="org.sssit.B"></bean>**
10. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>
12. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the display method.

1. **package** org.sssit;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **public** **class** Test {
5. **public** **static** **void** main(String[] args) {
6. ApplicationContext context=**new** ClassPathXmlApplicationContext("applicationContext.xml");
7. A a=context.getBean("a",A.**class**);
8. a.display();
9. }
10. }

Output:

b is created

a is created

hello a

hello b

## 2) byType autowiring mode

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

It internally uses setter injection.

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

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

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

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

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

In such case, it will throw exception.

## 3) constructor autowiring mode

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

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

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

## 4) no autowiring mode

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

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

**@Autowired  annotation** – We can use Spring @Autowired annotation for spring bean autowiring. @Autowired annotation can be applied on variables and methods for autowiring byType. We can also use @Autowired annotation on constructor for constructor based spring autowiring.

For @Autowired annotation to work, we also need to enable annotation based configuration in spring bean configuration file. This can be done by **context:annotation-config** element or by defining a bean of type org.springframework.beans.factory.annotation.AutowiredAnnotation

<bean class=*"org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"*></bean>

**@Qualifier  annotation** – This annotation is used to avoid conflicts in bean mapping and we need to provide the bean name that will be used for autowiring. This way we can avoid issues where multiple beans are defined for same type. This annotation usually works with the @Autowired annotation. For constructors with multiple arguments, we can use this annotation with the argument names in the method.

**package** di.annotation.example\_3;

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

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

**public** **class** Employee {

// Instance variables

**private** **int** employeeId;

**private** String employeeName;

**private** **double** salary;

//Use autowired and understand the difference

**@Autowired**

**/\* (using this we do not need setter or constructor to create object i.e it automatically inject object )\*/**

**@Qualifier(value="empHome")**

**/\*it is used to remove ambiguity problem & its value should be same as bean id of that class\*/**

**private** Address homeAddress;

//Use autowired and understand the difference

**@Autowired**

**@Qualifier(value="empOffice")**

**private** Address officeAddress;

// Parameterized Constructor

**public** Employee(**int** employeeId, String employeeName, **double** salary) {

**this**.employeeId = employeeId;

**this**.employeeName = employeeName;

**this**.salary = salary;

}

// Getter and setter methods

**public** **int** getEmployeeId() {

**return** employeeId;

}

**public** **void** setEmployeeId(**int** employeeId) {

**this**.employeeId = employeeId;

}

**public** String getEmployeeName() {

**return** employeeName;

}

**public** **void** setEmployeeName(String employeeName) {

**this**.employeeName = employeeName;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

@Override

**public** String toString() {

**return** "Employee [employeeId=" + employeeId + ", employeeName="

+ employeeName + ", salary=" + salary + "] \n"

+ "Home Address : " + getHomeAddress() + "\n"

+ "Office Address : " + getOfficeAddress() + "\n";

}

**public** Address getHomeAddress() {

**return** homeAddress;

}

// public void setHomeAddress(Address homeAddress) {

// this.homeAddress = homeAddress;

// }

**public** Address getOfficeAddress() {

**return** officeAddress;

}

// public void setOfficeAddress(Address officeAddress) {

// this.officeAddress = officeAddress;

// }

}

**package** di.annotation.example\_3;

**public** **class** Address {

// Instance variables

**private** **int** doorNo;

**private** String streetName;

**private** String city;

// Getter and setter methods

**public** **int** getDoorNo() {

**return** doorNo;

}

**public** **void** setDoorNo(**int** doorNo) {

**this**.doorNo = doorNo;

}

**public** String getStreetName() {

**return** streetName;

}

**public** **void** setStreetName(String streetName) {

**this**.streetName = streetName;

}

**public** String getCity() {

**return** city;

}

**public** **void** setCity(String city) {

**this**.city = city;

}

@Override

**public** String toString() {

**return** "Address [doorNo=" + doorNo + ", streetName=" + streetName

+ ", city=" + city + "]";

}

}

package di.annotation.example\_3;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestMain {

public static void main(String[] args)

{

ApplicationContext context = new ClassPathXmlApplicationContext("spring\_3.xml");

Employee emp= (Employee)context.getBean("employee");

System.out.println(emp.toString());

}

}

<?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*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.0.xsd"*

xmlns:context=*"http://www.springframework"* >

<bean id=*"employee"* class=*"di.annotation.example\_3.Employee"*>

<constructor-arg index=*"0"* value=*"1"*/>

<constructor-arg index=*"1"* value=*"Chandra"*/>

<constructor-arg index=*"2"* value=*"7500"*/>

</bean>

<bean id = *"homeAddress"* class = *"di.annotation.example\_3.Address"*>

<!-- <qualifier value="empHome"></qualifier> -->

<property name = *"doorNo"* value = *"55"* />

<property name = *"streetName"* value = *"Vijayanagar"* />

<property name = *"city"* value = *"Bangalore"* />

</bean>

<bean id = *"officeAddress"* class = *"di.annotation.example\_3.Address"*>

<property name = *"doorNo"* value = *"184"* />

<property name = *"streetName"* value = *"Whitefield"* />

<property name = *"city"* value = *"Bangalore"* />

</bean>

<bean class=*"org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"*></bean>

</beans>

# Dependency Injection with Factory Method in Spring

# (Spring IOC Part 26- 27 by Naveen)

Spring framework provides facility to inject bean using factory method. To do so, we can use two attributes of bean element.

