Bài 3

Spring AOP - AspectJ

Aspect Oriented Programming with Spring

Aspect-oriented Programming (AOP) complements Object-oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Aspects enable the modularization of concerns (such as transaction management) that cut across multiple types and objects. (Such concerns are often termed "crosscutting" concerns in AOP literature.)

One of the key components of Spring is the AOP framework. While the Spring IoC container does not depend on AOP (meaning you do not need to use AOP if you don't want to), AOP complements Spring IoC to provide a very capable middleware solution.

Spring AOP with AspectJ pointcuts

Spring provides simple and powerful ways of writing custom aspects by using either a schema-based approach or the @AspectJ annotation style. Both of these styles offer fully typed advice and use of the AspectJ pointcut language while still using Spring AOP for weaving.

This chapter discusses the schema- and @AspectJ-based AOP support. The lower-level AOP support is discussed in the following chapter.

AOP is used in the Spring Framework to:

- Provide declarative enterprise services. The most important such service is declarative transaction management.
- Let users implement custom aspects, complementing their use of OOP with AOP.

AOP Concepts

- Aspect: A modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented by using regular classes (the schema-based approach) or regular classes annotated with the `@Aspect` annotation (the @AspectJ style).
- Join point: A point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.
- Advice: Action taken by an aspect at a particular join point. Different types of advice include “around”, “before” and “after” advice. (Advice types are discussed later.) Many AOP frameworks, including Spring, model an advice as an interceptor and maintain a chain of interceptors around the join point.
- Pointcut: A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.
- Introduction: Declaring additional methods or fields on behalf of a type. Spring AOP lets you introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an `IsModified` interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)
- Target object: An object being advised by one or more aspects. Also referred to as the “advised object”. Since Spring AOP is implemented by using runtime proxies, this object is always a proxied object.
- AOP proxy: An object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy is a JDK dynamic proxy or a CGLIB proxy.

AOP Concepts

Spring AOP includes the following types of advice:

- Before advice: Advice that runs before a join point but that does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
- After returning advice: Advice to be run after a join point completes normally (for example, if a method returns without throwing an exception).
- After throwing advice: Advice to be run if a method exits by throwing an exception.
- After (finally) advice: Advice to be run regardless of the means by which a join point exits (normal or exceptional return).
- Around advice: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

AOP Proxy

AOP Proxy

Spring AOP defaults to using standard JDK dynamic proxies for AOP proxies. This enables any interface (or set of interfaces) to be proxied.

Spring AOP can also use CGLIB proxies. This is necessary to proxy classes rather than interfaces. By default, CGLIB is used if a business object does not implement an interface. As it is good practice to program to interfaces rather than classes, business classes normally implement one or more business interfaces. It is possible to force the use of CGLIB, in those (hopefully rare) cases where you need to advise a method that is not declared on an interface or where you need to pass a proxied object to a method as a concrete type.

It is important to grasp the fact that Spring AOP is proxy-based. See Understanding AOP Proxies for a thorough examination of exactly what this implementation detail actually means.

```
<bean id="movieServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">

    <property name="target" ref="movieService" />
    <property name="interceptorNames">
        <list>
            <!-- <value>around</value> -->
             <value>movieAdvisor</value>
        </list>
    </property>
</bean>
```

AOP Proxy

```
<!-- Advice -->
<bean id="around" class="home.aop.MovieTrackerAroundMethod" />

<!-- Point Cut By Name -->
<bean id="moviePointcut"
    class="org.springframework.aop.support.NameMatchMethodPointcut">
    <!-- Math the method name -->
    <property name="mappedName" value="printName" />
</bean>

<!-- Advisor: Group of Advice and Point Cut -->
<bean id="movieAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="pointcut" ref="moviePointcut" />
    <property name="advice" ref="around" />
</bean>

<!-- Advisor: Point Cut Regular Expression -->
<!-- <bean id="movieAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="patterns">
        <list>
            <value>home.*</value>
        </list>
    </property>
    <property name="advice" ref="around" />
</bean> -->

<bean id="movieServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">

    <property name="target" ref="movieService" />
    <property name="interceptorNames">
        <list>
            <!-- <value>around</value> -->
             <value>movieAdvisor</value>
        </list>
    </property>
</bean>
```

