2. Spring AOP

- `횡단 관심사(Cross-Cutting Concern)` :
  여러 개의 모듈에 걸치는 시스템 전체적으로 부가해야 하는 요구사항을 다룬다. 대표적인 횡단 관심사의 일부 예들 든다면 인증 ,로깅,트랜잭션 무결성, 오류검사, 정책시행 등이 있다.

- `핵심 관심사(Primary Concern)` :
  각 모듈에서 수행해야 하는 기본적이고 대표적인 업무처리 기능을 다룬다.

IoC를 이용하여 협력하는 객체를 쉽게 연결시키는 것이 가능하지만 때때로 전체 애플리케이션에 걸쳐 사용되어야 하는 `공통적인 기능(횡단 관심사)`이 필요할 수도 있다.

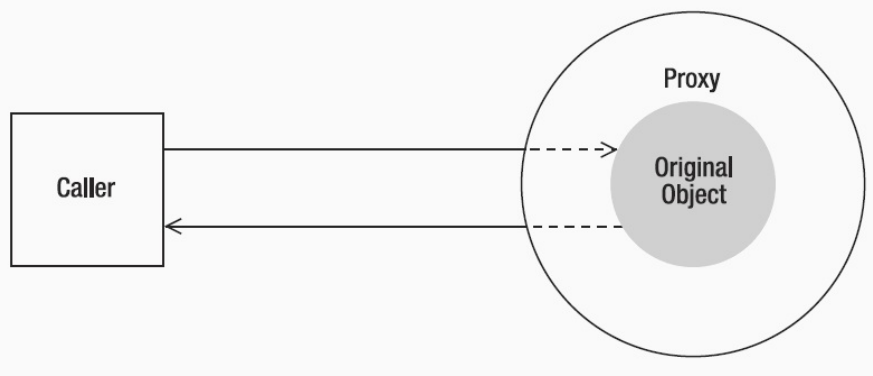
DI의 목적이 객체레벨에서의 결합도를 떨어뜨리는 것이 목표였다면 AOP의 목적은 횡단관심사와 이에 영향을 받는 객체들의 메소드 레벨에서의 결합도를 떨어뜨리는 것이다. 즉, AOP는 횡단관심사를 모듈화하는 프로그래밍 기법으로 공통적인 기능을 Aspect라 불리는 분리된 곳에 정의한다.

새로운 기능을 적용하려고 하는 클래스를 수정할 필요 없이 그런 기능을 어디에 어떻게 적용할 것인지 선언적 또는 애노테이션으로 설정하여 적용할 수 있다. 각 횡단관심사에 관한 로직이 애플리케이션 전체에 걸쳐 관리되지 않고 Aspect라고 하는 곳에서 분리되어 관리되면 애플리케이션은 고유의 비지니스로직 처리에 보다 더 충실할 수 있어서 관리성이 증대된다.

예를 들어 로깅처리는 시스템 내에서 거의 모든 경우에 적용하긴 하지만 특정한 비즈니스 로직과는 전혀 관련이 없다. 만일 AOP방식으로 구현한다면 실제 비즈니즈 로직의 구현을 담당하는 객체는 로그와 관련된 로직 정보가 없고 이 객체의 특정 메소드를 호출하는 시점에 지정된 다른 로깅 서비스 로직이 실행된다.

Spring AOP 구현 방법

- 소스코드에서 직접 AOP 구현 : ProxyFactory 기반 AOP
- 선언(XML, 애노테이션)적인 AOP 설정 메커니즘
    - `ProxyFactoryBean` 사용 : 스프링 ApplicationContext를 XML에 선언적으로 선언하여 빈 정의를 기반으로 AOP 프록시를 생성한다.
    - `aop 네임스페이스` : aop 네임스페이스를 이용하여 Aspect 및 DI 요구사항을 간단히 정의. aop 네임스페이스도 내부적으로 ProxyFactoryBean을 사용한다.
    - `@ApectJ Annotation` : `@AspectJ` 방식의 애노테이션을 사용하여 클래스 내에서 AOP 설정이 가능하다, 이 방식은 AspectJ를 기반으로 하고 있으며 AspectJ 라이브러리가 필요하다. 이 역시 스프링에서는 프록시 메커니즘을 이용하는데 ApplicationContext를 부트스트랩 할 때 타겟에 대해 프록시를 생성한다.
- Spring AOP는 메소드 가로채기로 제한한다. 만약 그이상의 기능
  (생성자 또는 멤버변수에 대한 가로채기)을 필요로 하는 경우 Proxy 기반 AOP
  대신 AspectJ를 이용해서 Aspect를 구현해야 한다.
- 스프링의 충고(Advice)는 자바로 작성 : Pointcut의 경우 Advice를 어디에 적용할지를 정의하는 데 XML 설정 파일, AspectJ Expression으로 정의한다. AspectJ는 자바언어를 확장한 형태로 구현되면 이를 사용하기 위해서 새로운 도구와 문법을 배워야 한다.
- 스프링의 Aspect는 실행시간에 만들어진다. 빈을 감싸는 Proxy 객체를 실행시간에 생성하므로 Aspect가 Spring 관련 빈에 위빙(Weaving)된다. Proxy 객체는 타겟 객체로 위장해서 메소드 호출을 가로채고, 타겟 객체로 호출을 전달한다. 어플리케이션이 실제 Proxy 객체를 필요로 할 때까지 target를 생성하지 않으므로 즉 Proxy가 실시간으로 생성되므로 Aspect를 위빙 하기 위해 별도 컴파일러가 필요 없다.



- 스프링은 AOP연맹의 인터페이스를 구현하며 메소드 호출 결합점(Join Point)만 제공한다.

