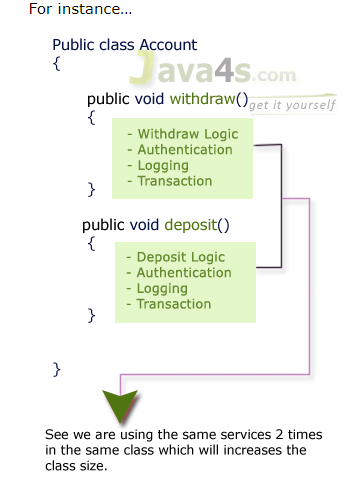
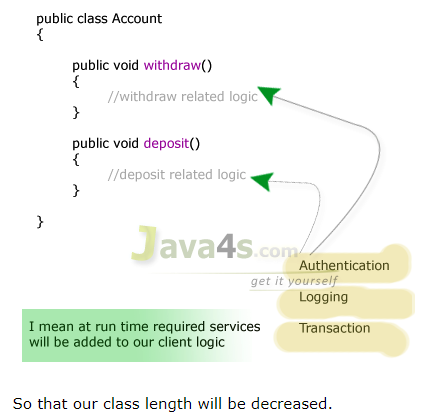
Spring AOP (Aspect Oriented Programming)

Spring AOP (Aspect Oriented Programming), very important module of spring framework.  In the enterprise level application programming, we used to add different cross-cutting functionalities [cross-cutting functionalities means adding different types of services to the application at runtime automatically]



See in the above example, we are using **3** cross-cutting functionalities [ authentication, logging, transaction] in 2 methods.  If we are going to do any modifications in logging service, we need to modify 2 times right? may be 100 times if we wrote in 100 methods which is very difficult.  If I want to remove one service, we need to open class file and delete the required things and need recompile, re-deploy needed.

So, to avoid this, in spring AOP we can add these cross-cutting functionalities at run time :-) I mean we can separate the services [ cross-cutting functionalities] and our client logic.



Finally……………………….

* While implementing business logic for real time applications, apart from business logic some other services also will be added to make that as enterprise level one.
* According to spring, the services that are overlapping on the business logic are called as cross-cutting functionalities, we already saw the drawbacks of implementing business logic + services.
* In order to overcome the above problems, we need to separate the business logic and the services, we call this process of separation as AOP, Using AOP the business logic and cross-cutting functionalities are implemented separately and executed at run time as combine.

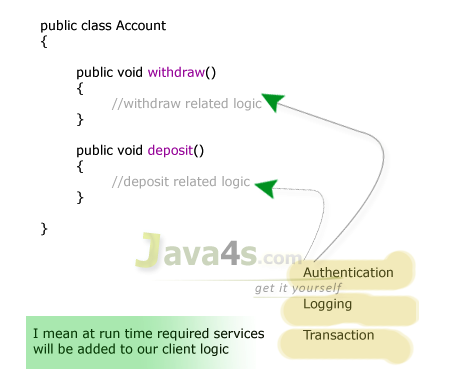
# Spring AOP Terminology, Terms We Should Know Before Entering The AOP

Let us see the terms we should know before moving forward into spring AOP (Aspect Oriented Programming).  Friends these are very important, in fact spring AOP is nothing but knowing these terms in detail, noting in AOP :-)

* Aspect
* Advice
* JoinPoint
* Pointcut
* Introduction
* Target
* Proxy
* Weaving
* Adviser

# Spring Aspect Oriented Programming – Aspect Introduction & Example

* An aspect represents the cross-cutting functionality name, remember just name only.
* One real time service required for a business logic is called one Aspect.
* Aspect denotes only the cross-cutting functionality name not its implementation and all.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | |  |  |  |  |   See here we are adding 3 services to us withdraw() and deposit() methods at run time.  So, what is this?  we have 3 Aspects here which are Authentication Aspect, Logging Aspect, Transaction Aspect May be Mailing Aspect in future.  Hope you understood right? am again saying Aspect means just service name, its implementation is irrelevant as of now.  That’s about Aspect in spring AOP. |

# Spring AOP – Types of Advice

We did see about Aspect already; Advice is the implementation of Aspect.  An Advice provides the code for implementation of the service. As an example, consider logging service, logging is an Aspect and Advice denotes the implementation of Log4j.

