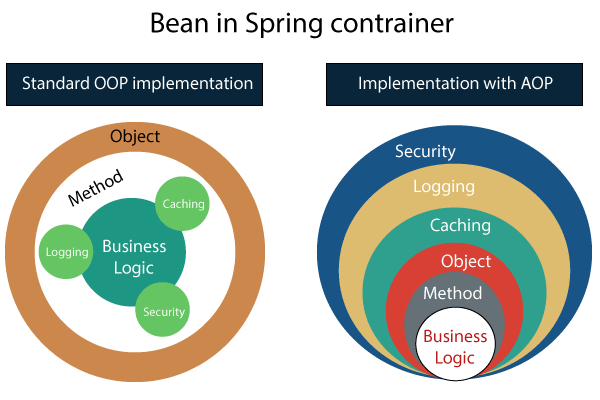
|  |  |
| --- | --- |
| **AOP** | **OOP** |
| **Aspect:** A code unit that encapsulates pointcuts, advices, and attributes. | **Class:** A code unit that encapsulates methods and attributes. |
| **Pointcut:** It defines the set of entry points in which advice is executed. | **Method signature:** It defines the entry points for the execution of method bodies. |
| **Advice:** It is an implementation of cross-cutting concerns. | **Method bodies:** It is an implementation of the business logic concerns. |
| **Waver: It constructs code (source or object) with advice.** | **Compiler: It converts source code to object code.** |



AOP

AOP **(Aspect-Oriented Programming)** is a programming pattern that increases modularity by allowing the separation of the **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.

Spring's AOP framework helps us to implement these cross-cutting concerns.

Using AOP, we define common functionality in one place. We are free to define how and where this functionality is applied without modifying the class to which we are applying the new feature. The cross-cutting concern can now be modularized into special classes, called **aspect**.

There are **two** benefits of aspects:

* First, the logic for each concern is now in one place instead of scattered all over the codebase.
* Second, the business modules only contain code for their primary concern. The secondary concern has been moved to the **aspect**.

The aspects have the responsibility that is to be implemented, called **advice**. We can implement an aspect's functionality into a program at one or more join points.

Benefits of AOP

* It is implemented in pure Java.
* There is no requirement for a special compilation process.
* It supports only method execution Join points.
* Only run time weaving is available.
* Two types of AOP proxy is available: **JDK dynamic proxy** and **CGLIB proxy.**

## Cross-cutting concern

The cross-cutting concern is a concern that we want to implement in multiple places in an application. It affects the entire application.

AOP Terminology

* **Aspect:** An aspect is a module that encapsulates **advice** and **pointcuts** and provides **cross-cutting** An application can have any number of aspects. We can implement an aspect using regular class annotated with **@Aspect** annotation.
* **Pointcut:** A pointcut is an expression that selects one or more join points where advice is executed. We can define pointcuts using **expressions** or **patterns**. It uses different kinds of expressions that matched with the join points. In Spring Framework, **AspectJ** pointcut expression language is used.
* **Join point:** A join point is a point in the application where we apply an **AOP aspect**. Or it is a specific execution instance of an advice. In AOP, join point can be a **method execution, exception handling, changing object variable value**, etc.
* **Advice:** The advice is an action that we take either **before** or **after** the method execution. The action is a piece of code that invokes during the program execution. There are **five** types of advices in the Spring AOP framework: **before, after, after-returning, after-throwing,**and **around advice.**Advices are taken for a particular **join point.**We will discuss these advices further in this section.
* **Target object:** An object on which advices are applied, is called the **target object**. Target objects are always a **proxied** It means a subclass is created at run time in which the target method is overridden, and advices are included based on their configuration.
* **Weaving:** It is a process of **linking aspects** with other application types. We can perform weaving at **run time, load time,** and **compile time**.

**Proxy:** It is an object that is created after applying advice to a target object is called **proxy**. The Spring AOP implements the **JDK dynamic proxy** to create the proxy classes with target classes and advice invocations. These are called AOP proxy classes.

Types of AOP Advices

There are five types of AOP advices are as follows:

**Pointcut**: Its Select The Function

**execution(expression):** It is an expression on which advice is to be applied.

* Before Advice
* After Advice
* Around Advice
* After Throwing
* After Returning

**Before Advice:** An advice that executes before a join point, is called before advice. We use **@Before** annotation to mark an advice as Before advice.

**After Advice:** An advice that executes after a join point, is called after advice. We use **@After**annotation to mark an advice as After advice.

**Around Advice:** An advice that executes before and after of a join point, is called around advice.

**After Throwing Advice:** An advice that executes when a join point throws an exception.

**After Returning Advice:** An advice that executes when a method executes successfully.

Before implementing the AOP in an application, we are required to add **Spring AOP** dependency in the pom.xml file

# **Spring Boot AOP Before Advice**

Before advice is used in Aspect-Oriented Programming to achieve the cross-cutting. It is an advice type which ensures that an advice runs before the method execution. We use **@Before** annotation to implement the before advice.