스프링 AOP 용어

결합점(Join point)	무수히 많은 Advice를 적용할 수 있는 지점(Aspect를 플러그인 할 수 있는 애플리케이션의 실행 지점). **타겟 클래스가 구현한 인터페이스의 모든 메소드가 조인 포인트가 된다.**
교차점(pointcut)	**어드바이스를 적용할 조인 포인트를 선별하는 작업 또는 그 기능을 정의한 모듈.** Advice가 위빙(Weaving)되어야 할 하나 이상의 Join Point. 명시적인 클래스의 이름, 메소드의 이름이나 클래스나 메소드의 이름과 패턴이 일치하는 Join Point를 지정 가능토록 해준다.
충고(Advice)	타겟 클래스에 제공할 **횡단 관심사 기능을 구현한 것**. 특정 조인포인트 서 실행되는 코드. Aspect의 실제 구현체로 Aspect가 해야할 작업, Aspect가 무엇을 언제할지 정의함
에스팩트(Aspect)	**Aspect는 AOP의 기본모듈. Advice와 pointcut을 합친 것이고** 구현 하고자 하는 횡단 관심사의 기능을 모듈화 한것. 애플리케이션에 포함시킬 로직의 정의와 이 로직이 실행되는 위치를 정한다.
대상(target)	**충고를 받는 클래스**를 대상(target)라고 한다. 대상은 여러분이 작성한 클래스는 물론, 별도의 기능을 추가하고자 하는 써드파티 클래스가 될 수 있다.
위빙(Weaving)	에스펙트를 대상 객체에 적용하여 **새로운 프록시 객체를 생성하는 과정**을 말한다.
프록시(Proxy)	**Advice를 target 객체에 적용하는 생기는 객체**. 클라이언트 객체 관점에서 target 객체(AOP 적용 전)와 proxy 객체(AOP 적용 후)는 차이가 없다. 스프링이 런타임 중 동적으로 생성하므로 프록시를 위한 별도의 컴파일러는 필요없다.

스프링이 Proxy 객체를 생성하는 2가지 방법

1. 대상 객체가 특정 메소드를 공개하는 인터페이스를 구현한다면 JDK의 java.lang.reflect.Proxy 클래스를 이용하며 이 클래스는 필요한 인터페이스를 구현한 새로운 Proxy 객체를 동적으로 생성할 수 있으며 target 객체의 인터페이스를 통한 호출은 모두 가로채서 Advice를 수행한다.

2. 대상 클래스가 어떤 인터페이스를 구현하고 있지 않다면 CGLIB이라는 라이브러리를 이용하여 대상클래스의 서브클래스를 생성 시킨다. 이 서브클래스를 생성시킴으로써 스프링은 충고를 엮을 수 있으며 서브클래스에 대한 호출을 대상 클래스에 위임 할 수 있다. 이 경우 final 메소드는 충고를 받을 수 없으며(final Method는 대상클래스의 서브클래스를 생성해서 메소드를 재정의 해야 하는데 final인 경우 곤란함) 애플리케이션이 좀 더 느슨하게 결합되게 하기 위해 이 방법보단 인터페이스를 통해 프록시를 만드는 것이 선호된다.

- 스프링 어스팩트 : 스프링 AOP에서 애스팩트는 Advisor 인터페이스를 구현한 클래스의 인스턴스로 표시하는데 Advisor를 구현한 몇 개의 클래스를 제공한다. Advisor의 하위 인터페이스로는 IntroductionAdvisor, PointcutAdvisor등이 있는데 PointcutAdvisor 인터페이스는 포인트컷을 사용하여 조인포인트에 어드바이스를 적용하는 모든 Advisor가 구현한다.

프록시 연습 프로젝트

새 프로젝트 만들기

```
STS >> New >> Spring Starter Project

Project Name : spring-proxy-to-aop
Type : Maven
Packaging : Jar
Java : 8
Package : com.example.demo

Project Dependencies : Web, Lombok, Aspects
```

Proxy implements Target's Interfaces

타겟 클래스가 인터페이스를 구현하고 있는 경우, 타겟 클래스를 프록싱하는 클래스도 같은 인터페이스를 구현하는 방식으로 프록시 클래스를 만든다.

Hello.java

```
package com.example.demo.proxy1;

public interface Hello {
    public void say();
}
```

HelloImpl.java

```
package com.example.demo.proxy1;

// Target: 핵심로직을 제공하는 객체
public class HelloImpl implements Hello {

    @Override
    public void say() {
        // Primary Concern: 언제나 실행되어야 하는 로직
        System.out.println("핵심로직: Hello~!!!");
    }

}
```

Proxy는 Hello 인터페이스 구현체인 HelloImpl의 대리자의 역할을 수행한다. 이해를 돕기위해서 예를 들어보면 HelloImpl은 가수이고 Proxy는 매니저라고 비유를 들 수 있다. 클라이언트는 가수에게 노래를 불러달라고 요청할 수 있고 대신 매니저에게 노래를 불러달라고 요청할 수도 있다. 어느 경우에도 노래를 부르는 것은 가수다.

Proxy.java

```
package com.example.demo.proxy1;

public class Proxy implements Hello {

    private Hello hello;
```

```
    public Proxy(Hello hello) {
        this.hello = hello;
    }

    @Override
    public void say() {
        // 상황에 따라 실행되는 로직
        System.out.println("-----전 처리 부가로직-----");

        hello.say(); // 위임(Delegation): 핵심로직을 호출

        // 상황에 따라 실행되는 로직
        System.out.println("=====후 처리 부가로직=====");
    }

}
```

Test.java

```
package com.example.demo.proxy1;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Test { // Test는 Hello를 사용하는 Client 역할이다.
    @Autowired
    private Hello h;

    public void useHello() {
        h.say();
        // 위 메소드는 1번, 2번 중에 무엇과 같을까요? 코드만 보고는 모른다.
        // Hello 인터페이스의 구현체 중 하나일 것이라고 추측만이 가능하다.
        // h 변수가 가리키는 실체는 HelloImpl 객체일 수도 있고,
        // Proxy 객체일 수도 있다.
        // 실체는 런타임시에 결정되므로 코드 레벨에서는 미리 판단할 수 없다.
    }

    // 클라이언트 객체가 사용하는 로직이 빈번하게 변경되는 상황입니다.
    // 클라이언트 클래스의 소스코드를 바꾸지 않고도 이를 적용할 수 있는
    // 방법이 필요합니다.
    public static void main(String[] args) {
        Hello hello = new HelloImpl();
        hello.say(); // 1. 핵심로직만 실행

        System.out.println();

        Hello proxy = new Proxy(hello);
        proxy.say(); // 2. 핵심로직 + 부가로직을 실행
    }
}
```

클라이언트의 역할을 수행하는 Test가 실체가 HelloImpl인 객체의 say() 메소드를 호출하면 타겟 객체의 핵심 로직만이 수행된다.

