### AOP (Aspect Oriented Programming)

**Aspect Oriented Programming** is used in Spring framework **to allow the users to implement custom advices**.

**AOP** **provides** **the way to dynamically add the cross-cutting concerns before, after, or around the actual logic** **using** **simple pluggable configurations**.

**Advantages**:

- It makes **easy to maintain the code in the present and future as well**.

- you **can add/remove the concerns** **without recompiling complete source code** **simply by changing the configuration file**.

**AspectJ is** the industry standard implementation for AOP whereas Spring AOP for some cases.

**Spring AOP** supports AspectJ annotations.

**Spring AOP** supports only proxy based AOP, so it can be applied only to method executions joint point. AspectJ support all kinds of pointcut.

One of the shortcomings of Spring AOP is that it can be applied only to the beans created through Spring Context.

**Aspect**

- Aspect **is the cross-cutting concern which is applied for multiple methods in the application**.

- A **modularization of a** concerns **that cut across multiple classes**.

- you define a class as Aspect **using @Aspect annotation**.

**Joint Point**

- Joint point is **a specific point in the application** **on which the Aspect is applied**.

- Different Joint Points are method execution, exception handling, changing the value of the variable etc.

- **Spring AOP supports only the method execution to use a joint point**.

- **In Spring AOP, a joint point always represents the method execution**.

**Advice**

- Advice is **the action taken by an aspect** **at particular joint point.**

- In simple words, it is a **method which gets executed for the matching joint point and point cut**.

- Advice is **associated with point cut expressions** and **run at joint point matched by point cut**.

**Pointcut**

- Pointcut is **a predicate or expression that matches joint point**.

- Pointcut are **expressions that is matched with joint point** **to determine whether advice needs to be executed or not.**

- **pointcut expression which** determines exactly which **method we are applying the advice to.**

- syntax:

execution(access-specifier package-name class-name method-name(argument-list))

\* - to specify anything as the access specifier, we can use \*.

**Advice 5 types**

@Before, @After, @AfterReturning, @AfterThrowing, @Around

**@Before**

- @Before **advice that executes before the joint point methods execution**.

- @Before annotation **can be used to mark advice type** **as a Before advice**.

**@After**

- @After **advice that gets executed after the joint point methods execution**.

- **Even if the method throws any exception**, **after advice will be executed**.

- @After annotation **can be used to mark advice type** as a After advice.

**@AfterReturning**

- @AfterReturning **advice that gets executed after the joint point method executes normally.**

- @AfterReturning **advice will not be executed if the method throws any exception**.

- @AfterReturning annotation **can be used to mark advice type as** a After returning advice.

**@AfterThrowing**

- @AfterThrowing **advice that gets executed if the method throws an exception**.

- @AfterThrowing **advice will not be executed if the method executes normally**.

- @AfterThrowing **advice** **can be used if we want to roll back the transaction declaratively**.

- @AfterThrowing annotation can be used to mark advice type as a After Throwing advice.

**@Around**

- @Around **advice surrounds the joint point method** which means it **gets executed before and after the execution of the joint point methods**.

- @Around annotation can be used to mark advice type as Around advice.

**import** org.aspectj.lang.ProceedingJoinPoint;

**import** org.aspectj.lang.annotation.Around;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.springframework.stereotype.Component;

**@Aspect**

@Component

**public** **class** ExecutionTimeTrackerAdvice {

Logger logger = LoggerFactory.*getLogger*(ExecutionTimeTrackerAdvice.**class**);

@Around("@annotation(com.example.demo.config.aop.advice.TrackExecutionTime)")

**public** **Object** trackTime(ProceedingJoinPoint pjp) **throws** Throwable {

**long** startTime = System.*currentTimeMillis*();

Object obj = pjp.proceed();

**long** endTime = System.*currentTimeMillis*();

logger.info("classname:"+pjp.getClass()+" Method name :"+pjp.getSignature()+" time taken to execute :"+(endTime-startTime));

return obj;

}

}

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

//**Target annotation tells us where our annotation will be applicable.**

//**Here we are using ElementType.METHOD, which means it will only work on methods**.

@Target(ElementType.***METHOD***)

//Retention annotation states that **whether the annotation will be available to the JVM at runtime or not**.

//By default it is not

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** TrackExecutionTime {

}

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration

@ComponentScan("com.example.SpringAOP.AroundAdvice")

**@EnableAspectJAutoProxy**

**public** **class** AppConfig {

}

**import** org.springframework.stereotype.Component;

@Component

**public** **class** SimpleCalculator {

**public** **int** add(**int** x,**int** y) {

System.***out***.println("Total:"+ (x+y));

**return** x+y;

}

}

**import** org.aspectj.lang.ProceedingJoinPoint;

**import** org.aspectj.lang.annotation.Around;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.springframework.stereotype.Component;

@Aspect

@Component

**public** **class** MyAroundAspect {

//@Around("execution (\* com.example.SpringAOP.AroundAdvice.SimpleCalculator.\*(..))")

//@Around("execution (\* add(..))")

//@Around("execution (\* com.example..\*.\*(..))")

//@Around("execution (\* com.example..\*.add(..))")

//@Around("execution (\* com.example..SimpleCalculator.add(..))")

//@Around("execution (\* com.example..SimpleCalculator.\*(..))")

@Around("within (com.example..\*)")

**public** Object myAdvise(ProceedingJoinPoint pjp) **throws** Throwable {

System.***out***.println("Before Method runs");

Object obj = pjp.proceed();

System.***out***.println("After Method runs");

**return** obj;

}

}

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;

**public** **class** Application {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** AnnotationConfigApplicationContext(AppConfig.**class**);

SimpleCalculator calculator = context.getBean(SimpleCalculator.**class**);

calculator.add(1, 2);

}

}

**OUTPUT:**

Before Method runs

Total:3

After Method runs