Aspect Oriented Programming (AOP) in Spring Framework

Last Updated : 07 Aug, 2025

Spring AOP (Aspect-Oriented Programming) is a programming technique in the Spring Framework that helps separate cross-cutting concerns (like logging, security, transactions) from the main business logic. Instead of adding this logic inside every class, AOP allows you to write it once and apply it wherever needed.

In Spring, AOP works using proxies and you can define behaviors using annotations like @Aspect, @Before, @After and @Around. This keeps your code clean, modular and easier to maintain by focusing only on the core business functionality while AOP handles the repetitive system-level tasks behind the scenes.

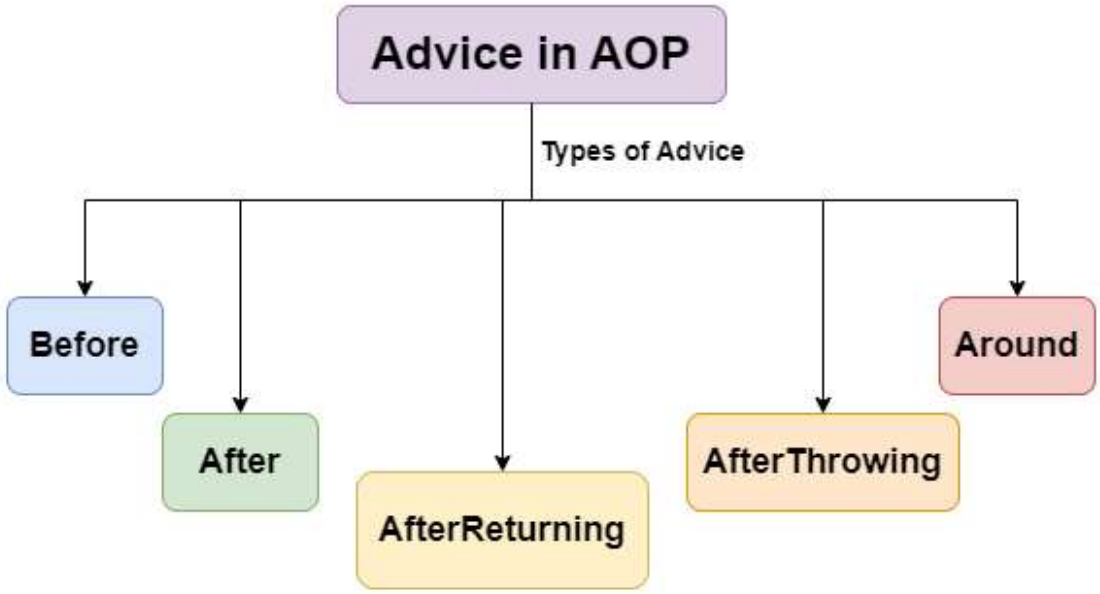
Understanding AOP Concepts

1. Aspect: An Aspect is a modular unit of cross-cutting concerns. For example, a logging aspect can be applied across various methods in different classes.

2. Advice: This is the action taken by an aspect at a particular join point. There are five types of advice:

- **Before**: Executed before the method call.
- **After**: Executed after the method call, regardless of its outcome.
- **AfterReturning**: Executed after the method returns a result, but not if an exception occurs.

- **Around:** Surrounds the method execution, allowing you to control the method execution and its result.
- **AfterThrowing**: Executed if the method throws an exception.