클라이언트의 역할을 수행하는 Test가 실체가 Proxy인 객체의 say() 메소드를 호출하면 프록시의 메소드가 가진 로직이 수행된다. 더불어서, 프록시가 타겟 객체의 say() 메소드를 호출하면 타겟 객체의 핵심로직이 추가로 수행된다.

엔터프라이즈 환경에서는 핵심로직만 수행되어야 하는 경우만 추가적으로 부가로직도 수행되어야 하는 경우가 많이 발생한다. 더불어서, 프로그램 전역적으로 이러한 현상이 발생한다.

JDK Dynamic Proxy

자바가 제공하는 기술을 사용하면 동적으로 프록시 객체를 만들 수 있다.

MyAdvice.java

```
package com.example.demo.proxy1.jdk;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

// 개발자가 앞서서 만든 Proxy 클래스는 오로지 HelloImpl 타겟 클래스만
// 프록싱 할 수 있으나 InvocationHandler 인터페이스를 구현한 MyAdvice
// 클래스는 모든 타겟 클래스를 프록싱 할 수 있습니다.
public class MyAdvice implements InvocationHandler {
    private Object target;

    public MyAdvice(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

    // 상황에 따라 실행되는 로직
        System.out.println("-----Around Before Advice-----");

//        ((HelloImpl)target).say(); // 이렇게 사용하면 범용성을 잃어 버리게 됩니다.

        // 타겟 클래스의 핵심로직을 호출하는 방법 또한 범용성을 확보해야 하므로
        // 자바 리플렉션에서 지원하는 Method 클래스의 invoke 메소드를 사용합니다.
        // Method 객체는 내부에 호출해야 하는 메소드 정보를 갖고 있습니다.
        // 다음 코드는 target 객체 안에 메소드를 호출하면서 args 를 파라미터로
        // 전달하는 뜻이 됩니다.

    // 위임(Delegation): 핵심로직을 호출
        Object ret = method.invoke(target, args);

    // 상황에 따라 실행되는 로직
        System.out.println("=====Around After Advice======");

        return ret;
    }

}
```

Builder.java

```
package com.example.demo.proxy1.jdk;

public interface Builder {
    public boolean build();
```

```
}
```

BigBuilder.java

```
package com.example.demo.proxy1.jdk;

public class BigBuilder implements Builder {

    @Override
    public boolean build() {
        System.out.println("큰 건물을 건설한다.");
        return true;
    }

}
```

Test.java

```
package com.example.demo.proxy1.jdk;

import java.lang.reflect.Proxy;

public class Test { // Client

    public static void main(String[] args) {

        Hello hello = new HelloImpl();
        hello.say(); // 1. 핵심로직만 실행

        System.out.println();

        // 프록시 클래스를 타겟클래스의 숫자만큼 만드는 것은 굉장히
        // 번거로운 작업입니다. 그래서 스프링은 개발자 대신 프록시
        // 객체를 자동으로 생성하는 기술을 사용합니다.
        // Proxy 클래스는 개발자가 만들 필요가 없으므로 삭제했습니다.
        // 그런데 어디엔가는 부가로직을 정의해서 갖고 있어야 합니다.
        // 그래서 InvocationHandler 인터페이스를 구현하는 클래스를 만들고
        // invoke 메소드가 부가로직을 갖고 있도록 조치합니다.

        // JDK Dynamic Proxy
        Hello proxy = (Hello) Proxy.newProxyInstance(
                Hello.class.getClassLoader(), // 접근방법
                new Class<?>[] {Hello.class}, // 메소드 정보
                new MyAdvice(new HelloImpl())); // 부가로직

        proxy.say(); // 2. 핵심로직 + 부가로직을 실행

        System.out.println("\n---------------\n");

        Builder builder = new BigBuilder();
        builder.build();

        System.out.println();

        Builder proxyBuilder = (Builder) Proxy.newProxyInstance(
                Builder.class.getClassLoader(),
                new Class<?>[] {Builder.class},
                new MyAdvice(new BigBuilder())); // 어드바이스 클래스를 재사용
```

```
        proxyBuilder.build();
    }

}
```

Proxy extends Target

타겟 클래스가 인터페이스를 구현하고 있지 않은 경우, 프록시 클래스는 타겟 클래스를 상속하는 방식으로 만든다.

Hello.java

```
package com.example.demo.proxy2;

public class Hello {
    public void say() {
    // 언제나 실행되어야 하는 로직
        System.out.println("핵심로직: Hello~!!!");
    }
}
```

Proxy.java

```
package com.example.demo.proxy2;

public class Proxy extends Hello {

    @Override
    public void say() {
    // 상황에 따라 실행되는 로직
        System.out.println("-----전 처리 부가로직-----");

        super.say(); // 위임(Delegation): 핵심로직을 호출

    // 상황에 따라 실행되는 로직
        System.out.println("=====후 처리 부가로직=====");
    }

}
```

Test.java

```
package com.example.demo.proxy2;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Test { // Client
    @Autowired
    private Hello h;
```

```
    public void useHello() {
        h.say();
        // h가 가리키는 실체는 Hello로 만든 객체일 수도 있고,
        // Proxy로 만든 객체일 수도 있다.
        // 스프링이 무엇을 DI하는냐에 따라서 사용하는 실체가
        // 변경되고 수행로직이 달라진다.
    }

    public static void main(String[] args) {

        Hello hello = new Hello();
        hello.say(); // 1. 핵심로직만 실행

        System.out.println();

        Hello proxy = new Proxy();
        proxy.say(); // 2. 핵심로직 + 부가로직을 실행

    }

}
```

CGLIB Dynamic Proxy

CGLIB 라이브러리가 제공하는 기술을 사용하여 타겟 클래스를 상속하는 프록시 객체를 동적으로 만들 수 있다.

