**Spring AOP Overview**

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.

**Aspect Oriented Programming Core Concepts**

Before we dive into implementation of Spring AOP implementation, we should understand the core concepts of AOP.

1. **Aspect**: An aspect is a class that implements enterprise application concerns that cut across multiple classes, such as transaction management. Aspects can be a normal class configured through Spring XML configuration or we can use Spring AspectJ integration to define a class as Aspect using @Aspectannotation.
2. **Join Point**: A join point is the specific point in the application such as method execution, exception handling, changing object variable values etc. In Spring AOP a join points is always the execution of a method.
3. **Advice**: Advices are actions taken for a particular join point. In terms of programming, they are methods that gets executed when a certain join point with matching pointcut is reached in the application. You can think of Advices as [Struts2 interceptors](http://www.journaldev.com/2210/struts-2-interceptor-example) or [Servlet Filters](http://www.journaldev.com/1933/java-servlet-filter-example-tutorial).
4. **Pointcut**: Pointcut are expressions that is matched with join points to determine whether advice needs to be executed or not. Pointcut uses different kinds of expressions that are matched with the join points and Spring framework uses the AspectJ pointcut expression language.
5. **Target Object**: They are the object on which advices are applied. Spring AOP is implemented using runtime proxies so this object is always a proxied object. What is means is that a subclass is created at runtime where the target method is overridden and advices are included based on their configuration.
6. **AOP proxy**: Spring AOP implementation uses JDK dynamic proxy to create the Proxy classes with target classes and advice invocations, these are called AOP proxy classes. We can also use CGLIB proxy by adding it as the dependency in the Spring AOP project.
7. **Weaving**: It is the process of linking aspects with other objects to create the advised proxy objects. This can be done at compile time, load time or at runtime. Spring AOP performs weaving at the runtime.

**AOP Advice Types**

Based on the execution strategy of advices, they are of following types.

1. **Before Advice**: These advices runs before the execution of join point methods. We can use @Beforeannotation to mark an advice type as Before advice.
2. **After (finally) Advice**: An advice that gets executed after the join point method finishes executing, whether normally or by throwing an exception. We can create after advice using @After annotation.
3. **After Returning Advice**: Sometimes we want advice methods to execute only if the join point method executes normally. We can use @AfterReturning annotation to mark a method as after returning advice.
4. **After Throwing Advice**: This advice gets executed only when join point method throws exception, we can use it to rollback the transaction declaratively. We use @AfterThrowing annotation for this type of advice.
5. **Around Advice**: This is the most important and powerful advice. This advice surrounds the join point method and we can also choose whether to execute the join point method or not. We can write advice code that gets executed before and after the execution of the join point method. It is the responsibility of around advice to invoke the join point method and return values if the method is returning something. We use @Around annotation to create around advice methods.

The points mentioned above may sound confusing but when we will look at the implementation of Spring AOP, things will be more clear. Let’s start creating a simple Spring project with AOP implementations. Spring provides support for using AspectJ annotations to create aspects and we will be using that for simplicity. All the above AOP annotations are defined in org.aspectj.lang.annotation package.

## 6.4. Choosing which AOP declaration style to use

Once you have decided that an aspect is the best approach for implementing a given requirement, how do you decide between using Spring AOP or AspectJ, and between the Aspect language (code) style, @AspectJ annotation style, or the Spring XML style? These decisions are influenced by a number of factors including application requirements, development tools, and team familiarity with AOP.

### 6.4.1. Spring AOP or full AspectJ?

Use the simplest thing that can work. Spring AOP is simpler than using full AspectJ as there is no requirement to introduce the AspectJ compiler / weaver into your development and build processes. If you only need to advise the execution of operations on Spring beans, then Spring AOP is the right choice. If you need to advise objects not managed by the Spring container (such as domain objects typically), then you will need to use AspectJ. You will also need to use AspectJ if you wish to advise join points other than simple method executions (for example, field get or set join points, and so on).

When using AspectJ, you have the choice of the AspectJ language syntax (also known as the "code style") or the @AspectJ annotation style. Clearly, if you are not using Java 5+ then the choice has been made for you... use the code style. If aspects play a large role in your design, and you are able to use the [AspectJ Development Tools (AJDT)](https://www.eclipse.org/ajdt/) plugin for Eclipse, then the AspectJ language syntax is the preferred option: it is cleaner and simpler because the language was purposefully designed for writing aspects. If you are not using Eclipse, or have only a few aspects that do not play a major role in your application, then you may want to consider using the @AspectJ style and sticking with a regular Java compilation in your IDE, and adding an aspect weaving phase to your build script.