AOP Proxy
- Join Point
- Advisior
  - Advice
  - PointCut

AOP Proxy

```
<!-- Auto Proxy more powerful -->
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" />

<!-- Get proxy via bean name directly >> movieService -->
<!-- <bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">

    <property name="beanNames">
        <list>
            <value>*Service</value>
        </list>
    </property>
    <property name="interceptorNames">
        <list>
          <value>around</value>
            <value>movieAdvisor</value>
        </list>
    </property>
</bean> -->
```

AOP Auto Proxy

==========

<aop:aspectj-autoproxy />

==========

@EnableAspectJAutoProxy

Live Demo with AOP

Live demo with AOP

Step By Step

1. Create Join Point Movie Service with 2 Join Point methods
2. Create Advice
    1. Before Advice
    2. After – No Throw Advice
    3. After – Throw Advice – Removed from Spring 5
    4. Around Advice
3. Create Aspect configuration with XML
    1. Option 1
        1. Join Point
        2. Advice
        3. Proxy
    2. Option 2
        1. Join Point
        2. Advisor
            1. Advice
            2. Point Cut
        3. Proxy
    3. Option 3
        1. Auto Proxy
4. Demo with ApplicationContext

Live demo with AOP

Step 1: Create Join Point Movie Service with 2 Join Point methods

```java
package home.service;

// Join Point Bean
public class MovieService {
    private String name;
    private String catalog;

    public void setName(String name) {
        this.name = name;
    }

    public void setCatalog(String catalog) {
        this.catalog = catalog;
    }

    // Join Point methods
    public void printName() {
        System.out.println("MovieService >> " + name);
    }

    public void printCatalog() {
        System.out.println("MovieService >> " + catalog);
    }

    public void throwException() {
        throw new NullPointerException("MovieService >> message ");
    }
}
```

Live demo with AOP

Step 2: Create Advice

Before Advice

```
public class MovieTrackerBeforeMethod implements MethodBeforeAdvice {

    @Override
    public void before(Method method, Object[] args, Object target) throws Throwable {
        System.out.println("MovieTrackerBeforeMethod >> before ...");
    }
}
```

AfterReturning Advice

```
public class MovieTrackerAfterMethod implements AfterReturningAdvice {

    @Override
    public void afterReturning(Object returnValue, Method method,
            Object[] args, Object target) throws Throwable {
        System.out.println("MovieTrackerAfterMethod >> afterReturning ...");
    }
}
```

Live demo with AOP

Step 2: Create Advice

Around Advice

```
public class MovieTrackerAroundMethod implements MethodInterceptor {
    @Override
    public Object invoke(MethodInvocation methodInvocation) throws Throwable {

        System.out.println("Method name : "
                + methodInvocation.getMethod().getName());
        System.out.println("Method arguments : "
                + Arrays.toString(methodInvocation.getArguments()));

        System.out.println("===== Before proceed ====");

        try {
            Object result = methodInvocation.proceed();
            System.out.println("===== After proceed ====");
            return result;

        } catch (IllegalArgumentException e) {
            System.out.println("==== Throw Exception ====");
            throw e;
        }
    }
}
```

Live demo with AOP

Step 3: Create Aspect with XML configuration

Option 1

```
<bean id="movieService" class="home.service.MovieService">
    <property name="name" value="Avatar" />
    <property name="catalog" value="Adventure" />
</bean>

<!--
<bean id="before" class="home.aop.MovieTrackerBeforeMethod" />
<bean id="after" class="home.aop.MovieTrackerAfterMethod" />
<bean id="throw" class="home.aop.MovieTrackerThrowException" />
-->
<bean id="around" class="home.aop.MovieTrackerAroundMethod" />

<bean id="movieServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">

    <property name="target" ref="movieService" />
    <property name="interceptorNames">
        <list>
            <!-- <value>before</value>
            <value>after</value>
            <value>throw</value> -->

            <value>around</value>
        </list>
    </property>
</bean>
```

