**Aspect Oriented Programming**

**AOP is programming paradigm that aims to increase modularity by allowing separation of cross cutting concern**

 These cross-cutting concerns are different from the main business logic. We can add additional behavior to existing code without modification of the code itself.

AOP with Spring

* [Spring](https://www.geeksforgeeks.org/introduction-to-spring-framework/)**:** It uses XML based configuration for implementing AOP, also it uses annotations which are interpreted by using a library supplied by AspectJ for parsing and matching.

**Common terminologies in AOP:**

**1.Aspect:** The class which implements the JEE application cross-cutting concerns(transaction, logger etc) is known as the aspect. It can be normal class configured through XML configuration or through regular classes annotated with @Aspect. An aspect is a module that encapsulates **advice** and **pointcuts**

and provides **cross-cutting** An application can have any number of aspects

**2.Advice:** The job which is meant to be done by an Aspect or it can be defined as the action taken by the Aspect at a particular point. There are five types of Advice namely: Before, After, Around, AfterThrowing and AfterReturning. Let’s have a brief discussion about all the five types.

3. **JoinPoints:** An application has thousands of opportunities or points to apply Advice. These points are known as join points. For example – Advice can be applied at every invocation of a method or exception be thrown or at various other points.

**4.Pointcut:** Since it is not feasible to apply advice at every point of the code, therefore, the selected join points where advice is finally applied are known as the Pointcut. Often you specify these pointcuts using explicit class and method names or through regular expressions that define a matching class and method name patterns.

**Types of Advice** :

1. **Before:** Runs before the advised method is invoked. It is denoted by **@Before** annotation.
2. **After:** Runs after the advised method completes regardless of the outcome, whether successful or not. It is denoted by **@After** annotation.
3. **AfterReturning:** Runs after the advised method successfully completes ie without any runtime exceptions. It is denoted by **@AfterReturning** annotation.
4. **Around:** This is the strongest advice among all the advice since it wraps around and runs before and after the advised method. This type of advice is used where we need frequent access to a method or database like- caching. It is denoted by **@Around** annotation.
5. **AfterThrowing:** Runs after the advised method throws a Runtime Exception. It is denoted by **@AfterThrowing** annotation.