**factory-method:** represents the factory method that will be invoked object to inject the bean.

**factory-bean:** represents the reference of the bean by which factory method will be invoked. It is used if factory method is non-static.

A method that returns instance of a class is called **factory method**.

1. **public** **class** A {
2. **public** **static** A getA(){//factory method
3. **return** **new** A();
4. }
5. }

## Factory Method Types

There can be three types of factory method:

1) A **static factory method** that returns instance of **its own** class. It is used in singleton design pattern.

<bean id="a" **class**="com.javatpoint.A" factory-method="getA"></bean>

2) A **static factory method** that returns instance of **another** class. It is used instance is not known and decided at runtime.

<bean id="b" **class**="com.javatpoint.A" factory-method="getB"></bean>

3) A **non-static factory** method that returns instance of **another** class. It is used instance is not known and decided at runtime.

<bean id="a" **class**="com.javatpoint.A"></bean>

<bean id="b" **class**="com.javatpoint.A" factory-method="getB" factory-bean="a"></bean>

**Type 1**

**Let's see the simple code to inject the dependency by static factory method.**

<bean id="a" **class**="com.javatpoint.A" factory-method="getA"></bean>

Let's see the full example to inject dependency using factory method in spring. To create this example, we have created 3 files.

1. **A.java**
2. **applicationContext.xml**
3. **Test.java**

**A.java**

This class is a singleton class.

1. **package** com.javatpoint;
2. **public** **class** A {
3. **private** **static** **final** A obj=**new** A();
4. **private** A()
5. {System.out.println("private constructor");}
6. **public** **static** A getA()
7. {
8. System.out.println("factory method ");
9. **return** obj;
10. }
11. **public** **void** msg(){
12. System.out.println("hello user");
13. }
14. }

**applicationContext.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="a" **class**="com.javatpoint.A" factory-method="getA"></bean>
11. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the msg method.

1. **package** org.sssit;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **public** **class** Test {
5. **public** **static** **void** main(String[] args) {
6. ApplicationContext context=**new** ClassPathXmlApplicationContext("applicationContext.xml");
7. A a=(A)context.getBean("a");
8. a.msg();
9. }
10. }

Output:

private constructor

factory method

hello user

## Type 2

Let's see the simple code to inject the dependency by static factory method that returns the instance of another class.

To create this example, we have created 6 files.

1. **Printable.java**
2. **A.java**
3. **B.java**
4. **PrintableFactory.java**
5. **applicationContext.xml**
6. **Test.java**

**Printable.java**

1. **package** com.javatpoint;
2. **public** **interface** Printable {
3. **void** print();
4. }

**A.java**

1. **package** com.javatpoint;
2. **public** **class** A **implements** Printable{
3. @Override
4. **public** **void** print() {
5. System.out.println("hello a");
6. }
8. }

**B.java**

1. **package** com.javatpoint;
2. **public** **class** B **implements** Printable{
3. @Override
4. **public** **void** print() {
5. System.out.println("hello b");
6. }
7. }

**PrintableFactory.java**

1. **package** com.javatpoint;
2. **public** **class** PrintableFactory {
3. **public** **static** Printable getPrintable(){
4. //return new B();
5. **return** **new** A();//return any one instance, either A or B
6. }
7. }

**applicationContext.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="p" **class**="com.javatpoint.PrintableFactory" factory-method="getPrintable"></bean>
11. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the print() method.

1. **package** org.sssit;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **public** **class** Test {
5. **public** **static** **void** main(String[] args) {
6. ApplicationContext context=**new** ClassPathXmlApplicationContext("applicationContext.xml");
7. Printable p=(Printable)context.getBean("p");
8. p.print();
9. }
10. }

Output:

hello a

## Type 3

Let's see the example to inject the dependency by non-static factory method that returns the instance of another class.

To create this example, we have created 6 files.

1. **Printable.java**
2. **A.java**
3. **B.java**
4. **PrintableFactory.java**
5. **applicationContext.xml**
6. **Test.java**

All files are same as previous, you need to change only 2 files: PrintableFactory and applicationContext.xml.

