Aspect-Oriented Programming (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. (Such concerns are often termed crosscutting concerns in AOP literature.)

One of the key components of Spring is the AOP framework. While the Spring IoC container does not depend on AOP, meaning you do not need to use AOP if you don't want to, AOP complements Spring IoC to provide a very capable middleware solution.

AOP is used in the Spring Framework to...

* ... provide declarative enterprise services, especially as a replacement for EJB declarative services. The most important such service is [declarative transaction management](https://docs.spring.io/spring/docs/2.5.x/reference/transaction.html#transaction-declarative).
* ... allow users to implement custom aspects, complementing their use of OOP with AOP.

Most of the enterprise applications have some common crosscutting concerns that is applicable for different types of Objects and modules. Some of the common crosscutting concerns are logging, transaction management, data validation etc. In Object Oriented Programming, modularity of application is achieved by Classes whereas in Aspect Oriented Programming application modularity is achieved by Aspects and they are configured to cut across different classes.

Spring AOP takes out the direct dependency of crosscutting tasks from classes that we can’t achieve through normal object oriented programming model. For example, we can have a separate class for logging but again the functional classes will have to call these methods to achieve logging across the application.

One of the key components of Spring Framework is the **Aspect oriented programming (AOP)** framework. Aspect-Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like logging, auditing, declarative transactions, security, caching, etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect. AOP is like triggers in programming languages such as Perl, .NET, Java, and others.

Spring AOP module provides interceptors to intercept an application. For example, when a method is executed, you can add extra functionality before or after the method execution.

**Spring AOP and AspectJ**

In Spring AOP some of the Aspect oriented programming functionality is provided but has its own limitation. The implementation provided by spring AOP can be applied to the objects managed by spring (it should be spring bean).

Spring AOP can only intercept methods call if we want intercept any fields value has been changed spring cannot be used.

If we want to intercept any field level thing or any class which is not managed by spring AOP we have to use full-fledged AspectJ.

**Aspects:** in simple words aspect is the functionality we are trying to achieve in multiple classes.A standard code/feature that is scattered across multiple places in the application and is typically different than the actual Business Logic (for example, Transaction management). Each aspect focuses on a specific cross-cutting functionality

**PointCut:** is the expression which determines which methods where the aspect will be applied.

**Advice:** This is the code needs to be executed for every time one method is intercepted .It can before method execution ,after execution etc.

Example:@Before,@After,@AfterReturning

**Joinpoint:** A joinpoint is a candidate point in the **Program Execution** of the application where an aspect can be plugged in. This point could be a method being called, constructor is called an exception being thrown, or even a field being modified. These are the points where your aspect’s code can be inserted into the normal flow of your application to add new behaviour.

Ex. Spring AOP can intercept method calls .so method call of some service class is joint point. For example if we are trying to intercept the modification of some field in some service using AspectJ framework then modification of field is the trigger point and that can be called as joinpoint.

**Weaving:** the process of linking aspects with targeted objects to create an advised object

**AOP concepts**

**Aspect:** a modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in J2EE applications. In Spring AOP, aspects are implemented using regular classes (the [schema-based approach](https://docs.spring.io/spring/docs/2.5.x/reference/aop.html#aop-schema)) or regular classes annotated with the @Aspect annotation (the [@AspectJ style](https://docs.spring.io/spring/docs/2.5.x/reference/aop.html#aop-ataspectj)).

1. [**schema-based approach**](https://docs.spring.io/spring/docs/2.5.x/reference/aop.html#aop-schema)
2. <?xml version="1.0" encoding="UTF-8"?>
3. <beans xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. **xmlns:aop="http://www.springframework.org/schema/aop"**
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
8. **http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"**>
9. <!-- <bean/> definitions here -->
10. </beans>

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

Aop in 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.

How to enable AOP?

With Spring, you can declare advice using AspectJ annotations, but you must first apply the @EnableAspectJAutoProxy annotation to your configuration class, which will enable support for handling components marked with AspectJ’s @Aspect annotation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Configuration  @ComponentScan(basePackages = {"org.baeldung.dao", "org.baeldung.aop"})  @EnableAspectJAutoProxy  public class TestConfig {      ...  } |

**Before advice**

This advice, as the name implies, is executed before the join point. It does not prevent the continued execution of the method it advises unless an exception is thrown.