Hello.java

```
package com.example.demo.proxy2.cglib;

public class Hello {
    public void say() {
    // 언제나 실행되어야 하는 로직
        System.out.println("핵심로직: Hello~!!!");
    }

    public void bye() {
    // 언제나 실행되어야 하는 로직
        System.out.println("핵심로직: Goodbye~!");
    }
}
```

MyAdvice.java

```
package com.example.demo.proxy2.cglib;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class MyAdvice implements MethodInterceptor {

    @Override
```

```
    public Object invoke(MethodInvocation invocation) throws Throwable {

    // 상황에 따라 실행되는 로직
        System.out.println("-----Around Before Advice-----");

    // 위임(Delegation): 핵심로직을 호출
        Object ret = invocation.proceed();

    // 상황에 따라 실행되는 로직
        System.out.println("=====Around After Advice======");

        return ret;
    }

}
```

MethodInterceptor는 메소드 호출용 어라운드 어드바이스의 표준 인터페이스이다. MethodInvocation은 어드바이스를 추가하기 위한 메소드 호출을 나타내며 이 객체를 사용하면 메소드 호출이 실행되는 시점(메소드 실행 전 / 후)을 제어할 수 있다.

Test.java

```
package com.example.demo.proxy2.cglib;

import org.aopalliance.aop.Advice;
import org.springframework.aop.framework.ProxyFactory;

public class Test { // Client

    public static void main(String[] args) {

        Hello hello = new Hello();
        hello.say(); // 1. 핵심로직만 실행
        hello.bye();

        System.out.println();

        Advice advice = new MyAdvice();

        // CGLIB Dynamic Proxy
        ProxyFactory factory = new ProxyFactory();
        factory.setTarget(hello); // 타겟 객체 지정
        factory.addAdvice(advice); // 어드바이스 객체 설정

        // 타겟 객체와 어드바이스 객체만 지정하면 기본적으로
        // 타겟 객체의 모든 메소드를 호출할 때 어드바이스 로직이 작동합니다.

//        -----Around Before Advice-----
//        핵심로직: Hello~!!!
//        =====Around After Advice======

//        -----Around Before Advice-----
//        핵심로직: Goodbye~!
//        =====Around After Advice======

        // 그런데 때때로 타겟 객체의 일부 메소드만 어드바이스를 적용하고
        // 싶을 때가 있습니다. 이를 조절하는 기술로 Pointcut을 추가로
        // 적용해야 합니다.

        Hello proxy = (Hello) factory.getProxy();
```

```
        proxy.say(); // 2. 핵심로직 + 부가로직을 실행
        proxy.bye();


    }

}
```

대상 선택 전략 : Pointcut

- Pointcut은 모든 Join Point중 `Advice가 Weaving 되어야 할 Join Point의 집합(충고가 적용될 메소드)` 을 정의한 것이다.
- 교차점(PointCut)은 특정한 클래스의 특정한 메소드가 특정한 기준과 일치하는지를 판단한다. 만약 그 메소드가 실제로 일치한다면 충고가 적용된다.
- 스프링은 충고를 받으려고 하는 클래스와 메소드의 관점에서 교차점을 정의하며 충고는 클래스의 이름과 메소드 시그네처(Method Signature)와 같은 특징에 기초하여 대상 클래스와 메소드에 엮인다.
- 스프링의 교차점 프레임워크를 위한 핵심 인터페이스는 PointCut, PointCut은 메소드와 클래스에 기초하여 충고를 어디에 엮을지 결정한다.
- Pointcut 구현체를 사용하려면 먼저 Advisor 인터페이스의 인스턴스를 생성하거나 좀 더 구체적으로 PointcutAdvisor 인터페이스의 인스턴스를 생성해야 한다.

org.springframework.aop.support.StaticMethodMatcherPointcut	Target클래스의 정적인 정보(클래스이름, 메소드 이름)을 기반으로 pointcut을 만들때 사용하는 클래스
org.springframework.aop.support.DynamicMethodMatcherPointcut	Target클래스의 정적인 정보를 포함하여 동적인 정보(메소드 호출시 전달되는 인자 값)를 기반으로 Pointcut을 만들때 사용하는 클래스
Perl5RegexpMathodPointcut,	정규표현식을 이용하는 것이 가능하도록 지원
org.springframework.aop.support.NameMatchMethodPointCut	Pointcut에 전달되는 메소드 이름에 해당하는 메소드에 대해서만 Advice가 적용되도록 지원
org.springframework.aop.support.JdkRegexpMethodPointCut	JDK1.4의 정규식 지원 기능을 이용해 포인트컷 을 정의할 수 있게 해 준다.
Org.springframework.aop.support.annotation.AnnotationMatchingPointcut	클래스나 메소드에서 특정 자바 애노테이션을 찾는 포인트컷(JDK1.5이상)
org.springframework.aop.aspectj.AspectJExpressionPointcut	AspectJ 위버를 사용해 AspectJ 구문으로 포인트컷 표현식을 해석
org.springframework.aop.support.ComposablePointcut	둘 이상의 Pointcut을 union(), intercection() 연산을 통해 모으는데 사용
org.springframework.aop.support.ControlFlowPointcut	다른 메소드의 제어 흐름에 속한 모든 메소드에 대응되는 특수 사례 포인트컷으로 다른 메소드의 호출 결과에 의해 직.간접적으로 호출되는 메소드

- 충고자(Advisor)
  Aspect는 행동을 정의한 충고와 실행돼야 할 위치를 정의한 교차점(Pointcut)의 조합으로 이루어 진다.