**PrintableFactory.java**

1. **package** com.javatpoint;
2. **public** **class** PrintableFactory {
3. //non-static factory method
4. **public** Printable getPrintable(){
5. **return** **new** A();//return any one instance, either A or B
6. }
7. }

**applicationContext.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. **<bean id="pfactory" class="com.javatpoint.PrintableFactory"></bean>**
10. **<bean id="p" class="com.javatpoint.PrintableFactory" factory-method="getPrintable"   factory-bean="pfactory"></bean>**
11. </beans>

Output:

hello a

**Spring – Bean Life Cycle**

Spring IOC Part 28 – 30 by Naveen

## 1. Bean life cycle

When container starts – a Spring bean needs to be instantiated, based on Java or XML bean definition. It may also be required to perform some post-initialization steps to get it into a usable state. Same bean life cycle is for [*spring boot*](https://howtodoinjava.com/spring-boot-tutorials/) applications as well.

After that, when the bean is no longer required, it will be removed from the IoC container.

Spring bean factory is responsible for managing the life cycle of beans created through spring container.

#### 1.1. Life cycle callbacks

Spring bean factory controls the creation and destruction of beans. To execute some custom code, it provides the call back methods which can be categorized broadly in two groups:

* **Post-initialization** call back methods
* **Pre-destruction** call back methods

## 2. Life cycle callback methods

Spring framework provides following **4 ways for controlling life cycle events** of a bean:

1. **InitializingBean and DisposableBean callback interfaces**
2. **\*Aware interfaces for specific behavior**
3. **Custom init() and destroy() methods in bean configuration file**
4. **@PostConstruct and @PreDestroy annotations**

**Spring InitializingBean and DisposableBean example**

In Spring, **InitializingBean** and **DisposableBean** are two marker interfaces, a useful way for Spring to perform certain actions upon bean initialization and destruction.

1. For bean implemented InitializingBean, it will run afterPropertiesSet() after all bean properties have been set.

Let us take example if we want to load mysql driver we will establish connection in Initializaing bean . so when we call Application context the connection is establish automatically once for the entire life cycle instead of creating it again & again & it will destroy when the container is relaesed .

1. For bean implemented DisposableBean, it will run destroy() after Spring container is released the bean.

package com.mkyong.customer.services;

import org.springframework.beans.factory.DisposableBean;

import org.springframework.beans.factory.InitializingBean;

public class CustomerService implements InitializingBean, DisposableBean

{

String message;

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message;

}

public void afterPropertiesSet() throws Exception {

System.out.println("Init method after properties are set : " + message);

}

public void destroy() throws Exception {

System.out.println("Spring Container is destroy! Customer clean up");

}

}

File : Spring-Customer.xml

<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-2.5.xsd">

<bean id="customerService" class="com.mkyong.customer.services.CustomerService">

<property name="message" value="i'm property message" />

</bean>

</beans>

Run it

package com.mkyong.common;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.mkyong.customer.services.CustomerService;

public class App

{

public static void main( String[] args )

{

ConfigurableApplicationContext context =

new ClassPathXmlApplicationContext(new String[] {"Spring-Customer.xml"});

CustomerService cust = (CustomerService)context.getBean("customerService");

System.out.println(cust);

context.close();

}

}

The **ConfigurableApplicationContext.close()** will close the application context, releasing all resources and destroying all cached singleton beans. It’s use for destroy() method demo purpose only 🙂

Output

Init method after properties are set : im property message

com.mkyong.customer.services.CustomerService@47393f

...

INFO: Destroying singletons in org.springframework.beans.factory.

support.DefaultListableBeanFactory@77158a:

defining beans [customerService]; root of factory hierarchy

Spring Container is destroy! Customer clean up

The afterPropertiesSet() method is called, after the message property is set; while the destroy() method is call after the context.close();

# Spring AOP Tutorial

**Aspect Oriented Programming** (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class.

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns**.

A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

#### Why use AOP?

It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic. Suppose there are 10 methods in a class as given below:

**class** A

{

**public** **void** m1() {...}

**public** **void** m2() {...}

**public** **void** m3() {...}

**public** **void** m4() {...}

**public** **void** m5() {...}

**public** **void** n1() {...}

**public** **void** n2() {...}

**public** **void** p1() {...}

**public** **void** p2() {...}

**public** **void** p3() {...}

}

There are 5 methods that starts from m, 2 methods that starts from n and 3 methods that starts from p.

**Understanding Scenario,** I have to maintain log and send notification after calling methods that starts from m.

**Problem without AOP** We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.

But, if client says in future, I don't have to send notification, you need to change all the methods. It leads to the maintenance problem.

**Solution with AOP** We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class. Its entry is given in the xml file.

In future, if client says to remove the notifier functionality, we need to change only in the xml file. So, maintenance is easy in AOP.

#### Where use AOP?

AOP is mostly used in following cases:

* to provide declarative enterprise services such as declarative transaction management.
* It allows users to implement custom aspects.