Consider the following aspect that simply logs the method name before it is called:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @Component  @Aspect  public class LoggingAspect {        private Logger logger = Logger.getLogger(LoggingAspect.class.getName());        @Pointcut("@target(org.springframework.stereotype.Repository)")      public void repositoryMethods() {};        @Before("repositoryMethods()")      public void logMethodCall(JoinPoint jp) {          String methodName = jp.getSignature().getName();          logger.info("Before " + methodName);      }  } |

### @AfterReturning Advice

@AfterReturning annotation is used to create *after returning* advice. @AfterReturning has attribute as *returning* which can be used to get retuned data from a method which is covered by defined pointcut with @AfterReturningannotation.   
**TimeLoggingAspect.java**

package com.concretepage.aspect;

import java.util.Date;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.Aspect;

import org.springframework.stereotype.Component;

@Component

@Aspect

public class TimeLoggingAspect {

@AfterReturning (pointcut = "execution (\* com.concretepage.service.\*.\*(..))",

returning="val")

public void logAfterReturning(Object val){

System.out.println("Method return value:"+ val);

System.out.println ("@AfterReturning:"+new Date ());

}

}

Find the output.

2\*3= 6

Method return value:6

@AfterReturning:Wed Dec 16 20:58:41 IST 2015

### @AfterThrowing Advice

Using AOP we can also create an advice which will run just after throwing any exception. For this we need to use @AfterThrowing annotation to define our advice in aspect class. To work with exception, @AfterThrowing has an attribute as *throwing*.   
**TimeLoggingAspect.java**

package com.concretepage.aspect;

import java.util.Date;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Aspect;

import org.springframework.stereotype.Component;

@Component

@Aspect

public class TimeLoggingAspect {

@AfterThrowing(pointcut = "execution(\* com.concretepage.service.\*.\*(..))",

throwing="exception")

public void logAfterThrowing(Exception exception){

System.out.println("@AfterReturning:"+new Date());

System.out.println("Exception caught:"+ exception.getMessage());

}

}

Find the UserService, in which we are dividing a number by another number.   
**UserService.java**

package com.concretepage.service;

import org.springframework.stereotype.Service;

@Service

public class UserService {

public Integer devide(int a, int b){

int res = a/b;

System.out.println(a+ "/" + b +"= " + res);

return res;

}

}

To throw the exception, we will use zero divisor. In SpringAOPTest, we need to use below line.

UserService userService = ctx.getBean(UserService.class);

userService.devide(5,0);

What we will observe that the exception will be caught by @AfterThrowing. Find the output.

@AfterReturning:Thu Dec 17 11:31:32 IST 2015

Exception caught:/ by zero

Exception in thread "main" java.lang.ArithmeticException: / by zero

at com.concretepage.service.UserService.devide(UserService.java:6)

**PointCut**

A pointcut expression can appear as a value of the @Pointcut annotation:

|  |  |
| --- | --- |
| 1  2 | @Pointcut("within(@org.springframework.stereotype.Repository \*)")  public void repositoryClassMethods() {} |

The method declaration is called the **pointcut signature**. It provides a name that can be used by advice annotations to refer to that pointcut.

|  |  |
| --- | --- |
| 1  2  3  4 | @Around("repositoryClassMethods()")  public Object measureMethodExecutionTime(ProceedingJoinPoint pjp) throws Throwable {      ...  } |

A pointcut expression could also appear as the value of the expression property of an aop:pointcut tag:

|  |  |
| --- | --- |
| 1  2  3  4 | <aop:config>      <aop:pointcut id="anyDaoMethod"        expression="@target(org.springframework.stereotype.Repository)"/>  </aop:config> |

## **3. Pointcut Designators**

A pointcut expression starts with a **pointcut designator (PCD)**, which is a keyword telling Spring AOP what to match. There are several pointcut designators, such as the execution of a method, a type, method arguments, or annotations.

### ****3.1****execution

The primary Spring PCD is execution, which matches method execution join points.

|  |  |
| --- | --- |
| 1 | @Pointcut("execution(public String org.baeldung.dao.FooDao.findById(Long))") |

This example pointcut will match exactly the execution of findById method of the FooDao class. This works, but it is not very flexible. Suppose we would like to match all the methods of the FooDao class, which may have different signatures, return types, and arguments. To achieve this we may use wildcards:

|  |  |
| --- | --- |
| 1 | @Pointcut("execution(\* org.baeldung.dao.FooDao.\*(..))") |

Here the first wildcard matches any return value, the second matches any method name, and the (..) pattern matches any number of parameters (zero or more).