## Types of Advices

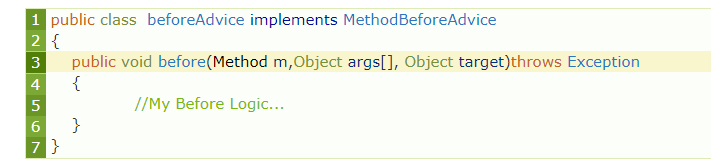
* Before Advice
* After Advice
* Throws Advice
* Around Advice

Let us see one by one with explanation and example.

## Before Advice

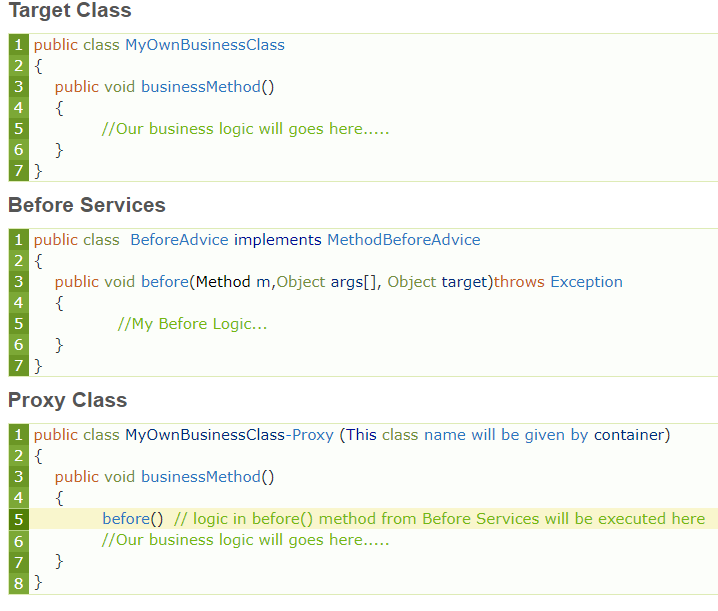
* This advice contains implementation of the services which are need to be applied before business logic of method is going to execute.
* During compilation time the services will not be applied to our logic, services will apply only at run time.
* In order to create a Before advice, class should implement MethodBeforeAdvice interface.
* MethodBeforeAdvice interface is given in org.sp-fw.aop.\* package.
* If we implement MethodBeforeAdvice interface then we need to override a method called before() method.
* The services which are implemented in before() method are executed at before business logic

## Syntax Of  MethodBeforeAdvice  — before() Method



* The first parameter Method m, is a class at java.lang.reflect.\* package, and this parameter is used to access the name of the business method, through getName(), I will show you this in the example.
* Second parameter object[] is used to access the parameter values of the business method, the parameters of business method stored in object array and those parameters are given to before() method by the container in the form of objects only.
* The third parameter is an object to whom this service will be going to apply, usually this will take care by container, as a programmer we no need to care this ;)
* The service implemented in before advice will be called from a proxy class which is generated by the container for a target class.

Confused…? what are these targets and proxies? :-) let me explain



**Explanation**:

* See MyOwnBusinessClass.java is our own java class file, just our business logic will go here
* BeforeAdvice.java contains the services, what we need to execute before our business logic
* So proxy is the class generated by the container at run time, see at run time container is executing the logic of before() method from BeforeAdvice class, then our business method logic as a single class.
* In fact we cannot see this proxy class physically :-) , container will creates in the memory and gives the output normally.

