**Spring Notes**

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

Inversion Of Control (IOC) and Dependency Injection

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

1. **class** Employee
2. {
3. Address address;
4. Employee()
5. {
6. address=**new** Address();
7. }
8. }

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:

1. **class** Employee{
2. Address address;
3. City city;
4. Employee(Address address){
5. **this**.address=address;
6. }
7. Employee(City c)
8. {
9. City=new City()

}

1. }

//

Employee ee =new Employee(City);

Employee ee =new Employee(Address);

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

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

#### Advantage of Dependency Injection

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

### Advantages of Spring Framework

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

#### 1) Predefined Templates

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

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

#### 2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

#### 3) Easy to test

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

#### 4) Lightweight

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

#### 5) Fast Development

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

#### 6) Powerful abstraction

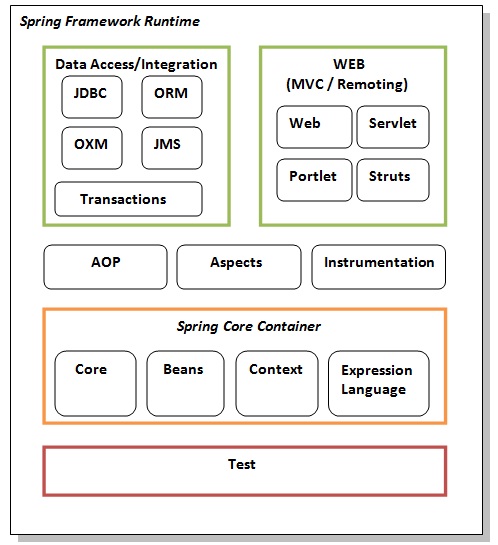
It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

#### 7) Declarative support

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

# Spring Modules

The Spring framework comprises of many modules such as core, beans, context, expression language, AOP, Aspects, Instrumentation, JDBC, ORM, OXM, JMS, Transaction, Web, Servlet, Struts etc. These modules are grouped into Test, Core Container, AOP, Aspects, Instrumentation, Data Access / Integration, Web (MVC / Remoting) as displayed in the following diagram.



## Steps to create spring application

Let's see the 5 steps to create the first spring application.

### 1) Create Java class

This is the simple java bean class containing the name property only.

1. **package** com.techvision;
3. **public** **class** Student
4. {
5. **private** String name;
6. **public** String getName() {
7. **return** name;
8. }
9. **public** **void** setName(String name) {
10. **this**.name = name;
11. }
12. **public** **void** displayInfo(){
13. System.out.println("Hello: "+name);
14. }
15. }

This is simple bean class, containing only one property name with its getters and setters method. This class contains one extra method named displayInfo() that prints the student name by the hello message.

### 2) Create the xml file

In case of myeclipse IDE, you don't need to create the xml file as myeclipse does this for yourselves. Open the applicationContext.xml file, and write the following code:

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="studentbean" **class**="com.techvision.Student">
10. <property name="name" value="Satyam Sinha"></property>
11. </bean>
13. <bean id="student1bean" **class**="com.techvision.Student1"/>
14. <bean id="student3bean" **class**="com.techvision.Student2"/>

</beans>

The **bean** element is used to define the bean for the given class. The **property** subelement of bean specifies the property of the Student class named name. The value specified in the property element will be set in the Student class object by the IOC container.

### 3) Create the test class

Create the java class e.g. Test. Here we are getting the object of Student class from the IOC container using the getBean() method of BeanFactory. Let's see the code of test class.

1. **package** com.techvision;
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. {
10. **public** **static** **void** main(String[] args)
11. {
12. Resource resource=**new** ClassPathResource("applicationContext.xml");
13. BeanFactory factory=**new** XmlBeanFactory(resource);
15. Student student=(Student)factory.getBean("studentbean");
16. student.displayInfo();
17. }
18. }

The **Resource** object represents the information of applicationContext.xml file. The Resource is the interface and the **ClassPathResource** is the implementation class of the Reource interface. The **BeanFactory** is responsible to return the bean. The **XmlBeanFactory** is the implementation class of the BeanFactory. There are many methods in the BeanFactory interface. One method is **getBean()**, which returns the object of the associated class.