 Use this over the main class **@EnableAspectJAutoProxy.**

1. @Aspect
2. @Component
3. **public** **class** EmployeeServiceAspect
4. {
5. @Before(value = "execution(\* com.javatpoint.service.EmployeeService.\*(..)) and args(empId, fname, sname)")
6. **public** **void** beforeAdvice(JoinPoint joinPoint, String empId, String fname, String sname) {
7. System.out.println("Before method:" + joinPoint.getSignature());
8. System.out.println("Creating Employee with first name - " + fname + ", second name - " + sname + " and id - " + empId);
9. }
10. }

In the above class:

* **execution(expression):** The expression is a method on which advice is to be applied.
* **@Before:** It marks a function as an advice to be executed before method that covered by PointCut.

# **Spring Boot AOP After Advice**

After advice is used in Aspect-Oriented Programming to achieve the cross-cutting. It is an advice type which ensures that an advice runs after the method execution. We use **@After** annotation to implement the after advice.

1. @Aspect
2. @Component
3. public class EmployeeServiceAspect
4. {
5. @After(value = "execution(\* com.javatpoint.service.EmployeeService.\*(..)) and args(empId, fname, sname)")
6. public void afterAdvice(JoinPoint joinPoint, String empId, String fname, String sname) {
7. System.out.println("After method:" + joinPoint.getSignature());
8. System.out.println("Creating Employee with first name - " + fname + ", second name - " + sname + " and id - " + empId);
9. }
10. }

In the above class:

* **execution(expression):** The expression is a method on which advice is to be applied.
* **@After:** The method annotated with **@After** executes after all the methods that matched with the pointcut expression.

# **Spring Boot AOP Around Advice**

# Around advice is represented by **@Around** annotation. It executes before and after a join point. It is the most powerful advice. It also provides more control for end-user to get deal with **ProceedingJoinPoint.**

1. @Aspect
2. @Component
3. **public** **class** BankAspect
4. {
5. //Displays all the available methods i.e. the advice will be called for all the methods
6. @Pointcut(value= "execution(\* com.javatpoint.service.BankService.\*(..))")
7. **private** **void** logDisplayingBalance()
8. {
9. }
10. //Declares the around advice that is applied before and after the method matching with a pointcut expression
11. @Around(value= "logDisplayingBalance()")
12. **public** **void** aroundAdvice(ProceedingJoinPoint jp) **throws** Throwable
13. {
14. System.out.println("The method aroundAdvice() before invokation of the method " + jp.getSignature().getName() + " method");
15. **try**
16. {
17. jp.proceed();
18. }
19. **finally**
20. {
22. }
23. System.out.println("The method aroundAdvice() after invokation of the method " + jp.getSignature().getName() + " method");
24. }
25. }

In the following class, we have defined two methods :: named **logDisplayingBalance()** and **aroundAdvice()** method and proceed method is used for next advice to perform

# **Spring Boot AOP After Returning Advice**

**After returning** is an advice in Spring AOP that invokes after the execution of join point complete (execute) normally. It does not invoke if an exception is thrown. We can implement after returning advice in an application by using **@AfterReturning** annotation. The annotation marks a function as an advice to be executed before the method covered by PointCut.

After returning advice runs when a matched method execution returns a value normally. The name that we define in the return attribute must correspond to the name of a parameter in the advice method. When a method returns a value, the value will be passed to the advice method as the corresponding argument value.

1. @Aspect
2. @Component
3. **public** **class** AccountAspect
4. {
5. //implementing after returning advice
6. @AfterReturning(value="execution(\* com.javatpoint.service.impl.AccountServiceImpl.\*(..))",returning="account")
7. **public** **void** afterReturningAdvice(JoinPoint joinPoint, Account account)
8. {
9. System.out.println("After Returing method:"+joinPoint.getSignature());
10. System.out.println(account);
11. }
12. }

# **Spring Boot AOP After Throwing Advice**

After throwing is an advice type in Spring AOP. It ensures that an advice runs if a method throws an exception. We use **@AfterThrowing** annotation to implement the after throwing advice.

**Syntax:**

@AfterThrowing(PointCut="execution(expression) ", throwing="name")

Where:

**PointCut:** It selects a function.

**execution(expression):** It is an expression on which advice is to be applied.

**throwing:** The name of the exception to be returned.

1. @Aspect
2. @Component
3. **public** **class** AccountAspect
4. {
5. //implementing after throwing advice
6. @AfterThrowing(value="execution(\* com.javatpoint.service.impl.AccountServiceImpl.\*(..))",throwing="ex")
7. **public** **void** afterThrowingAdvice(JoinPoint joinPoint, Exception ex)
8. {
9. System.out.println("After Throwing exception in method:"+joinPoint.getSignature());
10. System.out.println("Exception is:"+ex.getMessage());
11. }
12. }

#### **Note: The name (ex) that we define in the throwing attribute must correspond to the name of a parameter in the advice method. Otherwise, advice will not run.**