## AOP Concepts and Terminology

AOP concepts and terminologies are as follows:

* Join point
* Advice
* Pointcut
* Introduction
* Target Object
* Aspect
* Interceptor
* AOP Proxy
* Weaving

#### Join point

Join point is any point in your program such as method execution, exception handling, field access etc. Spring supports only method execution join point.

#### Advice

Advice represents an action taken by an aspect at a particular join point. There are different types of advices:

* **Before Advice**: it executes before a join point.
* **After Returning Advice**: it executes after a joint point completes normally.
* **After Throwing Advice**: it executes if method exits by throwing an exception.
* **After (finally) Advice**: it executes after a join point regardless of join point exit whether normally or exceptional return.
* **Around Advice**: It executes before and after a join point.

#### Pointcut

It is an expression language of AOP that matches join points.

#### Introduction

It means introduction of additional method and fields for a type. It allows you to introduce new interface to any advised object.

#### Target Object

It is the object i.e. being advised by one or more aspects. It is also known as proxied object in spring because Spring AOP is implemented using runtime proxies.

#### Aspect

It is a class that contains advices, joinpoints etc.

#### Interceptor

It is an aspect that contains only one advice.

#### AOP Proxy

It is used to implement aspect contracts, created by AOP framework. It will be a JDK dynamic proxy or CGLIB proxy in spring framework.

#### Weaving

It is the process of linking aspect with other application types or objects to create an advised object. Weaving can be done at compile time, load time or runtime. Spring AOP performs weaving at runtime.

### AOP Implementations

AOP implementations are provided by:

1. AspectJ
2. Spring AOP
3. JBoss AOP

### Spring AOP

Spring AOP can be used by 3 ways given below. But the widely used approach is Spring AspectJ Annotation Style. The 3 ways to use spring AOP are given below:

1. [By Spring1.2 Old style (dtd based) (also supported in Spring3)](https://www.javatpoint.com/spring-aop-example)
2. [By AspectJ annotation-style](https://www.javatpoint.com/spring-aop-aspectj-annotation-example)
3. [By Spring XML configuration-style(schema based)](https://www.javatpoint.com/spring-aop-aspectj-xml-configuration-example)

# Spring AOP Example

There are given examples of **Spring1.2 old style AOP** (dtd based) implementation.

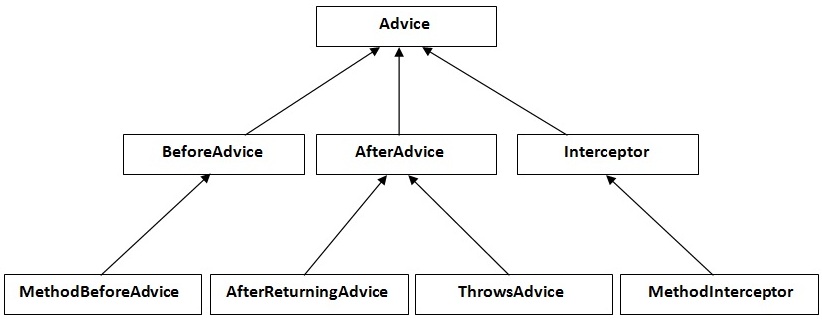
Though it is supported in spring 3, but it is recommended to use spring aop with aspectJ that we are going to learn in next page.

There are 4 types of advices supported in spring1.2 old style aop implementation.

1. **Before Advice** it is executed before the actual method call.
2. **After Advice** it is executed after the actual method call. If method returns a value, it is executed after returning value.
3. **Around Advice** it is executed before and after the actual method call.
4. **Throws Advice** it is executed if actual method throws exception.

#### Understanding the hierarchy of advice interfaces

Let's understand the advice hierarchy by the diagram given below:



All are interfaces in aop.

**MethodBeforeAdvice** interface extends the **BeforeAdvice** interface.

**AfterReturningAdvice** interface extends the **AfterAdvice** interface.

**ThrowsAdvice** interface extends the **AfterAdvice** interface.

**MethodInterceptor** interface extends the **Interceptor** interface. It is used in around advice.

#### 1) MethodBeforeAdvice Example

Create a class that contains actual business logic.

*File: A.java*

**package** com.javatpoint;

**public** **class** A

 {

**public** **void** m(){System.out.println("actual business logic");}

}

Now, create the advisor class that implements MethodBeforeAdvice interface.

*File: BeforeAdvisor.java*

**package** com.javatpoint;

**import** java.lang.reflect.Method;

**import** org.springframework.aop.MethodBeforeAdvice;

**public** **class** BeforeAdvisor **implements** MethodBeforeAdvice

{

    @Override

**public** **void** before (Method method, Object [] args, Object target ) **throws** Throwable

{

         System.out.println("additional concern before actual logic");

     }

}

In xml file, create 3 beans, one for A class, second for Advisor class and third for **ProxyFactoryBean** class.