**Example (Create a maven project and put Spring.xml file in resources folder)**

**AOP with *Annotations***

**Pom.xml**

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.8 </version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.8 </version>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjrt</artifactId>

<version>1.9.4</version>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.9.4</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.3</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

**A.java**

**public** **class** A {

**public** **void** m(){

System.***out***.println("actual business logic");

**BeforeAdvisor.java**

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

**public** **void** before(Method arg0, Object[] arg1, Object arg2)

**throws** Throwable {

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

**Spring.xml**

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

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

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

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

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

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

[*http://www.springframework.org/schema/context/spring-context.xsd*](http://www.springframework.org/schema/context/spring-context.xsd)*"*>

<bean id=*"obj"* class=*"p1.A"*></bean>

<bean id=*"ba"* class=*"p1.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>

**App.java**

**public** **class** App {

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

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

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

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

a.m();

**Parameters of before Method**

**BeforeAdvisor.java**

**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());

**Test.java**

     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();

## After Advice (After Returning Advice)

* This is also same as Before Advice, but this advice contains services which are applied after completion of our business method logic
* In order to create an after returning advice in spring, our class should implement an interface called AfterReturningAdvice, given in org.sp-fw.aop.\* package and we need to override a method given by this interface called afterReturning()

**Take above example just change interface to AfterReturningAdvice**

# Spring AOP Throws Advice

* In this type of Advice, we implement services which are executed whenever the business logic of the method throws an exception.  For creating a Throws Advice our class must implement ThrowsAdvice interface.
* ThrowsAdvice is a marker interface given in org.spfw.aop.\*; package, and there are no methods in this interface to provide implementation :-)
* while creating a ThrowsAdvice class in spring AOP, we should implement our services in a method called afterThrowing() with 1 or 4 parameter(s).
* In fact, afterThrowing() method is not given in ThrowsAdvice, but we should implement our services in afterThrowing() method only because whenever an Exception is occurred in the business logic then the IOC container internally calls afterThrowing() method to apply the services.

ThAdvice.java

**import** java.lang.reflect.Method;

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

**public** **class** ThAdvice **implements** ThrowsAdvice{

**public** **void** afterThrowing(Exception e) {

System.***out***.println(e);

}

**public** **void** afterThrowing(Method m,Object[] obj,Objecttarget,ArithmeticException e) {

System.***out***.println(e);

* In case of single parameter, only Exception details are accessible, but in case of 4 parameter method, apart from Exception details we can also access method name, method parameters.  At the time of creating ThrowsAdvice, the method afterThrowing() can write any number of times, i mean we can write individual afterThrowing() methods for each Exception separately.
* If we write multiple afterThrowing() methods in a class then the IOC container will give the preference like follows..
* Verify whether afterThrowing() method is available with specific Exception type as parameter or not, if exist then the container calls that afterThrowing() method and executes services implemented in it.
* If a specific Exception type of afterThrowing() method is not available, then container verifies whether an afterThrowing() method with 4 parameters exist or not if exist then executes the services in it.
* If 4 parameters afterThrowing() is not exist in the class then the container verifies whether an afterThrowing() method with super class Exception parameter (Exception e) exists or not, if exists then executes the services implemented in it
* Actually we know 10/0 will gives arithmetic exception, in MyThrowsAdvice.java i have written afterThrowing(ArithmeticException e) so IOC will executed this method only in the about output.
* While you are executing this application, if you remove afterThrowing(ArithmeticException e) method then IOC will give the preference to afterThrowing(Exception e), and then to afterThrowing(Method m, Object args[], Object target,Exception e).

# Spring AOP Around Advice

* Around Advice is combination of both **Before** and **After** Advice.
* In a single Advice it is possible to implement both Before and After services.
* Around Advice is not given by spring framework and it is from Open-Source implementation called AOP alliance.
* Around Advice can be used by any framework which supports AOP.
* To create Around Advice, our class should implement an interface called MethodInterceptor.
* In Around Advice, we implement Before and After Advice in a single method called invoke(), in order to separate Before an After services to execute business logic, in the middle we call proceed() method.
* Around Advice can access the return value of business method and it can modify the value and it can return a different value back to the client, as return type is Object, but in the After Advice it’s not possible right, as its return type is **void**.

