<http://www.javatpoint.com/spring-aop-aspectj-xml-configuration-example>

<https://www.tutorialspoint.com/spring/aop_with_spring.htm>

**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.

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.

**Aspect:** A module which has set of APIs providing cross-cutting requirements. For example, a logging module would be called as AOP Aspect or logging. An application can have any number of aspects depending on the requirement.

**Join Point:** This represents a point in your application where you can plug-in AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.

**Advice:** This is the actual action to be taken either before or after the method execution. This is the actual piece of code that is invoked during program execution by Spring AOP framework.

**Pointcut:** This is a set of one or more joinpoints where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP framework.

**Types of Advice:**

**Before:** Run advice before method execution.

**After:** Run advice after method execution regardless of its outcome.

**After-returning:** Run advice after the method execution only if method completes successfully.

**After-throwing:** Run advice after the method execution only if method exits by throwing an exception.

**Around:** Run advice before and after the advised method is invoked.

Spring supports the @AspectJ annotation style approach and the schema-based approach to implement custom objects.

**XML Schema based:**

Aspects are implemented using regular classes along with XML based configuration.

**AspectJ based:** @AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations.

**Why use AOP?**

It provides pluggable way to dynamically add the additional concern before, after or around the actual logic.

There are 5 methods starts from m, 2 methods start from n and 3 methods starts from p.

**Understanding scenario:** I have to maintain log and send notification after calling method that starts with 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 starting with m. It leads to 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.

**Spring AOP AspectJ Annotation Example:**

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:

* @Before
* @After
* @AfterReturning
* @Around
* @AfterThrowing