*File: applicationContext.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:p="http://www.springframework.org/schema/p"

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

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

<bean id="obj" **class**="com.javatpoint.A"></bean>

<bean id="ba" **class**="com.javatpoint.BeforeAdvisor"></bean>

<bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target" ref="obj"></property>

<property name="interceptorNames">

<list>

<value>ba</value>

</list>

</property>

</bean>

</beans>

**Understanding ProxyFactoryBean class:**

The **ProxyFactoryBean** class is provided by Spring Famework. It contains 2 properties target and interceptorNames. The instance of A class will be considered as target object and the instance of advisor class as interceptor. You need to pass the advisor object as the list object as in the xml file given above.

The ProxyFactoryBean class is written something like this:

**public** **class** ProxyFactoryBean

{

**private** Object target;

**private** List interceptorNames;

//getters and setters

}

Now, let's call the actual method.

*File: Test.java*

**package** com.javatpoint;

**import** org.springframework.beans.factory.BeanFactory;

**import** org.springframework.beans.factory.xml.XmlBeanFactory;

**import** org.springframework.core.io.ClassPathResource;

**import** org.springframework.core.io.Resource;

**public** **class** Test

 {

**public** **static** **void** main(String[] args)

{

    Resource r=**new** ClassPathResource("applicationContext.xml");

      BeanFactory factory=**new** XmlBeanFactory(r);

            A a=factory.getBean("proxy",A.**class**);

     a.m();

}

}

#### Output

additional concern before actual logic actual business logic

#### Printing additional information in MethodBeforeAdvice

We can print additional information like method name, method argument, target object, target object class name, proxy class etc.

You need to change only two classes BeforeAdvisor.java and Test.java.

*File: BeforeAdvisor.java*

**package** com.javatpoint;

**import** java.lang.reflect.Method;

**import** org.springframework.aop.MethodBeforeAdvice;

**public** **class** BeforeAdvisor **implements** MethodBeforeAdvice

{

    @Override

**public** **void** before(Method method, Object[] args, Object target)**throws** Throwable

 {

         System.out.println("additional concern before actual logic");

    System.out.println("method info:"+method.getName()+" "+method.getModifiers());

         System.out.println("argument info:");

**for**(Object arg:args)

             System.out.println(arg);

         System.out.println("target Object:"+target);

         System.out.println("target object class name: "+target.getClass().getName());

    }

}

*File: Test.java*

**package** com.javatpoint;

**import** org.springframework.beans.factory.BeanFactory;

**import** org.springframework.beans.factory.xml.XmlBeanFactory;

**import** org.springframework.core.io.ClassPathResource;

**import** org.springframework.core.io.Resource;

**public** **class** Test

 {

**public** **static** **void** main(String[] args)

{

     Resource r=**new** ClassPathResource("applicationContext.xml");

     BeanFactory factory=**new** XmlBeanFactory(r);

       A a=factory.getBean("proxy",A.**class**);

         System.out.println("proxy class name: "+a.getClass().getName());

     a.m();

}

}

#### Output

proxy **class** name: com.javatpoint.A$$EnhancerByCGLIB$$409872b1 additional concern before actual logic method info:m 1 argument info: target Object:com.javatpoint.A@11dba45  target object **class** name: com.javatpoint.A  actual business logic

#### 2) AfterReturningAdvice Example

Create a class that contains actual business logic.

*File: A.java*

Same as in the previous example.

Now, create the advisor class that implements AfterReturningAdvice interface.

*File: AfterAdvisor.java*

**package** com.javatpoint;

**import** java.lang.reflect.Method;

**import** org.springframework.aop.AfterReturningAdvice;

**public** **class** AfterAdvisor **implements** AfterReturningAdvice

{

    @Override

**public** **void** afterReturning(Object returnValue, Method method,

         Object[] args, Object target) **throws** Throwable

  {

          System.out.println("additional concern after returning advice");

     }

}

Create the xml file as in the previous example, you need to change only the advisor class here.

*File: applicationContext.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:p="http://www.springframework.org/schema/p"

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

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

<bean id="obj" **class**="com.javatpoint.A"></bean>

<bean id="ba" **class**="com.javatpoint.AfterAdvisor"></bean>

<bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target" ref="obj"></property>

<property name="interceptorNames">

<list>

<value>ba</value>

</list>

</property>

</bean>

</beans>

*File: Test.java*

Same as in the previous example.

#### Output

actual business logic

additional concern after returning advice

#### 3) MethodInterceptor (AroundAdvice) Example

Create a class that contains actual business logic.

*File: A.java*

Same as in the previous example.

Now, create the advisor class that implements MethodInterceptor interface.

*File: AroundAdvisor.java*

**package** com.javatpoint;

**import** org.aopalliance.intercept.MethodInterceptor;

**import** org.aopalliance.intercept.MethodInvocation;

**public** **class** AroundAdvisor **implements** MethodInterceptor{

    @Override

**public** Object invoke(MethodInvocation mi) **throws** Throwable {

        Object obj;

        System.out.println("additional concern before actual logic");

        obj=mi.proceed();

        System.out.println("additional concern after actual logic");

**return** obj;

    }

}

Create the xml file as in the previous example, you need to change only the advisor class here.