### 4) Load the jar files required for spring framework

There are mainly three jar files required to run this application.

* **org.springframework.core-3.0.1.RELEASE-A**
* **com.springsource.org.apache.commons.logging-1.1.1**
* **org.springframework.beans-3.0.1.RELEASE-A**

For the future use, You can download the required jar files for spring core application.

# 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**
2. **ApplicationContext**

Difference between BeanFactory and the ApplicationContext

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

### 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:

1. Resource resource=**new** ClassPathResource("applicationContext.xml");
2. 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:

1. ApplicationContext context =
2. **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.

# Dependency Injection in Spring

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled. To understand the DI better, Let's understand the Dependency Lookup (DL) first:

### Dependency Lookup

The Dependency Lookup is an approach where we get the resource after demand. There can be various ways to get the resource for example:

1. A obj = **new** AImpl();

In such way, we get the resource(instance of A class) directly by new keyword. Another way is factory method:

1. A obj = A.getA();

This way, we get the resource (instance of A class) by calling the static factory method getA().

Alternatively, we can get the resource by JNDI (Java Naming Directory Interface) as:

1. Context ctx = **new** InitialContext();
2. Context environmentCtx = (Context) ctx.lookup("java:comp/env");
3. A obj = (A)environmentCtx.lookup("A");

There can be various ways to get the resource to obtain the resource. Let's see the problem in this approach.

#### Problems of Dependency Lookup

There are mainly two problems of dependency lookup.

* **tight coupling** The dependency lookup approach makes the code tightly coupled. If resource is changed, we need to perform a lot of modification in the code.
* **Not easy for testing** This approach creates a lot of problems while testing the application especially in black box testing.

Dependency Injection

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

1. **class** Employee{
2. Address address;
4. Employee(Address address){
5. **this**.address=address;
6. }
7. **public** **void** setAddress(Address address){
8. **this**.address=address;
9. }
11. }

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

Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

* By Constructor
* By Setter method

# 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:

1. **class** A{
2. **public** **void** m1(){...}
3. **public** **void** m2(){...}
4. **public** **void** m3(){...}
5. **public** **void** m4(){...}
6. **public** **void** m5(){...}
7. **public** **void** n1(){...}
8. **public** **void** n2(){...}
9. **public** **void** p1(){...}
10. **public** **void** p2(){...}
11. **public** **void** p3(){...}
12. }

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 Example

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

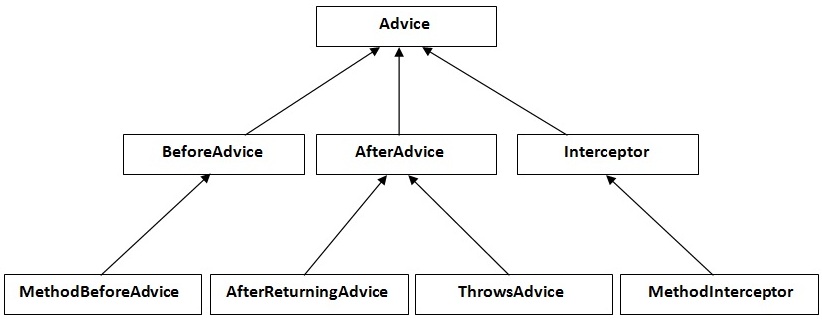
Though it is supported in spring 3, but it is recommended to use spring aop with aspectJ that we are going to learn in next page.

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*

1. **package** com.techvision;
2. **public** **class** A {
3. **public** **void** m(){System.out.println("actual business logic");}
4. }

Now, create the advisor class that implements MethodBeforeAdvice interface.