**MyAroundAdvice.java**

**public** **class** MyAroundAdvice **implements** MethodInterceptor{

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

// **TODO** Auto-generated method stub

System.***out***.println("Am before proceed() method");

Object ob = mi.proceed();

System.***out***.println("Am after proceed() method");

**return** ob;

# Spring AOP JoinPoint,

While creating the business logic of the method the additional services are needed to be injected (which we saw already) at different places or points, we call such points as join points.  At a join point a new service will be added into the normal flow of a business method.

While executing the business method, the services are required at the following **3** places (generally), we call them as JoinPoints.

* Before business logic of the method starts
* After business logic of the method got completed
* If business logic throws an exception at run time

**We will see in details in Annotation program later**

# Spring AOP Pointcut

let us describe regarding spring AOP pointcut, for what methods what services need to be executed will be taken care by pointcut. A pointcut defines what advices are required at what join points.  In fact, all business methods of a class don’t require all services.  So, a pointcut informs to the IOC container that what business methods of a class needs what type of services.

**we have 2 types of pointcuts in AOP**

* Static Pointcut
* Dynamic Pointcut

**File 2 Start Here**

**Pom.xml Update**

<properties>

<spring.version>5.1.3.RELEASE</spring.version>

<junit.version>4.12</junit.version>

</properties>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

if you want to use **${spring.version}** then first define its version in **properties** tag and it is good practice to use to define the version in properties tag because if we change the version then we don't need to do changes in the entire file just do change in properties tag.

**Circle.java**

**public** **class** Circle {

**private** String name; //Getter and Setter

@Override

**public** String toString() {

**return** "Circle [name=" + name + "]";

**Triangle.java**

**public** **class** Triangle {

**private** String name; //Getter and Setter

@Override

**public** String toString() {

**return** "Triangle [name=" + name + "]";

**ShapeService.java**

**public** **class** ShapeService {

Circle c;

Triangle t; //Getter and Setter

@Override

**public** String toString() {

**return** "ShapeService [c=" + c + ", t=" + t + "]";

**LoggingAspect.java**

@Aspect

**public** **class** LoggingAspect {

@Before("execution(public String getName())")

**public** **void** loggingAdvice(){

System.***out***.println("Before Executing");

**Spring.xml**

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

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

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

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

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

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

**<aop:aspectj-autoproxy></aop:aspectj-autoproxy>**

<bean id=*"c"* class=*"p1.Circle"*>

<property name=*"name"* value=*"from Circle"*></property>

</bean>

<bean id=*"t"* class=*"p1.Triangle"*>

<property name=*"name"* value=*"from Triangle"*></property>

</bean>

<bean id=*"shape"* class=*"p1.ShapeService"* autowire=*"byType"*></bean>

<bean id=*"myaspect"* class=*"p1.LoggingAspect"*></bean>

</beans>

**App.java**

**public class App {**

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

**ApplicationContext cont = new ClassPathXmlApplicationContext("Spring.xml");**

**ShapeService s = cont.getBean("shape",ShapeService.class);**

**System.out.println(s.getT().getName());**

Update same example with Annotation without Spring.xml file

**AppConfig.java**

@Configuration

@EnableAspectJAutoProxy

@ComponentScan(basePackages = {"p1"})

**public** **class** AppConfig {

@Bean

**public** Circle getCir() {

**return** **new** Circle();

}

@Bean

**public** Triangle getTri() {

**return** **new** Triangle();

}

@Bean

**public** ShapeService getS() {

**return** **new** ShapeService();

}

}

**LoggingAspect.java**

@Aspect

@Component

**public** **class** LoggingAspect {

@Before("execution(public String getName())")

**public** **void** iAmAspect(){

System.***out***.println("I am Aspect");

Remaining all file will be SAME

But on above example we are not defining that loggingAdvice method to run after **which** public String getName() method we have same method in two different class if we want to run this advice to particular class only we can do like

"execution(public String p1.Circle.getName())"

Now it will run for circle only

Just make change in LoggingAspect class and run

But what if I want this Advice should apply to all getter

"execution(public String get\*())"

Now it will apply to all getter which is returing String

But what if I want this Advice should apply to all getter no matter what’s it returning

"execution(public \* get\*())"

Now it will apply to all getter which is returing anything

But what if I want this Advice should apply to all getter which is having arguments

"execution(public \* get\*(\*))"

Now it will apply to all getter which is having some arguments

But what if I want this Advice should apply to all getter which is having arguments or not

"execution(public \* get\*(..))"