*File: applicationContext.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:p="http://www.springframework.org/schema/p"

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

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

<bean id="obj" **class**="com.javatpoint.A"></bean>

<bean id="ba" **class**="com.javatpoint.AroundAdvisor"></bean>

<bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target" ref="obj"></property>

<property name="interceptorNames">

<list>

<value>ba</value>

</list>

</property>

</bean>

</beans>

*File: Test.java*

Same as in the previous example.

#### Output

additional concern before actual logic actual business logic   additional concern after actual logic

#### 4) ThrowsAdvice Example

Create a class that contains actual business logic.

*File: Validator.java*

**package** com.javatpoint;

**public** **class** Validator {

**public** **void** validate(**int** age)**throws** Exception{

**if**(age<18){

**throw** **new** ArithmeticException("Not Valid Age");

        }

**else**{

            System.out.println("vote confirmed");

        }

    }

}

Now, create the advisor class that implements ThrowsAdvice interface.

*File: ThrowsAdvisor.java*

**package** com.javatpoint;

**import** org.springframework.aop.ThrowsAdvice;

**public** **class** ThrowsAdvisor **implements** ThrowsAdvice

{

**Public** **void** afterThrowing(Exception ex)

{

        System.out.println("additional concern if exception occurs");

    }

}

Create the xml file as in the previous example, you need to change only the Validator class and advisor class.

*File: applicationContext.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:p="http://www.springframework.org/schema/p"

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

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

<bean id="obj" **class**="com.javatpoint.Validator"></bean>

<bean id="ba" **class**="com.javatpoint.ThrowsAdvisor"></bean>

<bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target" ref="obj"></property>

<property name="interceptorNames">

<list>

<value>ba</value>

</list>

</property>

</bean>

</beans>

*File: Test.java*

**package** com.javatpoint;

**import** org.springframework.beans.factory.BeanFactory;

**import** org.springframework.beans.factory.xml.XmlBeanFactory;

**import** org.springframework.core.io.ClassPathResource;

**import** org.springframework.core.io.Resource;

**public** **class** Test {

**public** **static** **void** main(String[] args)

 {

    Resource r=**new** ClassPathResource("applicationContext.xml");

    BeanFactory factory=**new** XmlBeanFactory(r);

    Validator v=factory.getBean("proxy",Validator.**class**);

**Try**

{

     v.validate(12);

     }

**Catch** (Exception e)

{

e.printStackTrace();

}

}

}

#### Output

java.lang.ArithmeticException: Not Valid Age

additional concern **if** exception occurs

    at com.javatpoint.Validator.validate(Validator.java:7)

    at com.javatpoint.Validator$$FastClassByCGLIB$$562915cf.invoke(<generated>)

    at net.sf.cglib.proxy.MethodProxy.invoke(MethodProxy.java:191)

    at org.springframework.aop.framework.Cglib2AopProxy$CglibMethodInvocation.invoke

Joinpoint(Cglib2AopProxy.java:692)

    at org.springframework.aop.framework.ReflectiveMethodInvocation.

proceed(ReflectiveMethodInvocation.java:150)

    at org.springframework.aop.framework.adapter.ThrowsAdviceInterceptor.

invoke(ThrowsAdviceInterceptor.java:124)

    at org.springframework.aop.framework.ReflectiveMethodInvocation.

proceed(ReflectiveMethodInvocation.java:172)

    at org.springframework.aop.framework.Cglib2AopProxy$DynamicAdvisedInterceptor.

intercept(Cglib2AopProxy.java:625)

    at com.javatpoint.Validator$$EnhancerByCGLIB$$4230ed28.validate(<generated>)

#### 

# Spring AOP AspectJ Annotation Example

1. [@Before Example](https://www.javatpoint.com/spring-aop-aspectj-annotation-example)
2. [@After Example](https://www.javatpoint.com/spring-aop-aspectj-annotation-example)
3. [@AfterReturning Example](https://www.javatpoint.com/spring-aop-aspectj-annotation-example)
4. [@Around Example](https://www.javatpoint.com/spring-aop-aspectj-annotation-example)
5. [@AfterThrowing Example](https://www.javatpoint.com/spring-aop-aspectj-annotation-example)

The **Spring Framework** recommends you to use **Spring AspectJ AOP implementation** over the Spring 1.2 old style dtd based AOP implementation because it provides you more control and it is easy to use.

There are two ways to use Spring AOP AspectJ implementation:

1. By annotation: We are going to learn it here.
2. By xml configuration (schema based): We will learn it in next page.

Spring AspectJ AOP implementation provides many annotations:

1. **@Aspect** declares the class as aspect.
2. **@Pointcut** declares the pointcut expression.