스프링에서는 이를 위해 충고자라는 것을 제공한다. 충고자는 충고(Advice)와 교차점(Pointcut)을 하나의 객체로 합친것 이다.

AspectJExpressionPointcut

First.java

```
package com.example.demo.pointcut1.aspectj;

public class First {

    public void one() {
        System.out.println("First # one()");
    }

    public void one2() {
        System.out.println("First # one2()");
    }

    public void two() {
        System.out.println("First # two()");
    }

    public int add(int a, int b) {
        System.out.println("First # add(int a, int b)");
        return a + b;
    }

}
```

Second.java

```
package com.example.demo.pointcut1.aspectj;

public class Second {

    public void one() {
        System.out.println("Second # one()");
    }

    public void one2() {
        System.out.println("Second # one2()");
    }

    public void two() {
        System.out.println("Second # two()");
    }

    public double add(double a, double b) {
        System.out.println("Second # add(double a, double b)");
        return a + b;
    }

}
```

MyAdvice.java

```
package com.example.demo.pointcut1.aspectj;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class MyAdvice implements MethodInterceptor {

    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {

        System.out.println("-----Around Before Advice-----");

        Object ret = invocation.proceed();

        System.out.println("=====Around After Advice======");

        return ret;
    }

}
```

Test.java

```
package com.example.demo.pointcut1.aspectj;

import org.aopalliance.aop.Advice;
import org.springframework.aop.Advisor;
import org.springframework.aop.aspectj.AspectJExpressionPointcut;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;

public class Test {

    public static void main(String[] args) {

        Advice advice = new MyAdvice();

        /*
         *  execution() : 사용하는 표현식 종류
         *  * one*(..) : 리턴자료형 메소드명(파라미터) : 필수 항목
         *
         *  * : all
         *  one* : 메소드명이 one 문자열로 시작
         *  (..) : 파라미터 개수가 0~N개 가능, 각 파라미터 자료형은 상관이 없다.
         *
         *  포인트컷 조합 시 and, or , not을 사용할 수 있다.
         */
        AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();
        pointcut.setExpression("execution(* one*(..))");

        Advisor advisor = new DefaultPointcutAdvisor(pointcut, advice);

        ProxyFactory factory = new ProxyFactory();
        factory.setTarget(new First());
        factory.addAdvisor(advisor);

        First f = (First) factory.getProxy();
        f.one();
```

```
        f.one2();
        f.two();
        f.add(2, 3);

    }

}
```

PCD(Pointcut Designator)

execution PCD는 가장 많이 사용하는 포인트컷 지정자입니다. execution 포인트컷 지정자는 다음 3가지 항목이 필수적으로 선언되어야 합니다. 필수항목의 선언순서는 자바의 메소드 작성 시 사용하는 순서와 같습니다.

필수적인 선언 항목

1. 리턴자료형
2. 메소드명
3. 파라미터

선택적인 선언 항목

1. 접근제어자
2. 패키지
3. 클래스
4. 예외

XML에서는 `and` 나 `or` 연산자를 이용하여 표현식을 연결할 수 있다. Annotation에서는 `&&` 나 `||` 연산자를 이용하여 표현식을 연결할 수 있다. 인터페이스를 구현하고 있는 대상 객체에 대해서 포인트컷을 지정하려면 인터페이스를 기준으로 포인트컷을 작성한다.

- `execution(public * *(..))`
  execution(접근제어자 리턴자료형 메소드명(파라미터)) 순서로 선언합니다.
  메소드명, 파라미터는 상관이 없고 접근제어자가 public인 메소드가 포인트컷

- `execution(* com.example.*.*(..))`
  com.example 패키지 아래의 모든 클래스의 모든 메소드가 포인트컷

- `execution(* com.example..*.*(..))`
  com.example 패키지 및 하위 패키지의 모든 클래스의 모든 메소드가 포인트컷

- `execution(public void insert*(..))`
  접근제어자는 public이고 리턴은 없고 메서드명은 insert로 시작하며 파라미터는 0~N 개인 메서드가 포인트컷

- `execution(* com.example.*.*())`
  com.example 패키지 아래의 모든 클래스 내 파라미터가 없는 모든 메서드가 포인트컷

- `execution(* delete*(*))`

메서드명이 delete로 시작하며 파라미터가 1개인 메서드가 포인트컷
파라미터 부분에 `*`는 자료형이 무엇이든 상관이 없다는 의미입니다.

- `execution(* delete*(*,*))`
  메서드명이 delete로 시작하며 파라미터가 2개인 메서드가 포인트컷

- `execution(* find*(Integer, ..))`
  메서드명은 find로 시작하고 첫 번째 파라미터로 Integer 자료형의 값을 받으며, 그 다음 파라미터는 0~N 개가 올 수 있다.

- `execution(* *(..) throws SQL*)`
  `SQL` 문자열로 시작하는 예외가 메소드 뒤에 선언되어 있다면 포인트컷

단순히 클래스 자료형만을 이용하여 적용대상을 결정하고 싶을 때 within 포인트컷 지정자을 사용합니다.

- `within(org.example.dao.FooDao)`
  org.example.dao.FooDao 클래스 내 모든 메소드가 포인트컷

- `within(com.example.*)`
  com.example 패키지 내의 모든 클래스의 모든 메소드가 포인트컷

- `within(com.example..*)`
  com.example 패키지 및 하위 패키지의 모든 클래스의 모든 메소드가 포인트컷

빈 컨테이너에 등록된 객체들의 빈 이름을 이용하여 적용대상을 결정하고 싶을 때 bean 포인트컷 지정자을 사용합니다.

