Aspect Oriented Programming AOP

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

Have a Question?

Software University

sli.do

#java-web

What is AOP

Aspect Oriented Programming AOP (1)

- **AOP** breaks the program logic into distinct parts (called **concerns**)
- **Cross-cutting concern**
  - Concern that can affect the whole application and **should be centralized in one location**, such as transaction management, authentication, logging, security etc.

Why We Use AOP

Why Use AOP

- To **dynamically add the additional concern** before, after or around the actual logic
- Suppose that we have to maintain methods and needs to do actions before or after they are called
- We can solve the problem **with** or **without AOP**

Why Using AOP – Problem Example

- **Student class** with some methods whose activity we want to track

```
public class Student{
        public void actionOne(){...};
        public void actionTwo(){...};
        public void actionThree(){...};
        public void actionFour(){...};
        public void actionFive(){...};
}
```

Why Using AOP – Problem Solution

- **Solution without AOP**
  - If we need to log all activity of student, we need to write additional code in all tracked methods
  - It leads to the maintenance problem.
- **Solution with AOP**
  - We can define the additional concern like maintaining log, sending notification, etc. in the method of a class
  - Maintenance is easy in AOP

AOP Concepts and Terminology

Terminologies

- The AOP **concepts** and **terminologies** are
  - Join point
  - Advice
  - Pointcut
  - Introduction
  - Target Object
  - Aspect
  - Interceptor
  - AOP Proxy
  - Weaving

Join Point

- **Join point**
  - A Join point is **any point in your program** such as method execution, exception handling, field access etc.
  - We can have many Join points
  - Spring supports **only the method** execution join point

Advices and Types

- Represents an action taken by an aspect at a join point
  - **Before Advice**: it executes before a join point
  - **After Returning Advice**: it executes after a joint point completes normally
  - **After Throwing Advice**: it executes if method exits by throwing an exception
  - **After Advice**: it executes after a join point regardless of join point exit whether normally or exceptional return
  - **Around Advice**: It executes before and after a join point

Pointcut, Introduction, Target Object

- **Pointcut**
  - It is an expression language of AOP that matches join points
- **Introduction**
  - Introduction of additional method and fields for a type
- **Target Object**
  - The object i.e. being advised by one or more aspects
  - Also known as **Proxied Object**

Aspect, Interceptor, AOP Proxy, Weaving

- **Aspect**
  - A class that contains advices
- **Interceptor**
  - An aspect that contains only one advice
- **AOP Proxy**
  - Used to implement aspect contracts, created by AOP framework
- **Weaving**
  - The process of linking aspect with other application types or objects to create an advised object.

Spring AOP AspectJ Annotations

Spring AOP AspectJ (1)

- The 3 ways to use spring AOP are
  - By Spring 1.2 old style
  - By AspectJ annotation-style
    - The widely used approach is Spring AspectJ Annotation Style
  - By Spring XML configuration-style(schema based)

Spring AOP AspectJ (2)

- There are two ways to use Spring AOP AspectJ implementation
  - By annotation

```java
@Aspect
public class LoggingAspect {
    @Before("execution(* Student.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        ...
}}
```

  - By XML Configuration

```xml
<!-- Aspect -->
<bean id="logAspect" class="" />
<aop:config>
 <aop:aspect id="aspectLoggging" ref="logAspect" >
  <!-- @Before -->
  <aop:pointcut id="pointCutBefore"
    expression="execution(* Student.*(..))" />
  <aop:before method="logBefore" pointcut-ref="pointCutBefore" />
 </aop:aspect>
</aop:config>
```

AspectJ Annotations in Spring (1)

- **@Aspect**
  - Declares the class as aspect
- **@Pointcut**
  - Declares the pointcut expression
- **@Before**
  - Declares the before advice
  - Applied before calling the actual method

AspectJ Annotations in Spring (2)

- **@After**
  - Declares the after advice
  - Applied after calling the actual method and before returning result
- **@AfterReturning**
  - Declares the after returning advice
  - Applied after calling the actual method and before returning result, can get the result value in the advice

AspectJ Annotations in Spring (3)

- **@Around**
  - Declares the around advice
  - Applied before and after calling the actual method
- **@AfterThrowing**
  - Declares the throws advice
  - Applied if actual method throws exception