Now it will apply to all getter which is having some arguments or not

***@Order* in AOP**

**Now suppose I have 2 Aspect and we want to set a sequence**

**LoggingAspect.java**

@Aspect

@Component

**@Order(1)**

**public** **class** LoggingService {

@Before("execution(public String getName())")

**public** **void** secondAdvice() {

System.***out***.println("secondAdvice");

}

}

MyNewAspect.java

@Aspect

@Component

**@Order(2)**

**public** **class** MyNewAspect {

@Before("execution(public String getName())")

**public** **void** myNewAspect() {

System.***out***.println("myNewAspect");

}

}

**LoggingAspect.java**

@Aspect

**public** **class** LoggingAspect {

@Before("execution(public \* get\*())")

**public** **void** loggingAdvice(){

System.***out***.println("Before Executing");

}

@Before("execution(public \* get\*())")

**public** **void** secondAdvice(){

System.***out***.println("Second Advice Executing");

}

}

I am writing @Before("execution(public \* get\*())") two times instead of doing that what we will do we use something called pointcut annotation

**LoggingAspect.java**

@Aspect

**public** **class** LoggingAspect {

@Before("allGetters()")

**public** **void** loggingAdvice(){

System.***out***.println("Before Executing");

}

@Before("allGetters()")

**public** **void** secondAdvice(){

System.***out***.println("Second Advice Executing");

}

@Pointcut("execution(public \* get\*())")

**public** **void** allGetters(){}

}

Now the output would be like this

Before Executing

Second Advice Executing

Before Executing

Second Advice Executing

from Circle

come two time because I am using two getters in App.java syso line

System.out.println(s.getC().getName());

**Now if I want to run this for all the method of circle class only**

@Pointcut("within(p1.Circle)")

**public** **void** allCircleMethod(){}

**Now if I want to run this for all the class of p1 package**

@Pointcut("within(p1.\*)")

**public** **void** allCircleMethod(){}

**Now if I want to combine two pointCut together**

@Aspect

public class LoggingAspect {

***@Before("allGetters() && allCircleMethod()")***

**public void loggingAdvice(){**

**System.out.println("Before Executing");**

**}**

@Before("allGetters()")

public void secondAdvice(){

System.out.println("Second Advice Executing");

}

**@Pointcut("execution(public \* get\*())")**

public void allGetters(){}

**@Pointcut("within(p1.Circle)")**

public void allCircleMethod(){}

}

**JoinPoint** Start *Here*

We can pass argument in our Advice and that argument will keep the information of method because of which particular advice got run

@Before("allGetters() && allCircleMethod()")

**public** **void** loggingAdvice(JoinPoint j){

System.***out***.println(j.toString());

}

**If we want to get object of targeted class**

@Before("allGetters() && allCircleMethod()")

**public** **void** loggingAdvice(JoinPoint j){

Circle c = (Circle)j.getTarget();

}

**Now if I want to run any advice after that method whose take argument as a String**

@Before("args(String)")

**public** **void** secondAdvice(){

System.***out***.println("Second Advice Executing");

}

Now If I wanted to use that String value which is getting passed from actual method in our advice also, we can do

@Before("args(name)")

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

System.***out***.println("Second Advice Executing " + name);

}

@AfterReturning

It will get executed when method does not throw any exception and executed successfully

@AfterReturning("args(name)")

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

System.***out***.println("Second Advice Executing " + name);

}

@AfterThrowing

**It will be get executed when method throw an exception**