- `bean(emp*)`
  빈 이름이 emp로 시작되는 모든 객체의 모든 메소드가 포인트컷

- `bean(*dataSource) || bean(*DataSource)`
  빈 이름이 `dataSource` 또는 `DataSource`로 끝나는 모든 객체의 모든 메소드가 포인트컷

- `!bean(empService)`
  빈 이름이 empService인 객체를 제외한 모든 빈 객체의 모든 메소드가 포인트컷

Spring AOP는 프록시 기반 시스템이며 프록시 객체 자체(`this`에 바인딩 됨)와 프록시 뒤의 타겟 객체(`target`에 바인딩 됨)를 구별합니다.

- `this(com.example.dao.EmpDao)`
  다이나믹 하게 생성되는 프록시 객체를 EmpDao 자료형으로 지칭할 수 있을 때 포인트컷

- `target(com.example.dao.EmpDao)`
  타겟 객체를 EmpDao 자료형으로 지칭할 수 있을 때 포인트컷

- `args(java.io.Serializable)`
  메소드 파라미터가 한 개이고 Serializable 인터페이스 구현체일 때 포인트컷

`target` 포인트 컷 지정자와 `@target` 포인트 컷 지정자는 다른 의미로 사용합니다.

- `@target(org.springframework.transaction.annotation.Transactional)`

타겟 객체가 RetentionPolicy.RUNTIME으로 설정된 `@Transactional` 애노테이션을 가진다면 포인트컷

- `@within(org.springframework.transaction.annotation.Transactional)`
  타겟 객체가 RetentionPolicy.CLASS으로 설정된 `@Transactional` 애노테이션을 가진다면 포인트컷. 런타임 시 해당 애노테이션이 없어도 적용대상으로 선택되는 부분이 `@target`과의 차이점 입니다.

- `@annotation(org.springframework.transaction.annotation.Transactional)`
  타겟 객체의 메소드가 `@Transactional` 애노테이션을 가진다면 포인트컷

- `execution(* *(..)) and @annotation(@annotation)`
  `@annotation`이 있는 모든 메소드가 포인트컷

- `execution(* *(..,@annotation (*),..))`
  `@annotation`을 파라메터로 갖고 있는 모든 메소드가 포인트컷

- `@args(com.example.AdviceRequired)`
  파라미터인 객체가 `@AdviceRequired` 애노테이션을 가진다면 포인트컷

ProxyFactoryBean

Test.java

```
package com.example.demo.pointcut2.proxyfactorybean;

import org.aopalliance.aop.Advice;
import org.springframework.aop.Advisor;
import org.springframework.aop.aspectj.AspectJExpressionPointcut;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

    public static void main(String[] args) {

//        Advice advice = new MyAdvice();

        /*
         *  execution() : 사용하는 표현식 종류
         *  * one*(..) : 리턴자료형 메소드명(파라미터)
         *  * : all
         *  one* : 메소드명이 one 문자열로 시작
         *  (..) : 파라미터 개수가 0~N개 가능, 각 파라미터 자료형은 상관이 없다.
         */
//        AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();
//        pointcut.setExpression("execution(* one*(..))");
//
//        Advisor advisor = new DefaultPointcutAdvisor(pointcut, advice);
//
//        ProxyFactory factory = new ProxyFactory();
//        factory.setTarget(new First());
//        factory.addAdvisor(advisor);
//
```

```
//        First f = (First) factory.getProxy();
//        f.one();
//        f.one2();
//        f.two();
//        f.add(2, 3);

        ApplicationContext context = new ClassPathXmlApplicationContext(
                "proxy-factory-bean.xml");

        First f = (First) context.getBean("proxyFirst");
        f.one();
        f.one2();
        f.two();
        f.add(2, 3);
    }

}
```

proxy-factory-bean.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

<!--
    Advice advice = new MyAdvice();

    AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();
    pointcut.setExpression("execution(* one*(..))");

    Advisor advisor = new DefaultPointcutAdvisor(pointcut, advice);
 -->

    <bean id="advice"
        class="com.example.demo.pointcut2.proxyfactorybean.MyAdvice"></bean>

    <bean id="pointcut"
        class="org.springframework.aop.aspectj.AspectJExpressionPointcut">
        <property name="expression" value="execution(* one*(..))"></property>
    </bean>

    <bean id="advisor"
        class="org.springframework.aop.support.DefaultPointcutAdvisor">
        <constructor-arg ref="pointcut"></constructor-arg>
        <constructor-arg ref="advice"></constructor-arg>
    </bean>

<!--
    ProxyFactory factory = new ProxyFactory();
    factory.setTarget(new First());
    factory.addAdvisor(advisor);
```

```
    First f = (First) factory.getProxy();
-->

    <bean id="first"
        class="com.example.demo.pointcut2.proxyfactorybean.First"></bean>

    <!--
        ProxyFactoryBean은 자신이 빈컨테이너에 등록되는 것이 아니라
        first 객체를 프록싱하는 프록시객체를 등록합니다.
        따라서, id는 proxyFirst 라고 부르는 것이 적절하다 하겠습니다.
     -->
    <bean id="proxyFirst"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="first"></property>
        <property name="interceptorNames">
            <list>
                <value>advisor</value>
            </list>
        </property>
    </bean>

    <!--
        ProxyFactoryBean 설정은 1회용입니다. 100개 타겟 객체를 프록싱하는 프록시 객체를
        만들기 위해서 위 설정이 100번 반복되어야 합니다. 이는 매우 번거로운 작업입니다.
        그래서 스프링은 단 한번의 설정으로 다수의 프록시 객체를 사용할 수 있도록 설정할 수
        있는 aop 네임스페이스 태그를 지원합니다.
     -->