Pointcut

- Pointcut is an **expression language** of Spring AOP
- **@Pointcut** annotation is used to define the pointcut
- We can also **refer the pointcut expression by name**

```
@Pointcut("execution(public * *(..))")
private void trackStudentActions() {}
```

Pointcut Expressions

- Applied on all the public methods

```
@Pointcut("execution(public * *(..))")
```

- Applied on all methods of Student class

```
@Pointcut("execution(* Student.*(..))")
```

- Applied on all setter methods of Student class

```
@Pointcut("execution(* Student.set*(..))")
```

- Applied on all methods of class that returns an int value

```
@Pointcut("execution(int Student. *(..))")
```

Examples

Live Demonstration

Prepare for AOP Examples

- You remember from previous slides our Student class

```
public class Student {
      public void actionOne(){...};
      public void actionTwo(){...};
      public void actionThree(){...};
      public void actionFour(){...};
      public void actionFive(){...};
}
```

Create Aspect Class

- We need to create a class with **@Aspect**, that contains all advices

```
@Aspect
@Configuration
public class TrackStudent{
      @Pointcut("execution(* Student.*(..))")
      public track(){}

      //Can have more than one pointcuts
      //Here place all advices
}
```

@Before Example

- Add **before advice** to our TrackStudent class

```
@Aspect
@Configuration
public class TrackStudent {
      @Pointcut("execution(* Student.*(..))")
      public track(){}
      @Before("track()") // Execute before track pointcut
      public void beforeAdvice(JoinPoint joinPoint){
            System.out.println("Before advice executed");
}}
```

@After Example

- Add after advice to our TrackStudent class

```
@Aspect
@Configuration
public class TrackStudent {
      @Pointcut("execution(* Student.*(..))")
      public track(){}
      @After("track()") // Execute after track pointcut
      public void afterAdvice(JoinPoint joinPoint){
            System.out.println("After advice executed");
} }
```

@AfterReturning Example

- Add after returning advice to our TrackStudent class

```
...
@AfterReturning
(pointcut="execution(* Student.action())",returning="result")
public void afterReturning(JoinPoint joinPoint,
                                         Object result){
      System.out.println("AfterReturning advice executed");
      //In AfterReturning we can get the result of pointcut
}
...
```

@Around Example

- Add **around advice** to our TrackStudent class

```
@Around("track()")
public Object aroundAdvices(ProceedingJoinPoint pjp)
                                            throws Throwable {
       System.out.println("Before calling");
       Object obj = pjp.proceed();
       //We need to pass the pjp references in the advice
              method, so that we can proceed
              the request by calling the proceed method
       System.out.println("After calling");
}
...
```

@AfterThrowing Example

- Add after throwing advice to our TrackStudent class

```
...
@AfterThrowing
(pointcut="execution(* Student.action())",throwing="error")
Public void afterReturning(JoinPoint joinPoint,
                                 Throwable error){
      System.out.println("AfterReturning advice executed");
      System.out.println("Exception is: " + error);
      //In AfterThrowing we can get the exception
}
...
```

Specifying Aspects Ordering

- There are two ways:
  - By annotation

```
@Aspect
@Order(0)
public class TrackStudent{//...}
```

  - By implementing interface

```
@Aspect
public class TrackStudent implements Ordered {
        //Override this method
        public int getOrder(){ return 0; }
}
```

Summary

- **AOP – Aspect Oriented Programming**
  - Breaks the program logic into distinct parts (called **concerns**)
  - Maintenance is **easy** in **AOP**
- **Spring AOP AspectJ Annotation**
  - The widely used approach is Spring AspectJ Annotation Style

Software University

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

SoftUni Diamond Partners

Software University

DraftKings

INDEAVR
Serving the high achievers

Coca-Cola HBC Bulgaria

SCHWARZ

LIDL

Kaufland

SCHWARZ Produktion

prezero

Postbank
Решения за твоето утре

POKERSTARS | CAREERS

SUPER HOSTING .BG

SOFTWARE GROUP

SmartIT

PHAR VISION

createX

taulia®

Educational Partners

CODE IT UP

VIRTUAL RACING SCHOOL

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License