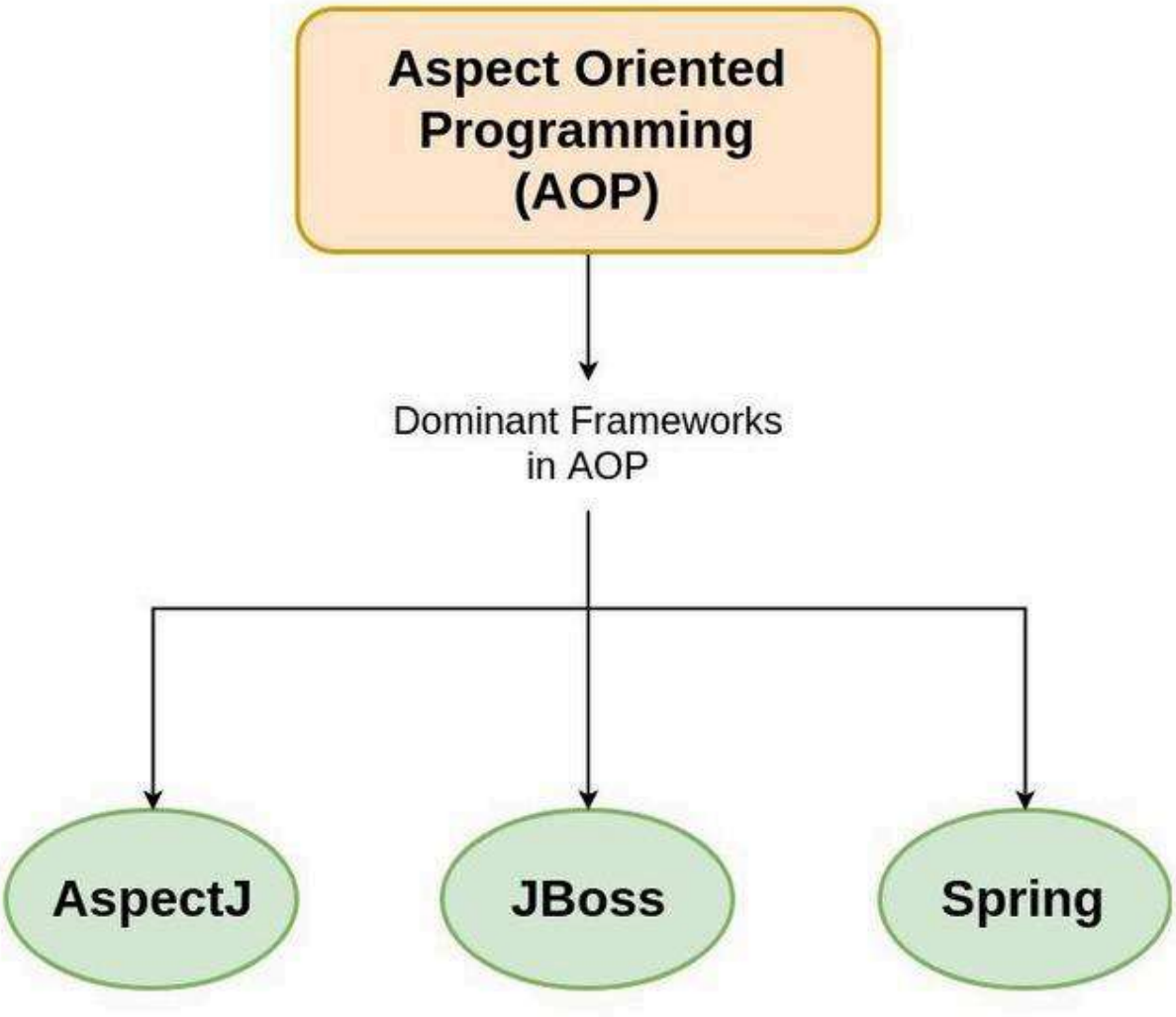
Advice in AOP

3. Join Point: A specific point in the execution of a program, such as method execution or exception handling, where an aspect can be applied.

4. Pointcut: A Pointcut is a predicate that defines where advice should be applied. It matches join points using expressions.

5. Weaving: This is the process of linking aspects with the target object. Spring AOP only supports runtime weaving using proxy-based mechanisms (JDK dynamic proxies for interfaces and CGLIB for concrete classes). It does not modify bytecode like AspectJ.

Dominant AOP Frameworks



AOP Frameworks

- **AspectJ**: A powerful and mature AOP framework that supports compile-time and load-time weaving. It offers full AOP support with its own syntax and tools.
- **JBoss AOP**: Part of the JBoss application server, offering integration with Java EE applications.
- **Spring AOP**: A simpler, proxy-based framework that integrates with the Spring Framework, using XML configurations or annotations to define aspects and pointcuts.

Example

Implementing Logging with AOP in Spring Aspect Class with Different Types of Advice

```java
package com.example.aspect;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Pointcut("execution(public void com.example.service.*.*(..))")
    public void allServiceMethods() {}

    @Before("allServiceMethods()")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("Before method: " +
joinPoint.getSignature().getName());
    }

    @After("allServiceMethods()")
    public void logAfter(JoinPoint joinPoint) {
        System.out.println("After method: " +
joinPoint.getSignature().getName());
    }

    @AfterReturning(pointcut = "allServiceMethods()", returning = "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        System.out.println("Method returned: " + result);
    }

    @AfterThrowing(pointcut = "allServiceMethods()", throwing = "error")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable error) {
        System.out.println("Method threw exception: " + error);
    }

    @Around("allServiceMethods()")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable
{
        System.out.println("Before and after method: " +
joinPoint.getSignature().getName());
        return joinPoint.proceed();
    }
}
```

Enabling AOP in Spring

Java-Based Configuration: To enable AOP in Spring, you need to configure your Spring application context appropriately.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@EnableAspectJAutoProxy
public class AopConfig {
}
```

XML Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans//"
       xmlns:xsi="https://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">

    <aop:aspectj-autoproxy/>

    <bean id="loggingAspect" class="com.example.aspect.LoggingAspect"/>

</beans>
```

Difference between Spring AOP and Spring IOC

Parameter	Spring AOP	Spring IOC
Full form	Spring Aspect-Oriented Programming	Spring Inversion of Control
Main Concept	Handling cross-cutting concerns	Dependency injection and object lifecycle management.

Parameter	Spring AOP	Spring IOC
Purpose	Logging, transaction management and security.	Managing object creation and injecting dependencies
Design Pattern	Proxy Pattern	Factory and Singleton pattern
Implementation	Using pointcuts, join points and advices.	Using BeanFactory or ApplicationContext.
Effects on code	Adds behavior without modifying actual code	Wires and manages object dependencies.
Configure	Annotation like @Aspact or via AOP namespace.	Java-based configurations or we can Configured XML file.