</beans>
```

Spring Advices

스프링은 어드바이스를 기동 순서와 기동 조건에 따라 다음 5가지로 분류한다.

주변충고 (around advice)	org.aopallince.intercept.MethodInterceptor	**메소드 호출 전/후 실행.** 필요하다면 자체 구현을 통해 메소드 실행을 SKIP 가능하며 타겟 객체의 메소드의 리턴값을 대체 가능하다.
사전충고 (before advice)	org.springframework.aop.MethodBeforeAdvice	스프링의 조인포인트는 항상 메소드 호출이므로 사전충고는 **메소드 실행전에 전처리** 기능을 한다. 대상메소드가 실행되기 전에 호출됨
사후충고 (after returning advice)	org.springframework.aop.AfterReturningAdvice	**대상메소드가 리턴한후에 호출됨(정상종료).** 메소드 호출타겟, 메소드로 넘어온 인자, 반환 값등에 접근가능하며 이 어드바이스가 호출될 때 이미 메소드가 실행되었으므로 메소드 호출제어 불가능
예외충고 (throws advice)	org.springframework.aop.ThrowsAdvice	대상메소드가 예외를 던질 때 호출됨. 이 충고를 이용하면 특정 예외 접근가능하며 예외를 던진 메소드, 메소드 호출 인자, 호출 타겟에 접근할 수 있다.
사후충고(finally) (after advice)	org.aopallince.intercept.AfterAdvice	After-returning 어드바이스는 조인포인트의 메소드 실행을 마치고 값을 반환한 후 실행. 하지만 after(finally) 어드바이스는 **메소드의 실행 결과와 관계없이 실행 즉 오류 나서 예외를 던지더라도 실행된다.**

AOP 전역설정 : XML

aop:advisor

Test.java

```
package com.example.demo.pointcut3.aopns;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

    public static void main(String[] args) {

        ApplicationContext context = new ClassPathXmlApplicationContext(
                "pointcut3-aopns.xml");

        First f = context.getBean(First.class);
        f.one();
        f.one2();
        f.two();
        f.add(2, 3);

        System.out.println();

        Second s = context.getBean(Second.class);
```

```
        s.one();
        s.one2();
        s.two();
        s.add(2, 3);
    }

}
```

pointcut3-aopns.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.example.demo.pointcut3.aopns"/>

    <bean id="pointcut"
        class="org.springframework.aop.aspectj.AspectJExpressionPointcut">
        <property name="expression" value="execution(* one*(..))"/>
    </bean>

    <!-- aop:config 안에 설정은 빈 컨테이너 등록 객체들 모두에게 적용 됩니다. -->
    <aop:config>
        <!--
        <bean id="advisor"
            class="org.springframework.aop.support.DefaultPointcutAdvisor">
            <constructor-arg ref="pointcut"/>
            <constructor-arg ref="myAdvice"/>
        </bean>
         -->

        <!-- 어드바이스, 포인트컷 객체를 이미 갖고 있을 때 aop:advisor 태그를 사용합니다. -->
        <aop:advisor advice-ref="myAdvice" pointcut-ref="pointcut"/>
    </aop:config>

</beans>
```

aop:pointcut

Test.java

```
package com.example.demo.pointcut4.aopns;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {
```

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

        ApplicationContext context = new ClassPathXmlApplicationContext(
                "pointcut4-aopns.xml");

        First f = context.getBean(First.class);
        f.one();
        f.one2();
        f.two();
        f.add(2, 3);

        System.out.println();

        Second s = context.getBean(Second.class);
        s.one();
        s.one2();
        s.two();
        s.add(2, 3);
    }

}
```

pointcut4-aopns.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/sch
        http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/sp

    <context:component-scan base-package="com.example.demo.pointcut4.aopns"/>

    <!-- aop:config 안에 설정은 빈 컨테이너 등록 객체들 모두에게 적용 됩니다. -->
    <aop:config>
        <!--
        <bean id="pointcut"
            class="org.springframework.aop.aspectj.AspectJExpressionPointcut">
            <property name="expression" value="execution(* one*(..))"/>
        </bean>
         -->
        <aop:pointcut expression="execution(* two(..))" id="pointcut"/>

        <aop:advisor advice-ref="myAdvice" pointcut-ref="pointcut"/>
    </aop:config>

</beans>
```

aop:aspect

MyAroundAdvice.java

```
package com.example.demo.pointcut5.aopns;
```

```
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.stereotype.Component;

@Component
public class MyAroundAdvice implements MethodInterceptor {

    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {

        System.out.println("-----Around Before Advice-----");

        Object ret = invocation.proceed(); // 타겟 객체의 핵심로직 메소드를 호출

        System.out.println("=====Around After Advice======");

        return ret;
    }

}
```

MyBeforeAdvice.java

```
package com.example.demo.pointcut5.aopns;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;
import org.springframework.stereotype.Component;

@Component
public class MyBeforeAdvice implements MethodBeforeAdvice {

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

}
```

MyAspect.java

```
package com.example.demo.pointcut5.aopns;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.stereotype.Component;

@Component
public class MyAspect {

    // 항상기동: 타겟 메소드 보다 앞서서 수행됩니다.
    public void myBefore(JoinPoint jointPoint) {
        System.out.println("-----MyAspect # myBefore------");
        Object[] args = jointPoint.getArgs();
        if (args != null && args.length > 0) {
            for (Object object : args) {
                System.out.println(">>> " + object);
            }
```

```
        }
    }

    // 앞은 항상기동, 뒤는 조건기동: 타겟 메소드를 감싼 형태로 앞, 뒤에서 수행됩니다.
    public Object myAround(ProceedingJoinPoint jointPoint) throws Throwable {
        System.out.println("-----MyAspect # myAround : Before------");

        // 여기서 타겟 메소드를 호출하자라는 코드가 있어야만
        // 위 코드는 Before로 실행되고 아래 코드는 After로 실행됩니다.
        Object ret = jointPoint.proceed();

        System.out.println("=====MyAspect # myAround : After======");

        return ret;
    }

    // 조건기동: 타겟 메소드가 정상적으로 처리된 후 수행됩니다.
    public void myAfterReturning(JoinPoint jointPoint, Object result) {
        System.out.println("=====MyAspect # myAfterReturning======");
        System.out.println(">>> " + result);
    }

    // 조건기동: 타겟 메소드 처리 시 예외가 발생한 경우 수행됩니다.
    public void myAfterThrowing(JoinPoint jointPoint, Throwable error) {
        System.out.println("=====MyAspect # myAfterThrowing======");
        System.out.println(">>> " + error.getMessage());
    }

    // 항상기동: 타겟 메소드가 수행된 다음 뒤에서 수행됩니다.
    public void myAfter(JoinPoint jointPoint) {
        System.out.println("=====MyAspect # myAfter======");
    }

}
```

JoinPoint 인터페이스

어드바이스에서 호출되는 타겟 객체의 메소드와 관련한 정보를 사용할 수 있다.

