Spring AOP - AspectJ Annotation

Last Updated : 11 Mar, 2022

AOP i.e **Aspect-Oriented Programming** complements OOP by enabling modularity of cross-cutting concerns. @AspectJ is mainly used for declaring aspects as regular Java classes annotated with annotations. Spring AspectJ AOP implementation has many annotations. They are explained below:

- **@Aspect**: It declares the class as an aspect.
- **@Pointcut**: It declares the pointcut expression.

Common AspectJ annotations used to create advice are as follows:

- **@Before**: It runs before the method execution.
- **@AfterReturning**: It runs after the result is returned by the method.
- **@AfterThrowing**: It runs after an exception is thrown by the method.
- **@After(Finally)**: It is executed after method execution or after an exception is thrown or the result is returned by the method.
- **@Around**: It can perform the behavior before and after the method invocation.
- **@Pointcut**: Pointcut is a signature that matches the join points.

@Pointcut

Example of pointcut expression with before advice is as follows:

```
// Annotation
@Before("execution(* abc.def.gettingstarted.dao.*.add(..))")
public void allMethods(Point Point)
{
  // Aspect body
}
```

@Before(Before Advice)

Before advice runs before execution of target method. Before advice can access all the arguments from the Target method. It cannot alter the values of the arguments to the original method. Before advice has to specify a Pointcut expression unless we use a separate @Pointcut expression provider.

Example: beforeAdvice() method annotated as before advice is executed before selectGetName() method is executed

```
@Aspect
public class Logging {

   @PointCut("execution(* com.gfg.Student.getName(..))")
   private void selectGetName(){}

   @Before("selectGetName()")
   public void beforeAdvice(){
      System.out.println("Going to setup student profile.");
   }

}
```

@AfterReturning(After Returning Advice)

After returning advice is a normal Java method, except that we have to add @AfterReturning annotation and a Pointcut expression. The Pointcut expression maps to the signature, class, and the package of the target method, and, the advice is applied to all such matching methods.

Example: In the below example methodCall method is executed when readFile method returns something.

```
@Slf4j
@Aspect
@Component
public class LoggingAspect {

    @AfterReturning("execution(*
com.gfg.spring.aop.service.FileSystemStorageService.readFile(..))")
```

```
    public void methodCall(JoinPoint jp, Exception ex) {
        log.error("Target method is successfully completed!!");
    }

}
```

@AfterThrowing(After Throwing Advice)

As the name suggests, AfterThrowing advice executes when the target method throws an exception. It will not run if the method finishes successfully, or the method swallows the thrown exception. Thus, the AfterThrowing advice is useful, when we want to build a common exception handler that can catch different exceptions thrown by a different target method and takes an action.

Example: In the below example, throwingAdvice method is executed when any method of the Student class throws an exception.

```
@AfterThrowing(PointCut = "execution(* com.gfg.Student.*(..))", throwing = "error")
public void throwingAdvice(JoinPoint jp, Throwable error){
   System.out.println("Method Signature: "  + jp.getSignature());
   System.out.println("Exception: "+error);
}
```

@After(After/Finally Advice)

An After advice runs after the target method is finished or includes when the target method results in an exception. This advice is similar to the final block in Java which always executes irrespective of the outcome.

Example: After advice is executed after method execution or after an exception is thrown or the result is returned by the method.

```
@After("execution(* com.gfg.saop.after.dao.*.*(..))")
public void logResult(JoinPoint joinPoint)
{
    System.out.println(
        "After advice executed: "
        + joinPoint.getSignature().toShortString());
}
```

@Around(Around Advice)

Spring AOP supports various types of advice and the around advice is one of those pieces of advice. This advice surrounds the target method execution. That means, on a target method execution, around advice runs and calls the target method. hence, the advice gets full control over applying additional code before and after method execution. The Around advice is very popular due to its flexibility. However, Spring recommends that we should use the advice type which is the most specific according to the requirement. It performs the behavior the same as before and after advice combined.