*File: BeforeAdvisor.java*

1. **package** com.techvision;
2. **import** java.lang.reflect.Method;
3. **import** org.springframework.aop.MethodBeforeAdvice;
4. **public** **class** BeforeAdvisor **implements** MethodBeforeAdvice{
5. @Override
6. **public** **void** before(Method method, Object[] args, Object target)**throws** Throwable {
7. System.out.println("additional concern before actual logic");
8. }
9. }

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

*File: 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="obj" **class**="com.techvision.A"></bean>
10. <bean id="ba" **class**="com.techvision.BeforeAdvisor"></bean>
12. <bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean">
13. <property name="target" ref="obj"></property>
14. <property name="interceptorNames">
15. <list>
16. <value>ba</value>
17. </list>
18. </property>
19. </bean>
21. </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:

1. **public** **class** ProxyFactoryBean{
2. **private** Object target;
3. **private** List interceptorNames;
4. //getters and setters
5. }

Now, let's call the actual method.

*File: Test.java*

1. **package** com.techvision;
2. **import** org.springframework.beans.factory.BeanFactory;
3. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
4. **import** org.springframework.core.io.ClassPathResource;
5. **import** org.springframework.core.io.Resource;
6. **public** **class** Test {
7. **public** **static** **void** main(String[] args) {
8. Resource r=**new** ClassPathResource("applicationContext.xml");
9. BeanFactory factory=**new** XmlBeanFactory(r);
11. A a=factory.getBean("proxy",A.**class**);
12. a.m();
13. }
14. }

#### Output

1. additional concern before actual logic
2. 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*

1. **package** com.techvision;
2. **import** java.lang.reflect.Method;
3. **import** org.springframework.aop.MethodBeforeAdvice;
5. **public** **class** BeforeAdvisor **implements** MethodBeforeAdvice{
6. @Override
7. **public** **void** before(Method method, Object[] args, Object target)**throws** Throwable {
8. System.out.println("additional concern before actual logic");
9. System.out.println("method info:"+method.getName()+" "+method.getModifiers());
10. System.out.println("argument info:");
11. **for**(Object arg:args)
12. System.out.println(arg);
13. System.out.println("target Object:"+target);
14. System.out.println("target object class name: "+target.getClass().getName());
15. }
16. }

*File: Test.java*

1. **package** com.techvision;
2. **import** org.springframework.beans.factory.BeanFactory;
3. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
4. **import** org.springframework.core.io.ClassPathResource;
5. **import** org.springframework.core.io.Resource;
6. **public** **class** Test {
7. **public** **static** **void** main(String[] args) {
8. Resource r=**new** ClassPathResource("applicationContext.xml");
9. BeanFactory factory=**new** XmlBeanFactory(r);
11. A a=factory.getBean("proxy",A.**class**);
12. System.out.println("proxy class name: "+a.getClass().getName());
13. a.m();
14. }
15. }

#### Output

1. proxy **class** name: com.techvision.A$$EnhancerByCGLIB$$409872b1
2. additional concern before actual logic
3. method info:m 1
4. argument info:
5. target Object:com.techvision.A@11dba45
6. target object **class** name: com.techvision.A
7. 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*

1. **package** com.techvision;
2. **import** java.lang.reflect.Method;
3. **import** org.springframework.aop.AfterReturningAdvice;
4. **public** **class** AfterAdvisor **implements** AfterReturningAdvice{
5. @Override
6. **public** **void** afterReturning(Object returnValue, Method method,
7. Object[] args, Object target) **throws** Throwable {
9. System.out.println("additional concern after returning advice");
10. }
12. }

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

*File: 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="obj" **class**="com.techvision.A"></bean>
10. <bean id="ba" **class**="com.techvision.AfterAdvisor"></bean>
12. <bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean">
13. <property name="target" ref="obj"></property>
14. <property name="interceptorNames">
15. <list>
16. <value>ba</value>
17. </list>
18. </property>
19. </bean>
21. </beans>

*File: Test.java*

Same as in the previous example.