The annotations used to create advices are given below:

1. **@Before** declares the before advice. It is applied before calling the actual method.
2. **@After** declares the after advice. It is applied after calling the actual method and before returning result.
3. **@AfterReturning** declares the after returning advice. It is applied after calling the actual method and before returning result. But you can get the result value in the advice.
4. **@Around** declares the around advice. It is applied before and after calling the actual method.
5. **@AfterThrowing** declares the throws advice. It is applied if actual method throws exception.

## Understanding Pointcut

Pointcut is an expression language of Spring AOP.

The **@Pointcut** annotation is used to define the pointcut. We can refer the pointcut expression by name also. Let's see the simple example of pointcut expression.

1. @Pointcut("execution(\* Operation.\*(..))")
2. **private** **void** doSomething() {}

The name of the pointcut expression is doSomething(). It will be applied on all the methods of Operation class regardless of return type.

#### Understanding Pointcut Expressions

Let's try the understand the pointcut expressions by the examples given below:

1. @Pointcut("execution(public \* \*(..))")

It will be applied on all the public methods.

1. @Pointcut("execution(public Operation.\*(..))")

It will be applied on all the public methods of Operation class.

1. @Pointcut("execution(\* Operation.\*(..))")

It will be applied on all the methods of Operation class.

1. @Pointcut("execution(public Employee.set\*(..))")

It will be applied on all the public setter methods of Employee class.

1. @Pointcut("execution(int Operation.\*(..))")

It will be applied on all the methods of Operation class that returns int value.

#### 1) @Before Example

The AspectJ Before Advice is applied before the actual business logic method. You can perform any operation here such as conversion, authentication etc.

Create a class that contains actual business logic.

*File: Operation.java*

1. **package** com.javatpoint;
2. **public**  **class** Operation{
3. **public** **void** msg(){System.out.println("msg method invoked");}
4. **public** **int** m(){System.out.println("m method invoked");**return** 2;}
5. **public** **int** k(){System.out.println("k method invoked");**return** 3;}
6. }

Now, create the aspect class that contains before advice.

*File: TrackOperation.java*

1. **package** com.javatpoint;
3. **import** org.aspectj.lang.JoinPoint;
4. **import** org.aspectj.lang.annotation.Aspect;
5. **import** org.aspectj.lang.annotation.Before;
6. **import** org.aspectj.lang.annotation.Pointcut;
8. @Aspect
9. **public** **class** TrackOperation{
10. @Pointcut("execution(\* Operation.\*(..))")
11. **public** **void** k(){}//pointcut name
13. @Before("k()")//applying pointcut on before advice
14. **public** **void** myadvice(JoinPoint jp)//it is advice (before advice)
15. {
16. System.out.println("additional concern");
17. //System.out.println("Method Signature: "  + jp.getSignature());
18. }
19. }

Now create the applicationContext.xml file that defines beans.

*File: applicationContext.xml*

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop.xsd">

11. <bean id="opBean" **class**="com.javatpoint.Operation">   </bean>
12. <bean id="trackMyBean" **class**="com.javatpoint.TrackOperation"></bean>
14. <bean **class**="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator"></bean>
16. </beans>

Now, let's call the actual method.

*File: Test.java*

1. **package** com.javatpoint;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Test{
6. **public** **static** **void** main(String[] args){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");
8. Operation e = (Operation) context.getBean("opBean");
9. System.out.println("calling msg...");
10. e.msg();
11. System.out.println("calling m...");
12. e.m();
13. System.out.println("calling k...");
14. e.k();
15. }
16. }

#### Output

1. calling msg...
2. additional concern
3. msg() method invoked
4. calling m...
5. additional concern
6. m() method invoked
7. calling k...
8. additional concern
9. k() method invoked

As you can see, additional concern is printed before msg(), m() and k() method is invoked.

Now if you change the pointcut expression as given below:

1. @Pointcut("execution(\* Operation.m\*(..))")

Now additional concern will be applied for the methods starting with m in Operation class. Output will be as this:

1. calling msg...
2. additional concern
3. msg() method invoked
4. calling m...
5. additional concern
6. m() method invoked
7. calling k...
8. k() method invoked

Now you can see additional concern is not printed before k() method invoked.

#### 2) @After Example

The AspectJ after advice is applied after calling the actual business logic methods. It can be used to maintain log, security, notification etc.

Here, We are assuming that **Operation.java**, **applicationContext.xml** and **Test.java** files are same as given in @Before example.

Create the aspect class that contains after advice.