Live demo with AOP

Step 3: Create Aspect with XML configuration

Option 2

```
<!-- Bean - Join Point  -->
<bean id="movieService" class="home.service.MovieService">
    <property name="name" value="Avatar" />
    <property name="catalog" value="Adventure" />
</bean>

<!-- Advice  -->
<bean id="around" class="home.aop.MovieTrackerAroundMethod" />

<!-- Point Cut By Name -->
<bean id="moviePointcut"
    class="org.springframework.aop.support.NameMatchMethodPo
    <!-- Math the method name -->
    <property name="mappedName" value="printName" />
</bean>

<!-- Advisor: Group of Advice and Point Cut -->
<bean id="movieAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="pointcut" ref="moviePointcut" />
    <property name="advice" ref="around" />
</bean>

<!-- Advisor: Point Cut Regular Expression -->
<!-- <bean id="movieAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="patterns">
        <list>
            <value>home.*</value>
        </list>
    </property>
    <property name="advice" ref="around" />
</bean> -->
```

```
<bean id="movieServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">

    <property name="target" ref="movieService" />
    <property name="interceptorNames">
        <list>
            <value>movieAdvisor</value>
        </list>
    </property>
</bean>
```

Live demo with AOP

Step 3: Create Aspect with XML configuration

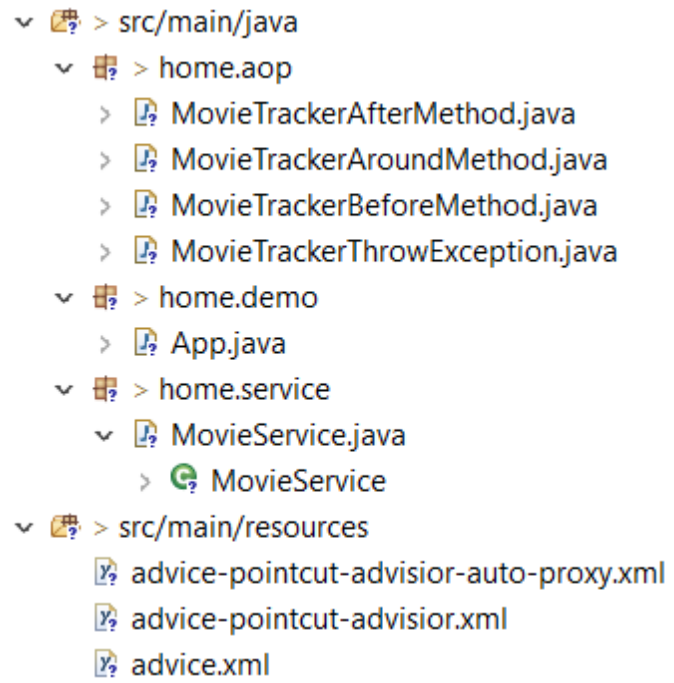
Option 3

```
<!-- Bean - Join Point  -->
<bean id="movieService" class="home.service.MovieService">
    <property name="name" value="Avatar" />
    <property name="catalog" value="Adventure" />
</bean>

<!-- Advice  -->
<bean id="around" class="home.aop.MovieTrackerAroundMethod" />

<!-- Point Cut By Name -->
<bean id="moviePointcut"
    class="org.springframework.aop.support.NameMatchMethodPointcut">
    <!-- Math the method name -->
    <property name="mappedName" value="printName" />
</bean>

<!-- Advisor: Group of Advice and Point Cut -->
<!-- If any of the beans is matched by an advisor, Spring will create a proxy for it automatically. -->
<bean id="movieAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="pointcut" ref="moviePointcut" />
    <property name="advice" ref="around" />
</bean>

<!-- Auto Proxy more powerful -->
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" />
```

Live demo with AOP

Step 4: Demo with ApplicationContext

```
private static final String XML = "advice-pointcut-advisior-auto-proxy.xml";

public static void main(String[] args) {
    ConfigurableApplicationContext context = new ClassPathXmlApplicationContext(XML);

    // MovieService movieService = context.getBean("movieServiceProxy", MovieService.class);
    MovieService movieService = context.getBean("movieService", MovieService.class);

    System.out.println("======================");
    movieService.printName();

    System.out.println("\n======================");
    movieService.printCatalog();

    context.close();

}
```



AspectJ support

AspectJ Support

To use @AspectJ aspects in a Spring configuration, you need to enable Spring support for configuring Spring AOP based on @AspectJ aspects and auto-proxying beans based on whether or not they are advised by those aspects. By auto-proxying, we mean that, if Spring determines that a bean is advised by one or more aspects, it automatically generates a proxy for that bean to intercept method invocations and ensures that advice is run as needed.