### ****3.2****within

Another way to achieve the same result from the previous section is by using the within PCD, which limits matching to join points of certain types.

|  |  |
| --- | --- |
| 1 | @Pointcut("within(org.baeldung.dao.FooDao)") |

We could also match any type within the org.baeldung package or a sub-package.

|  |  |
| --- | --- |
| 1 | @Pointcut("within(org.baeldung..\*)") |

### ****3.3****this****and****target

this limits matching to join points where the bean reference is an instance of the given type, while target limits matching to join points where the target object is an instance of the given type. The former works when Spring AOP creates a CGLIB-based proxy, and the latter is used when a JDK-based proxy is created. Suppose that the target class implements an interface:

|  |  |
| --- | --- |
| 1  2  3 | public class FooDao implements BarDao {      ...  } |

In this case, Spring AOP will use the JDK-based proxy and you should use the target PCD because the proxied object will be an instance of Proxy class and implement the BarDao interface:

|  |  |
| --- | --- |
| 1 | @Pointcut("target(org.baeldung.dao.BarDao)") |

On the other hand if FooDao doesn’t implement any interface or proxyTargetClass property is set to true then the proxied object will be a subclass of FooDao and the this PCD could be used:

|  |  |
| --- | --- |
| 1 | @Pointcut("this(org.baeldung.dao.FooDao)") |

### ****3.4****args

This PCD is used for matching particular method arguments:

|  |  |
| --- | --- |
| 1 | @Pointcut("execution(\* \*..find\*(Long))") |

This pointcut matches any method that starts with find and has only one parameter of type Long. If we want to match a method with any number of parameters but having the fist parameter of type Long, we could use the following expression:

|  |  |
| --- | --- |
| 1 | @Pointcut("execution(\* \*..find\*(Long,..))") |

### ****3.5****@target

The @target PCD (not to be confused with the target PCD described above) limits matching to join points where the class of the executing object has an annotation of the given type:

|  |  |
| --- | --- |
| 1 | @Pointcut("@target(org.springframework.stereotype.Repository)") |

### ****3.6****@args

This PCD limits matching to join points where the runtime type of the actual arguments passed have annotations of the given type(s). Suppose that we want to trace all the methods accepting beans annotated with @Entity annotation:

|  |  |
| --- | --- |
| 1  2 | @Pointcut("@args(org.baeldung.aop.annotations.Entity)")  public void methodsAcceptingEntities() {} |

To access the argument we should provide a JoinPoint argument to the advice:

|  |  |
| --- | --- |
| 1  2  3  4 | @Before("methodsAcceptingEntities()")  public void logMethodAcceptionEntityAnnotatedBean(JoinPoint jp) {      logger.info("Accepting beans with @Entity annotation: " + jp.getArgs()[0]);  } |

### ****3.7****@within

This PCD limits matching to join points within types that have the given annotation:

|  |  |
| --- | --- |
| 1 | @Pointcut("@within(org.springframework.stereotype.Repository)") |

Which is equivalent to:

|  |  |
| --- | --- |
| 1 | @Pointcut("within(@org.springframework.stereotype.Repository \*)") |

### ****3.8****@annotation

This PCD limits matching to join points where the subject of the join point has the given annotation. For example we may create a @Loggable annotation:

|  |  |
| --- | --- |
| 1  2 | @Pointcut("@annotation(org.baeldung.aop.annotations.Loggable)")  public void loggableMethods() {} |

Then we may log execution of the methods marked by that annotation:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Before("loggableMethods()")  public void logMethod(JoinPoint jp) {      String methodName = jp.getSignature().getName();      logger.info("Executing method: " + methodName);  } |

## **4. Combining Pointcut Expressions**

Pointcut expressions can be combined using **&&**, **||** and **!** operators:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Pointcut("@target(org.springframework.stereotype.Repository)")  public void repositoryMethods() {}    @Pointcut("execution(\* \*..create\*(Long,..))")  public void firstLongParamMethods() {}    @Pointcut("repositoryMethods() && firstLongParamMethods()")  public void entityCreationMethods() {} |

The use of annotations is a precise way to define when an aspect should be run. They are only run when a developer has used the annotation on an object or method.