### 6.4.2. @AspectJ or XML for Spring AOP?

If you have chosen to use Spring AOP, then you have a choice of @AspectJ or XML style. Clearly if you are not running on Java 5+, then the XML style is the appropriate choice; for Java 5 projects there are various tradeoffs to consider.

The XML style will be most familiar to existing Spring users. It can be used with any JDK level (referring to named pointcuts from within pointcut expressions does still require Java 5+ though) and is backed by genuine POJOs. When using AOP as a tool to configure enterprise services then XML can be a good choice (a good test is whether you consider the pointcut expression to be a part of your configuration you might want to change independently). With the XML style arguably it is clearer from your configuration what aspects are present in the system.

The XML style has two disadvantages. Firstly it does not fully encapsulate the implementation of the requirement it addresses in a single place. The DRY principle says that there should be a single, unambiguous, authoritative representation of any piece of knowledge within a system. When using the XML style, the knowledge of how a requirement is implemented is split across the declaration of the backing bean class, and the XML in the configuration file. When using the @AspectJ style there is a single module - the aspect - in which this information is encapsulated. Secondly, the XML style is slightly more limited in what it can express than the @AspectJ style: only the "singleton" aspect instantiation model is supported, and it is not possible to combine named pointcuts declared in XML. For example, in the @AspectJ style you can write something like:

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

public void propertyAccess() {}

@Pointcut(execution(org.xyz.Account+ \*(..))

public void operationReturningAnAccount() {}

@Pointcut(propertyAccess() && operationReturningAnAccount())

public void accountPropertyAccess() {}

In the XML style I can declare the first two pointcuts:

<aop:pointcut id="propertyAccess"

expression="execution(\* get\*())"/>

<aop:pointcut id="operationReturningAnAccount"

expression="execution(org.xyz.Account+ \*(..))"/>

The downside of the XML approach is that you cannot define the 'accountPropertyAccess' pointcut by combining these definitions.

The @AspectJ style supports additional instantiation models, and richer pointcut composition. It has the advantage of keeping the aspect as a modular unit. It also has the advantage the @AspectJ aspects can be understood (and thus consumed) both by Spring AOP and by AspectJ - so if you later decide you need the capabilities of AspectJ to implement additional requirements then it is very easy to migrate to an AspectJ-based approach. On balance the Spring team prefer the @AspectJ style whenever you have aspects that do more than simple "configuration" of enterprise services.

Spring-AOP Pros

* It is simpler to use than AspectJ, since you don't have to use LTW ([load-time weaving](http://www.eclipse.org/aspectj/doc/next/devguide/ltw.html)) or the AspectJ compiler.
* This can be change to AspectJ AOP when you use @Aspect annotation based Spring AOP.
* This use Proxy pattern and Decorator pattern

Spring-AOP Cons

* This is proxy-based AOP, so basically you can only use method-execution pointcut.
* Aspects aren't applied when calling another method within the same class.
* There can be a little runtime overhead.
* Spring-AOP cannot add an aspect to anything that is not created by the Spring factory

AspectJ Pros

* This supports all pointcuts. This means you can do anything.
* There is less runtime overhead than that of Spring AOP.

AspectJ Cons

* Be careful. Check if your aspects are weaved to only what you wanted to be weaved.
* You need extra build process with AspectJ Compiler or have to setup LTW (load-time weaving)

Apart from what others have stated - just to rephrase, there are two major differences:

1. One is related to the type of weaving.
2. Another to the joinpoint definition.

**Spring-AOP:** Runtime weaving through proxy using concept of dynamic proxy if interface exists or cglib library if direct implementation provided.

**AspectJ:** Compile time weaving through AspectJ Java Tools(ajc compiler) if source available or post compilation weaving (using compiled files). Also, load time weaving with Spring can be enabled - it needs the aspectj definition file and offers flexibility.

Compile time weaving can offer benefits of performance (in some cases) and also the joinpoint definition in Spring-aop is restricted to method definition only which is not the case for AspectJ.

An additional note: **If performance under high load is important, you'll want AspectJ which is 9-35x faster than Spring AOP**. 10ns vs 355ns might not sound like much, but I've seen people using LOTS of Aspects. 10K's worth of aspects. In these cases, your request might hit a thousands of aspects. In that case you're adding ms to that request.

<https://www.mkyong.com/spring3/spring-aop-aspectj-annotation-example/>