The @AspectJ support can be enabled with XML- or Java-style configuration. In either case, you also need to ensure that AspectJ's `aspectjweaver.jar` library is on the classpath of your application (version 1.8 or later). This library is available in the `lib` directory of an AspectJ distribution or from the Maven Central repository.

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.6.11</version>
</dependency>

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.6.11</version>
</dependency>
```

AOP Proxy
Join Point
Advisior
Advice
PointCut

AspectJ Support

Enabling @AspectJ Support with Java Configuration

To enable @AspectJ support with Java `@Configuration`, add the `@EnableAspectJAutoProxy` annotation, as the following example shows:

Java | Kotlin

JAVA

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {

}
```

Enabling @AspectJ Support with XML Configuration

To enable @AspectJ support with XML-based configuration, use the `aop:aspectj-autoproxy` element, as the following example shows:

```
<aop:aspectj-autoproxy/>
```

AOP Proxy
Join Point
Advisior
Advice
PointCut

AspectJ Support

```java
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {

    @Bean
    public MovieService movieService() {
        return new MovieServiceImpl();
    }

    @Bean
    public MovieLogAspect movieLogAspect() {
        return new MovieLogAspect();
    }

}
```

```xml
<aop:aspectj-autoproxy />

<!-- Joint Point  -->
<bean id="movieService" class="home.service.MovieServiceImpl" />

<!-- Aspect contains advice, point cut-->
<bean id="movieLogAspect" class="home.aspect.MovieLogAspect" />
```

AspectJ declaration

With @AspectJ support enabled, any bean defined in your application context with a class that is an @AspectJ aspect (has the `@Aspect` annotation) is automatically detected by Spring and used to configure Spring AOP. The next two examples show the minimal definition required for a not-very-useful aspect.

The first of the two example shows a regular bean definition in the application context that points to a bean class that has the `@Aspect` annotation:

```xml
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
    <!-- configure properties of the aspect here -->
</bean>
```

The second of the two examples shows the `NotVeryUsefulAspect` class definition, which is annotated with the `org.aspectj.lang.annotation.Aspect` annotation;