**The errors that can and will occur**

Using only annotations creates another problem that we don’t need to think about while using patterns; It will make our advice run twice(or more), because the annotation pointcut don’t specify if it should be run during execution or initialization. The reason for the advice in the pattern example not being executed twice is that the pattern uses the [combinator](https://eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html) execute(pattern) . We are practically saying that the advice should only look for code that’s executed. There are different [combinators](https://eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html)that we can use to define when we should run our advice; one of them is execute.

So instead of only using annotations we need to use annotations and a [combinator](https://eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html)with a pattern. The simplest way is to use a catch all pattern with the [combinator](https://eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html)execution and then combine it with an annotation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | package com.jayway.blog;  import org.aspectj.lang.ProceedingJoinPoint;  import org.aspectj.lang.annotation.\*;  import org.aspectj.lang.JoinPoint;    @Aspect  public class YourAspect {      //Defines a pointcut where the @YourAnnotation exists      //And combines that with a catch all pointcut with the scope of execution      @Around("@annotation(YourAnnotation) && execution(\* \*(..))")      //ProceedingJointPoint = the reference of the call to the method.      //Difference between ProceedingJointPoint and JointPoint is that a JointPoint can't be continued(proceeded)      //A ProceedingJointPoint can be continued(proceeded) and is needed for a Around advice      public Object aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable {          //Default Object that we can use to return to the consumer          Object returnObject = null;          try {              System.out.println("YourAspect's aroundAdvice's body is now executed Before yourMethodAround is called.");              //We choose to continue the call to the method in question              returnObject = joinPoint.proceed();              //If no exception is thrown we should land here and we can modify the returnObject, if we want to.          } catch (Throwable throwable) {              //Here we can catch and modify any exceptions that are called              //We could potentially not throw the exception to the caller and instead return "null" or a default object.              throw throwable;          }          finally {              //If we want to be sure that some of our code is executed even if we get an exception              System.out.println("YourAspect's aroundAdvice's body is now executed After yourMethodAround is called.");          }          return returnObject;      }  } |

In this example we are using the [combinator](https://eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html) execute with a catch-all pattern and annotations. Basically we are looking for any code that’s executed and has the annotation @YourAnnotation. There is a whole list of different [combinators](https://eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html) at [AspectJ’s](https://eclipse.org/aspectj/)homepage. Each one of them can take different patterns to help you define when your advice should be triggered.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | package com.jayway.blog;    public class YourClass {        public static void main(String[] args) {          YourClass yourClass = new YourClass();          yourClass.yourMethodAround();      }        @YourAnnotation      public void yourMethodAround(){          System.out.println("Executing TestTarget.yourMethodAround()");      }  } |

By adding the @YourAnnotation before any method the aspect will find the annotation and execute the advice body.

#### **Complexity and @Pointcut**

When there is a need to define pointcuts that are a bit more complex we can define a standalone pointcut that we can reuse. By using the @Pointcut attribute we can define a specific pointcut and when it should get run. We can then use the name of the pointcut as a reference in the @Before, @After, @AfterThrowing, @AfterReturn and @Around.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | package com.jayway.blog;    import org.aspectj.lang.ProceedingJoinPoint;  import org.aspectj.lang.annotation.\*;  import org.aspectj.lang.JoinPoint;    @Aspect  public class YourAspect {        //Defines a pointcut that we can use in the @Before,@After,  // @AfterThrowing,@AfterReturning,@Around specifications      //The pointcut will look for the @YourAnnotation      @Pointcut("@annotation(YourAnnotation)")      public void annotationPointCutDefinition(){      }        //Defines a pointcut that we can use in the @Before,@After, @AfterThrowing, @AfterReturning,@Around specifications      //The pointcut is a catch all pointcut with the scope of execution      @Pointcut("execution(\* \*(..))")      public void atExecution(){}        @After("annotationPointCutDefinition() && atExecution()")      //JointPoint = the reference of the call to the method      public void printNewLine(JoinPoint pointcut){          //Just prints new lines after each method that's executed in          System.out.print("\n\r");      }  } |

We got the same result as the earlier example but we are using the name of the pointcuts so we can reuse them.

#### **Conclusion**

Annotations can be a precise tool, as a pointcut will not trigger if the annotations are not in the code. But there are some weak points that we need to try and cover up with the use of combinatory and patterns. Combining patterns and annotations is a good middle road when we want to specify when the advice should be run.