#### Output

1. actual business logic
2. 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*

1. **package** com.techvision;
2. **import** org.aopalliance.intercept.MethodInterceptor;
3. **import** org.aopalliance.intercept.MethodInvocation;
4. **public** **class** AroundAdvisor **implements** MethodInterceptor{
6. @Override
7. **public** Object invoke(MethodInvocation mi) **throws** Throwable {
8. Object obj;
9. System.out.println("additional concern before actual logic");
10. obj=mi.proceed();
11. System.out.println("additional concern after actual logic");
12. **return** obj;
13. }
15. }

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

*File: 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="obj" **class**="com.techvision.A"></bean>
10. <bean id="ba" **class**="com.techvision.AroundAdvisor"></bean>
12. <bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean">
13. <property name="target" ref="obj"></property>
14. <property name="interceptorNames">
15. <list>
16. <value>ba</value>
17. </list>
18. </property>
19. </bean>
21. </beans>

*File: Test.java*

Same as in the previous example.

#### Output

1. additional concern before actual logic
2. actual business logic
3. additional concern after actual logic

#### 4) ThrowsAdvice Example

Create a class that contains actual business logic.

*File: Validator.java*

1. **package** com.techvision;
2. **public** **class** Validator {
3. **public** **void** validate(**int** age)**throws** Exception{
4. **if**(age<18){
5. **throw** **new** ArithmeticException("Not Valid Age");
6. }
7. **else**{
8. System.out.println("vote confirmed");
9. }
10. }
11. }

Now, create the advisor class that implements ThrowsAdvice interface.

*File: ThrowsAdvisor.java*

1. **package** com.techvision;
2. **import** org.springframework.aop.ThrowsAdvice;
3. **public** **class** ThrowsAdvisor **implements** ThrowsAdvice{
4. **public** **void** afterThrowing(Exception ex){
5. System.out.println("additional concern if exception occurs");
6. }
7. }

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

*File: 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="obj" **class**="com.techvision.Validator"></bean>
10. <bean id="ba" **class**="com.techvision.ThrowsAdvisor"></bean>
12. <bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean">
13. <property name="target" ref="obj"></property>
14. <property name="interceptorNames">
15. <list>
16. <value>ba</value>
17. </list>
18. </property>
19. </bean>
21. </beans>

*File: Test.java*

1. **package** com.techvision;
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. Validator v=factory.getBean("proxy",Validator.**class**);
14. **try**{
15. v.validate(12);
16. }**catch**(Exception e){e.printStackTrace();}
17. }
18. }

# 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.techvision;
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.techvision;
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.techvision.Operation">   </bean>
12. <bean id="trackMyBean" **class**="com.techvision.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.techvision;
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.techvision;
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.techvision;
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.techvision;
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.techvision;
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.techvision.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.techvision.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.techvision;
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.techvision;
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.techvision;
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

You can see that additional concern is printed before and after calling msg() and display methods.

#### 5) @AfterThrowing Example

By using after throwing advice, we can print the exception in the TrackOperation class. Let's see the example of AspectJ AfterThrowing advice.

Create the class that contains business logic.

*File: Operation.java*

1. **package** com.techvision;
2. **public**  **class** Operation{
3. **public** **void** validate(**int** age)**throws** Exception{
4. **if**(age<18){
5. **throw** **new** ArithmeticException("Not valid age");
6. }
7. **else**{
8. System.out.println("Thanks for vote");
9. }
10. }
12. }

Create the aspect class that contains after throwing advice.

Here, we need to pass the Throwable reference also, so that we can intercept the exception here.

*File: TrackOperation.java*

1. **package** com.techvision;
2. **import** org.aspectj.lang.JoinPoint;
3. **import** org.aspectj.lang.annotation.AfterThrowing;
4. **import** org.aspectj.lang.annotation.Aspect;
5. @Aspect
6. **public** **class** TrackOperation{
7. @AfterThrowing(
8. pointcut = "execution(\* Operation.\*(..))",
9. throwing= "error")
11. **public** **void** myadvice(JoinPoint jp,Throwable error)//it is advice
12. {
13. System.out.println("additional concern");
14. System.out.println("Method Signature: "  + jp.getSignature());
15. System.out.println("Exception is: "+error);
16. System.out.println("end of after throwing advice...");
17. }
18. }