Java | **Kotlin**

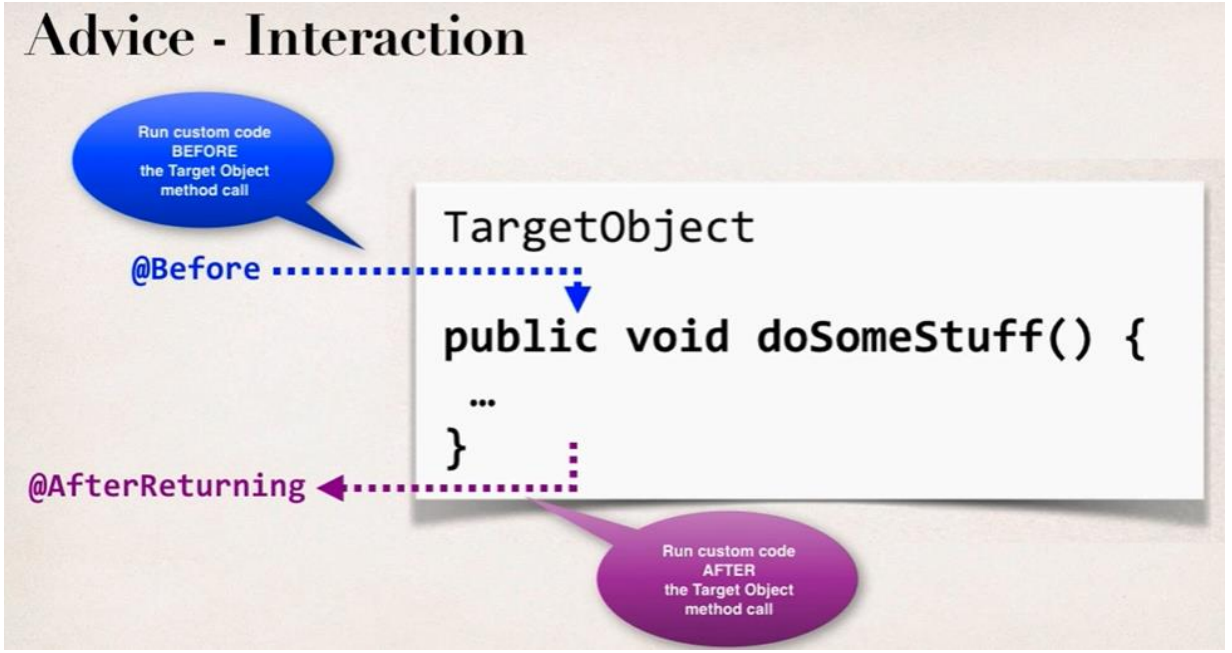
```java
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

Aspects (classes annotated with `@Aspect`) can have methods and fields, the same as any other class. They can also contain pointcut, advice, and introduction (inter-type) declarations.

Advice declaration

Advice Types

Before Advice: run before the method
After Returning Advice: run after the method(success execution)
After Throwing Advice: run after the method(if exception thrown)
After Finally Advice: run after the method(finally)
Around Advice: run before and after method



Advice declaration

You can declare before advice in an aspect by using the `@Before` annotation:

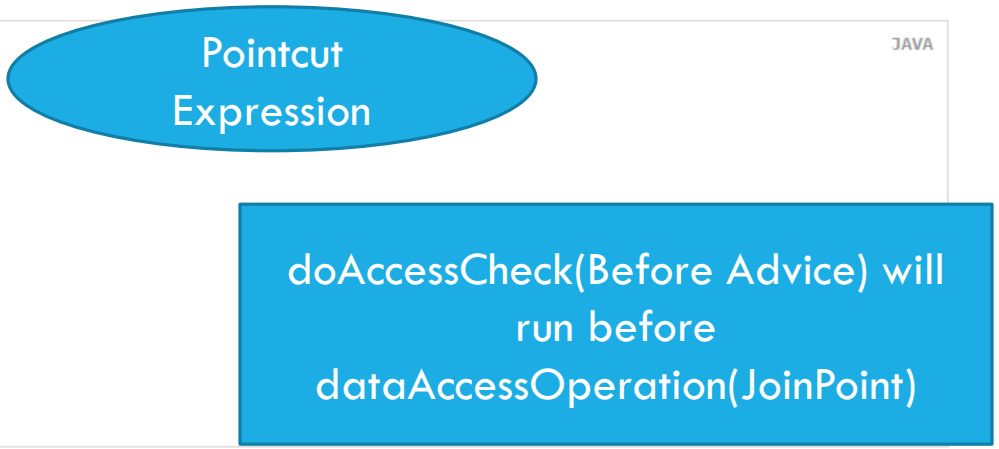
Java | Kotlin

```java
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

    @Before("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    public void doAccessCheck() {
        // ...
    }
}
```

If we use an in-place pointcut expression, we could rewrite the preceding example as the following example:

Java | Kotlin

```java
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

    @Before("execution(* com.xyz.myapp.dao.*.*(..))")
    public void doAccessCheck() {
        // ...
    }
}
```



Advice declaration

After returning advice runs when a matched method execution returns normally. You can declare it by using the `@AfterReturning` annotation:

Java | **Kotlin**

JAVA

```java
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

    @AfterReturning("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    public void doAccessCheck() {
        // ...
    }
}
```

PointCut Expression – Named Match and Return Type

- The execution of any method with a name that begins with `set`:

```
execution(* set*(..))
```

- The execution of any method defined by the `AccountService` interface:

```
execution(* com.xyz.service.AccountService.*(..))
```

- The execution of any method defined in the `service` package:

```
execution(* com.xyz.service.*.*(..))
```

- The execution of any method defined in the service package or one of its sub-packages:

```
execution(* com.xyz.service..*.*(..))
```

- Any join point (method execution only in Spring AOP) within the service package:

```
within(com.xyz.service.*)
```

- Any join point (method execution only in Spring AOP) within the service package or one of its sub-packages:

```
within(com.xyz.service..*)
```

- Any join point (method execution only in Spring AOP) where the proxy implements the `AccountService` interface:

```
this(com.xyz.service.AccountService)
```

PointCut Expression – Named Match and Return Type

```
execution(modifiers-pattern? return-type-pattern declaring-type-pattern?
        method-name-pattern(param-pattern) throws-pattern?)
```

Parameter Pattern Wildcards

- For param-pattern
  - **()** - matches a method with no arguments
  - **(*)** - matches a method with one argument of any type
  - **(..)** - matches a method with 0 or more arguments of any type

```
@Before("execution(* add*(com.spring.bean.Movie, ..))")
```

PointCut Expression

Problem
How can we reuse a pointcut expression
Want to apply to multiple advices


```
@Before("execution(* home.service.MovieService.*(..))")
public void before(JoinPoint joinPoint) {
    // ...
}

@Before("execution(* home.service.MovieService.*(..))")
public void analyse(JoinPoint joinPoint) {
    // ...
}
```

PointCut Expression

Problem
How can we reuse a pointcut expression
Want to apply to multiple advices

```
@Pointcut("execution(* home.service.MovieService.*(..))")
public void forMoviePointCut() {}


@Before("forMoviePointCut()")
public void before(JoinPoint joinPoint) {}

@Before("forMoviePointCut()")
public void analyse(JoinPoint joinPoint) {}
```

Advice declaration

Join Point

```
public interface MovieService {
    void addMovie();
    String getMovie();
    void validateMovie() throws Exception;
    void updateMovie(String name);
}
```

Advice declaration

```java
public interface MovieService {
    void addMovie();
    String getMovie();
    void validateMovie() throws Exception;
    void updateMovie(String name);
}
```

```java
@Before("execution(* home.service.MovieService.addMovie(..))")
public void before(JoinPoint joinPoint) {
    System.out.println("MovieLogAspect >> before >> is running!");
    System.out.println("signature : " + joinPoint.getSignature().getName());
    System.out.println("========================================");
}

@After("execution(* home.service.MovieService.addMovie(..))")
public void after(JoinPoint joinPoint) {
    System.out.println("========================================");
    System.out.println("MovieLogAspect >> after >> is running!");
    System.out.println("signature : " + joinPoint.getSignature().getName());
}
```

Advice declaration

```java
public interface MovieService {
    void addMovie();
    String getMovie();
    void validateMovie() throws Exception;
    void updateMovie(String name);
}
```

```java
@AfterReturning(
        pointcut = "execution(* home.service.MovieService.getMovie(..))",
        returning = "result")
public void logAfterReturning(JoinPoint joinPoint, Object result) {
    System.out.println("========================================");
    System.out.println("MovieLogAspect >> logAfterReturning >> is running!");
    System.out.println("signature : " + joinPoint.getSignature().getName());
    System.out.println("Returned value : " + result);
}

@AfterThrowing(
        pointcut = "execution(* home.service.MovieService.validateMovie(..))",
        throwing = "error")
public void logAfterThrowing(JoinPoint joinPoint, Throwable error) {
    System.out.println("========================================");
    System.out.println("MovieLogAspect >> logAfterThrowing >> is running!");
    System.out.println("signature : " + joinPoint.getSignature().getName());
    System.out.println("Exception : " + error);
}
```

Advice declaration

```java
public interface MovieService {
    void addMovie();
    String getMovie();
    void validateMovie() throws Exception;
    void updateMovie(String name);
}
```

```java
@Around("execution(* home.service.MovieService.updateMovie(..))")
public void logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    System.out.println("MovieLogAspect >> logAround >> is running!");
    System.out.println("Method name : " + joinPoint.getSignature().getName());
    System.out.println("Method arguments : " + Arrays.toString(joinPoint.getArgs()));

    System.out.println("Around before is running!");
    joinPoint.proceed(); // execute method
    System.out.println("Around after is running!");
}
```

Live Demo with AspectJ

Live demo with AspectJ

Step By Step

1. Create Join Point Movie Service with 4 Join Point methods
2. Create MovieLogAspect with @Aspect
    1. @Before
    2. @After
    3. @AfterReturning
    4. @AfterThrowing
    5. @Around
3. Create configuration
    1. AspectConfig(JoinPoint, LogAspect)
    2. AppConfig
4. Demo with ApplicationContext

Live demo with AOP

Step 1: Create Join Point Movie Service with 4 Join Point methods