*File: TrackOperation.java*

1. **package** com.javatpoint;
3. **import** org.aspectj.lang.JoinPoint;
4. **import** org.aspectj.lang.annotation.Aspect;
5. **import** org.aspectj.lang.annotation.After;
6. **import** org.aspectj.lang.annotation.Pointcut;
8. @Aspect
9. **public** **class** TrackOperation{
10. @Pointcut("execution(\* Operation.\*(..))")
11. **public** **void** k(){}//pointcut name
13. @After("k()")//applying pointcut on after advice
14. **public** **void** myadvice(JoinPoint jp)//it is advice (after advice)
15. {
16. System.out.println("additional concern");
17. //System.out.println("Method Signature: "  + jp.getSignature());
18. }
19. }

#### Output

1. calling msg...
2. msg() method invoked
3. additional concern
4. calling m...
5. m() method invoked
6. additional concern
7. calling k...
8. k() method invoked
9. additional concern

You can see that additional concern is printed after calling msg(), m() and k() methods.

#### 3) @AfterReturning Example

By using after returning advice, we can get the result in the advice.

Create the class that contains business logic.

*File: Operation.java*

1. **package** com.javatpoint;
2. **public**  **class** Operation{
3. **public** **int** m(){System.out.println("m() method invoked");**return** 2;}
4. **public** **int** k(){System.out.println("k() method invoked");**return** 3;}
5. }

Create the aspect class that contains after returning advice.

*File: TrackOperation.java*

1. **package** com.javatpoint;
3. **import** org.aspectj.lang.JoinPoint;
4. **import** org.aspectj.lang.annotation.AfterReturning;
5. **import** org.aspectj.lang.annotation.Aspect;
7. @Aspect
8. **public** **class** TrackOperation{
9. @AfterReturning(
10. pointcut = "execution(\* Operation.\*(..))",
11. returning= "result")
13. **public** **void** myadvice(JoinPoint jp,Object result)//it is advice (after returning advice)
14. {
15. System.out.println("additional concern");
16. System.out.println("Method Signature: "  + jp.getSignature());
17. System.out.println("Result in advice: "+result);
18. System.out.println("end of after returning advice...");
19. }
20. }

*File: applicationContext.xml*

It is same as given in @Before advice example

*File: Test.java*

Now create the Test class that calls the actual methods.

1. **package** com.javatpoint;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Test{
6. **public** **static** **void** main(String[] args){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");
8. Operation e = (Operation) context.getBean("opBean");
9. System.out.println("calling m...");
10. System.out.println(e.m());
11. System.out.println("calling k...");
12. System.out.println(e.k());
13. }
14. }

#### Output

1. calling m...
2. m() method invoked
3. additional concern
4. Method Signature: **int** com.javatpoint.Operation.m()
5. Result in advice: 2
6. end of after returning advice...
7. 2
8. calling k...
9. k() method invoked
10. additional concern
11. Method Signature: **int** com.javatpoint.Operation.k()
12. Result in advice: 3
13. end of after returning advice...
14. 3

You can see that return value is printed two times, one is printed by TrackOperation class and second by Test class.

#### 4) @Around Example

The AspectJ around advice is applied before and after calling the actual business logic methods.

Here, we are assuming that **applicationContext.xml** file is same as given in @Before example.

Create a class that contains actual business logic.

*File: Operation.java*

1. **package** com.javatpoint;
2. **public**  **class** Operation{
3. **public** **void** msg(){System.out.println("msg() is invoked");}
4. **public** **void** display(){System.out.println("display() is invoked");}
5. }

Create the aspect class that contains around advice.

You need to pass the **PreceedingJoinPoint** reference in the advice method, so that we can proceed the request by calling the proceed() method.

*File: TrackOperation.java*

1. **package** com.javatpoint;
2. **import** org.aspectj.lang.ProceedingJoinPoint;
3. **import** org.aspectj.lang.annotation.Around;
4. **import** org.aspectj.lang.annotation.Aspect;
5. **import** org.aspectj.lang.annotation.Pointcut;
7. @Aspect
8. **public** **class** TrackOperation
9. {
10. @Pointcut("execution(\* Operation.\*(..))")
11. **public** **void** abcPointcut(){}
13. @Around("abcPointcut()")
14. **public** Object myadvice(ProceedingJoinPoint pjp) **throws** Throwable
15. {
16. System.out.println("Additional Concern Before calling actual method");
17. Object obj=pjp.proceed();
18. System.out.println("Additional Concern After calling actual method");
19. **return** obj;
20. }
21. }

*File: Test.java*

Now create the Test class that calls the actual methods.

1. **package** com.javatpoint;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **public** **class** Test{
5. **public** **static** **void** main(String[] args){
6. ApplicationContext context = **new** classPathXmlApplicationContext("applicationContext.xml");
8. Operation op = (Operation) context.getBean("opBean");
9. op.msg();
10. op.display();
11. }
12. }

#### Output

1. Additional Concern Before calling actual method
2. msg() is invoked
3. Additional Concern After calling actual method
4. Additional Concern Before calling actual method
5. display() is invoked
6. Additional Concern After calling actual method