*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.techvision;
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 op = (Operation) context.getBean("opBean");
9. System.out.println("calling validate...");
10. **try**{
11. op.validate(19);
12. }**catch**(Exception e){System.out.println(e);}
13. System.out.println("calling validate again...");
15. **try**{
16. op.validate(11);
17. }**catch**(Exception e){System.out.println(e);}
18. }
19. }

#### Output

1. calling validate...
2. Thanks **for** vote
3. calling validate again...
4. additional concern
5. Method Signature: **void** com.techvision.Operation.validate(**int**)
6. Exception is: java.lang.ArithmeticException: Not valid age
7. end of after throwing advice...
8. java.lang.ArithmeticException: Not valid age

# Spring AOP AspectJ Xml Configuration Example

Spring enables you to define the aspects, advices and pointcuts in xml file.

In the previous page, we have seen the aop examples using annotations. Now we are going to see same examples by the xml configuration file.

Let's see the xml elements that are used to define advice.

1. **aop:before** It is applied before calling the actual business logic method.
2. **aop:after** It is applied after calling the actual business logic method.
3. **aop:after-returning** it is applied after calling the actual business logic method. It can be used to intercept the return value in advice.
4. **aop:around** It is applied before and after calling the actual business logic method.
5. **aop:after-throwing** It is applied if actual business logic method throws exception.

#### 1) aop: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.techvision;
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.techvision;
2. **import** org.aspectj.lang.JoinPoint;
3. **public** **class** TrackOperation{
4. **public** **void** myadvice(JoinPoint jp)//it is advice
5. {
6. System.out.println("additional concern");
7. //System.out.println("Method Signature: "  + jp.getSignature());
8. }
9. }

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-3.0.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
10. <aop:aspectj-autoproxy />
12. <bean id="opBean" **class**="com.techvision.Operation">   </bean>
13. <bean id="trackAspect" **class**="com.techvision.TrackOperation"></bean>
15. <aop:config>
16. <aop:aspect id="myaspect" ref="trackAspect" >
17. <!-- @Before -->
18. <aop:pointcut id="pointCutBefore"   expression="execution(\* com.techvision.Operation.\*(..))" />
19. <aop:before method="myadvice" pointcut-ref="pointCutBefore" />
20. </aop:aspect>
21. </aop:config>
23. </beans>

Now, let's call the actual method.

*File: Test.java*

1. **package** com.techvision;
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.

#### 2) aop: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**, **TrackOperation.java** and **Test.java** files are same as given in aop:before example.

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-3.0.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
10. <aop:aspectj-autoproxy />
12. <bean id="opBean" **class**="com.techvision.Operation">   </bean>
13. <bean id="trackAspect" **class**="com.techvision.TrackOperation"></bean>
15. <aop:config>
16. <aop:aspect id="myaspect" ref="trackAspect" >
17. <!-- @After -->
18. <aop:pointcut id="pointCutAfter"    expression="execution(\* com.techvision.Operation.\*(..))" />
19. <aop:after method="myadvice" pointcut-ref="pointCutAfter" />
20. </aop:aspect>
21. </aop:config>
23. </beans>

#### 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) aop:after-returning 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.techvision;
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.techvision;
3. **import** org.aspectj.lang.JoinPoint;
5. **public** **class** TrackOperation{
6. **public** **void** myadvice(JoinPoint jp,Object result)//it is advice (after advice)
7. {
8. System.out.println("additional concern");
9. System.out.println("Method Signature: "  + jp.getSignature());
10. System.out.println("Result in advice: "+result);
11. System.out.println("end of after returning advice...");
12. }
13. }

*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-3.0.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
10. <aop:aspectj-autoproxy />
12. <bean id="opBean" **class**="com.techvision.Operation">   </bean>
14. <bean id="trackAspect" **class**="com.techvision.TrackOperation"></bean>
16. <aop:config>
17. <aop:aspect id="myaspect" ref="trackAspect" >
18. <!-- @AfterReturning -->
19. <aop:pointcut id="pointCutAfterReturning"   expression="execution(\* com.techvision.Operation.\*(..))" />
20. <aop:after-returning method="myadvice" returning="result" pointcut-ref="pointCutAfterReturning" />
21. </aop:aspect>
22. </aop:config>
24. </beans>

*File: Test.java*

Now create the Test class that calls the actual methods.