```
public interface MovieService {
    void addMovie();
    void updateMovie(String name);
    void validateMovie() throws Exception;
    String getMovie();
}
```

Live demo with AOP

Step 2: Create MovieLogAspect with @Aspect

```
@Aspect
public class MovieLogAspect {

    @Before("execution(* home.service.MovieService.addMovie(..))")
    public void before(JoinPoint joinPoint) {
        System.out.println("MovieLogAspect >> before >> is running!");
        System.out.println("signature : " + joinPoint.getSignature().getName());
        System.out.println("========================================");
    }

    @After("execution(* home.service.MovieService.addMovie(..))")
    public void after(JoinPoint joinPoint) {
        System.out.println("========================================");
        System.out.println("MovieLogAspect >> after >> is running!");
        System.out.println("signature : " + joinPoint.getSignature().getName());
    }

    @AfterReturning(pointcut = "execution(* home.service.MovieService.getMovie(..))", returning = "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        System.out.println("========================================");
        System.out.println("MovieLogAspect >> logAfterReturning >> is running!");
        System.out.println("signature : " + joinPoint.getSignature().getName());
        System.out.println("Returned value : " + result);
    }
```

Live demo with AOP

Step 2: Create MovieLogAspect with @Aspect

```
@AfterThrowing(pointcut = "execution(* home.service.MovieService.validateMovie(..))", throwing = "error")
public void logAfterThrowing(JoinPoint joinPoint, Throwable error) {
    System.out.println("========================================");
    System.out.println("MovieLogAspect >> logAfterThrowing >> is running!");
    System.out.println("signature : " + joinPoint.getSignature().getName());
    System.out.println("Exception : " + error);
}

@Around("execution(* home.service.MovieService.updateMovie(..))")
public void logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    System.out.println("MovieLogAspect >> logAround >> is running!");
    System.out.println("Method name : " + joinPoint.getSignature().getName());
    System.out.println("Method arguments : " + Arrays.toString(joinPoint.getArgs()));

    System.out.println("Around before is running!");
    joinPoint.proceed(); // execute method
    System.out.println("Around after is running!");
}
```

Live demo with AOP

Step 3: Create Configuration

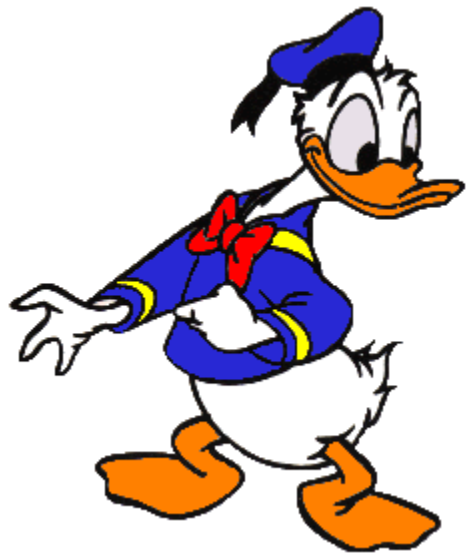
```
@Configuration
public class AspectConfig {

    @Bean
    public MovieService movieService() {
        return new MovieServiceImpl();
    }

    @Bean
    public MovieLogAspect movieLogAspect() {
        return new MovieLogAspect();
    }
}
```

```
@Configuration
@EnableAspectJAutoProxy
@Import(value = AspectConfig.class)
public class AppConfig {

}
```

Live demo with AOP

Step 4: Demo with ApplicationContext

```
public class App {
    public static void main(String[] args) throws Exception {
        // ConfigurableApplicationContext context = new ClassPathXmlApplicationContext("aspectj.xml");
        ConfigurableApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        MovieService service = context.getBean("movieService", MovieService.class);
        service.addMovie();
        // service.getMovie();
        // service.validateMovie();
        // service.updateMovie("Avatar");

        context.close();
    }
}
```



END