Spring AOP

AOP breaks the program logic into distinct parts (called concerns). Example it will separate the primary logic and secondary logic

Ex: - transaction management, authentication, logging, security before on every method execution

#### **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** Test {

**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 java file.

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

## **AOP Concepts and Terminology**

AOP concepts and terminologies are as follows:

1. Join point
2. Advice
3. Pointcut
4. Target Object
5. Aspect

#### **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.

#### **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.

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.

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

**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:

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

It will be applied on all the public methods.

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

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

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

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

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

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

**@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

**public** **class** Operation{

**public** **void** **msg**(){System.out.println("msg method invoked");}

**public** **int** **m**(){System.out.println("m method invoked");**return** 2;}

**public** **int** **k**(){System.out.println("k method invoked");**return** 3;}

}

Now, create the aspect class that contains before advice.

File: TrackOperation.java

**import** org.aspectj.lang.JoinPoint;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Before;

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

@Aspect

**public** **class** TrackOperation{

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

**public** **void** k(){}//pointcut name

@Before("k()")//applying pointcut on before advice

**public** **void** myadvice(JoinPoint jp)//it is advice (before advice)

{

System.out.println("additional concern");

//System.out.println("Method Signature: " + jp.getSignature());

}

}

Now create the applicationContext.xml file that defines beans.

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: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">

<bean id="opBean" **class**="com.epam.Operation"> </bean>

<bean id="trackMyBean" **class**="com.epam.TrackOperation"></bean>

<bean **class**="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator"></bean>

</beans>

Now, let's call the actual method.

File: Test.java

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

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

**public** **class** Test{

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

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

Operation e = (Operation) context.getBean("opBean");

System.out.println("calling msg...");

e.msg();

System.out.println("calling m...");

e.m();

System.out.println("calling k...");

e.k();

}

}

#### **Output**

calling msg...

additional concern

msg() method invoked

calling m...

additional concern

m() method invoked

calling k...

additional concern

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:

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

Now additional concern will be applied for the methods starting with m in Operation class. Output will be as this:  
 calling msg...    
 additional concern    
 msg() method invoked    
 calling m...    
 additional concern    
 m() method invoked    
 calling k...    
 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

**import** org.aspectj.lang.JoinPoint;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.After;

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

@Aspect

**public** **class** TrackOperation{

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

**public** **void** k(){}//pointcut name

@After("k()")//applying pointcut on after advice

**public** **void** myadvice(JoinPoint jp)//it is advice (after advice)

{

System.out.println("additional concern");

//System.out.println("Method Signature: " + jp.getSignature());

}

}

#### **Output**

calling msg...

msg() method invoked

additional concern

calling m...

m() method invoked

additional concern

calling k...

k() method invoked

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

**public** **class** Operation{

**public** **int** m(){System.out.println("m() method invoked");**return** 2;}

**public** **int** k(){System.out.println("k() method invoked");**return** 3;}

}

Create the aspect class that contains after returning advice.

File: TrackOperation.java

**import** org.aspectj.lang.JoinPoint;

**import** org.aspectj.lang.annotation.AfterReturning;

**import** org.aspectj.lang.annotation.Aspect;

@Aspect

**public** **class** TrackOperation{

@AfterReturning(

pointcut = "execution(\* Operation.\*(..))",

returning= "result")

**public** **void** myadvice(JoinPoint jp,Object result)//it is advice (after returning advice)

{

System.out.println("additional concern");

System.out.println("Method Signature: " + jp.getSignature());

System.out.println("Result in advice: "+result);

System.out.println("end of after returning advice...");

}

}

Now create the Test **class** that calls the actual methods.

File: Test.java

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

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

**public** **class** Test{

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

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

Operation e = (Operation) context.getBean("opBean");

System.out.println("calling m...");

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

System.out.println("calling k...");

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

}

}

#### **Output**

calling m...

m() method invoked

additional concern

Method Signature: **int** com.epam.Operation.m()

Result in advice: 2

end of after returning advice...

2

calling k...

k() method invoked

additional concern

Method Signature: **int** com.epam.Operation.k()

Result in advice: 3

end of after returning advice...

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.

Create a class that contains actual business logic.

File: Operation.java

**public** **class** Operation{

**public** **void** msg(){System.out.println("msg() is invoked");}

**public** **void** display(){System.out.println("display() is invoked");}

}

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

**import** org.aspectj.lang.ProceedingJoinPoint;

**import** org.aspectj.lang.annotation.Around;

**import** org.aspectj.lang.annotation.Aspect;

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

@Aspect

**public** **class** TrackOperation

{

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

**public** **void** abcPointcut(){}

@Around("abcPointcut()")

**public** Object myadvice(ProceedingJoinPoint pjp) **throws** Throwable

{

System.out.println("Additional Concern Before calling actual method");

Object obj=pjp.proceed();

System.out.println("Additional Concern After calling actual method");

**return** obj;

}

}

Now create the Test class that calls the actual methods.

File: Test.java

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

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

**public** **class** Test{

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

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

Operation op = (Operation) context.getBean("opBean");

op.msg();

op.display();

}

}

#### **Output**

Additional Concern Before calling actual method

msg() is invoked

Additional Concern After calling actual method

Additional Concern Before calling actual method

display() is invoked

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

**public** **class** Operation{

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

**if**(age<18){

**throw** **new** ArithmeticException("Not valid age");

}

**else**{

System.out.println("Thanks for vote");

}

}

}

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

**import** org.aspectj.lang.JoinPoint;

**import** org.aspectj.lang.annotation.AfterThrowing;

**import** org.aspectj.lang.annotation.Aspect;

@Aspect

**public** **class** TrackOperation{

@AfterThrowing(

pointcut = "execution(\* Operation.\*(..))",

throwing= "error")

**public** **void** myadvice(JoinPoint jp,Throwable error)//it is advice

{

System.out.println("additional concern");

System.out.println("Method Signature: " + jp.getSignature());

System.out.println("Exception is: "+error);

System.out.println("end of after throwing advice...");

}

}

Now create the Test class that calls the actual methods.

File: Test.java

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

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

**public** **class** Test{

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

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

Operation op = (Operation) context.getBean("opBean");

System.out.println("calling validate...");

**try**{

op.validate(19);

}**catch**(Exception e){System.out.println(e);}

System.out.println("calling validate again...");

**try**{

op.validate(11);

}**catch**(Exception e){System.out.println(e);}

}

}

#### **Output**

calling validate...

Thanks **for** vote

calling validate again...

additional concern

Method Signature: **void** com.javatpoint.Operation.validate(**int**)

Exception is: java.lang.ArithmeticException: Not valid age

end of after throwing advice...

java.lang.ArithmeticException: Not valid age