1. **package** com.techvision;
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.techvision.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.techvision.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) aop:around example

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

Create a class that contains actual business logic.

*File: Operation.java*

1. **package** com.techvision;
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.techvision;
2. **import** org.aspectj.lang.ProceedingJoinPoint;
3. **public** **class** TrackOperation
4. {
5. **public** Object myadvice(ProceedingJoinPoint pjp) **throws** Throwable
6. {
7. System.out.println("Additional Concern Before calling actual method");
8. Object obj=pjp.proceed();
9. System.out.println("Additional Concern After calling actual method");
10. **return** obj;
11. }
12. }

*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-3.0.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
10. <aop:aspectj-autoproxy />
12. <bean id="opBean" **class**="com.techvision.Operation">   </bean>
14. <bean id="trackAspect" **class**="com.techvision.TrackOperation"></bean>
16. <aop:config>
17. <aop:aspect id="myaspect" ref="trackAspect" >
18. <!-- @Around -->
19. <aop:pointcut id="pointCutAround"   expression="execution(\* com.techvision.Operation.\*(..))" />
20. <aop:around method="myadvice" pointcut-ref="pointCutAround" />
21. </aop:aspect>
22. </aop:config>
24. </beans>

*File: Test.java*

Now create the Test class that calls the actual methods.

1. **package** com.techvision;
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

You can see that additional concern is printed before and after calling msg() and display methods.

#### 5) aop:after-throwing example

By using after throwing advice, we can print the exception in the TrackOperation class. Let's see the example of AspectJ AfterThrowing advice.

Create the class that contains business logic.

*File: Operation.java*

1. **package** com.techvision;
2. **public**  **class** Operation{
3. **public** **void** validate(**int** age)**throws** Exception{
4. **if**(age<18){
5. **throw** **new** ArithmeticException("Not valid age");
6. }
7. **else**{
8. System.out.println("Thanks for vote");
9. }
10. }
12. }

Create the aspect class that contains after throwing advice.

Here, we need to pass the Throwable reference also, so that we can intercept the exception here.

*File: TrackOperation.java*

1. **package** com.techvision;
2. **import** org.aspectj.lang.JoinPoint;
3. **public** **class** TrackOperation{
5. **public** **void** myadvice(JoinPoint jp,Throwable error)//it is advice
6. {
7. System.out.println("additional concern");
8. System.out.println("Method Signature: "  + jp.getSignature());
9. System.out.println("Exception is: "+error);
10. System.out.println("end of after throwing advice...");
11. }
12. }

*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-3.0.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
9. <aop:aspectj-autoproxy />
10. <bean id="opBean" **class**="com.techvision.Operation">   </bean>
11. <bean id="trackAspect" **class**="com.techvision.TrackOperation"></bean>
13. <aop:config>
14. <aop:aspect id="myaspect" ref="trackAspect" >
15. <!-- @AfterThrowing -->
16. <aop:pointcut id="pointCutAfterThrowing"    expression="execution(\* com.techvision.Operation.\*(..))" />
17. <aop:after-throwing method="myadvice" throwing="error" pointcut-ref="pointCutAfterThrowing" />
18. </aop:aspect>
19. </aop:config>
21. </beans>

*File: Test.java*

Now create the Test class that calls the actual methods.

1. **package** com.techvision;
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 op = (Operation) context.getBean("opBean");
9. System.out.println("calling validate...");
10. **try**{
11. op.validate(19);
12. }**catch**(Exception e){System.out.println(e);}
13. System.out.println("calling validate again...");
15. **try**{
16. op.validate(11);
17. }**catch**(Exception e){System.out.println(e);}
18. }
19. }

#### Output

1. calling validate...
2. Thanks **for** vote
3. calling validate again...
4. additional concern
5. Method Signature: **void** com.techvision.Operation.validate(**int**)
6. Exception is: java.lang.ArithmeticException: Not valid age
7. end of after throwing advice...
8. java.lang.ArithmeticException: Not valid age