- `JoinPoint#getSignature()` 클라이언트가 호출한 메소드의 시그너처(리턴타입, 이름, 매개변수) 정보가 저장된 Signature 객체를 리턴한다.
  - `Signature#getName()`
    클라이언트가 호출한 메소드 이름을 리턴한다.
  - `Signature#toLongString()`
    클라이언트가 호출한 메소드의 리턴타입, 이름, 매개변수를 패키지 경로까지 포함하여 리턴한다.
  - `Signature#toShortString()`
    클라이언트가 호출한 메소드 시그너처를 축약된 문자열로 리턴한다.
- `JoinPoint#getTarget()`
  클라이언트가 호출한 메소드를 가지고 있는 타겟 객체를 리턴한다.
- `JoinPoint#getArgs()`
  클라이언트가 메소드를 호출할 때 넘겨준 파라미터 정보를 Object 배열로 리턴한다.

Test.java

```
package com.example.demo.pointcut5.aopns;
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

    public static void main(String[] args) {

        ApplicationContext context = new ClassPathXmlApplicationContext(
                "pointcut5-aopns.xml");

        First f = context.getBean(First.class);
        f.one();
        f.one2();
        f.two();
        f.add(2, 3);

        try {
            f.divide(4, 2);
            f.divide(4, 0);
        } catch (Exception ignore) {}

        System.out.println();

        Second s = context.getBean(Second.class);
        s.one();
        s.one2();
        s.two();
        s.add(2, 3);
    }

}
```

pointcut5-aopns.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.example.demo.pointcut5.aopns"/>

    <!-- aop:config 안에 설정은 빈 컨테이너 등록 객체들 모두에게 적용 됩니다. -->
    <aop:config>
        <aop:pointcut expression="execution(* two(..))" id="pointcut"/>

        <aop:advisor advice-ref="myBeforeAdvice" pointcut-ref="pointcut"/>
        <aop:advisor advice-ref="myAroundAdvice" pointcut-ref="pointcut"/>

        <!--
            인터페이스를 구현한 클래스를 어드바이스로 사용하는 경우에는 advisor 태그를 사용하고
            그렇지 않은 클래스를 어드바이스로 사용하는 경우에는 aspect 태그를 사용합니다.
         -->
        <aop:aspect ref="myAspect">
```

```
            <!--
                빈 아이디 myAspect가 가리키는 객체에 다수의 메소드가 존재합니다.
                스프링은 어느 메소드가 어드바이스 용 메소드인지 모릅니다.
                더불어서 어느 시점에 메소드를 기동시켜야 하는지도 모릅니다.
                따라서, 추가적으로 다음 태그들을 사용하여야 합니다.

                aop:before =같은의미= implements MethodBeforeAdvice
                aop:around =같은의미= implements MethodInterceptor
             -->
            <aop:before method="myBefore" pointcut="execution(* add(..))"/>
            <aop:around method="myAround" pointcut-ref="pointcut"/>
            <aop:after-returning method="myAfterReturning" pointcut="execution(* divide(..))" returnir
            <aop:after-throwing method="myAfterThrowing" pointcut="execution(* divide(..))" throwing='
            <aop:after method="myAfter" pointcut-ref="pointcut"/>
        </aop:aspect>

    </aop:config>

</beans>
```

AOP 전역설정 : Annotation

MyAspect.java

```
package com.example.demo.pointcut6.aopanno;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class MyAspect {

//<aop:config proxy-target-class="true">
//    <aop:pointcut expression="execution(* two(..))" id="pointcut"/>
//
//    <aop:aspect ref="myAspect">
//        <aop:before method="myBefore" pointcut="execution(* add(..))"/>
//        <aop:around method="myAround" pointcut-ref="pointcut"/>
//        <aop:after-returning method="myAfterReturning" pointcut="execution(* divide(..))" returning=
//        <aop:after-throwing method="myAfterThrowing" pointcut="execution(* divide(..))" throwing="er
//        <aop:after method="myAfter" pointcut-ref="pointcut"/>
//    </aop:aspect>
//</aop:config>

    @Pointcut("execution(* two(..))")
    public void pointcut() {}

    @Before("execution(* add(..))")
```

```
    public void myBefore(JoinPoint jointPoint) {
        System.out.println("-----MyAspect # myBefore------");
        Object[] args = jointPoint.getArgs();
        if (args != null && args.length > 0) {
            for (Object object : args) {
                System.out.println(">>> " + object);
            }
        }
    }

    @Around("pointcut()")
    public Object myAround(ProceedingJoinPoint jointPoint) throws Throwable {
        System.out.println("-----MyAspect # myAround : Before------");

        Object ret = jointPoint.proceed();

        System.out.println("=====MyAspect # myAround : After======");

        return ret;
    }

    @AfterReturning(pointcut="execution(* divide(..))", returning="result")
    public void myAfterReturning(JoinPoint jointPoint, Object result) {
        System.out.println("=====MyAspect # myAfterReturning======");
        System.out.println(">>> " + result);
    }

    @AfterThrowing(pointcut="execution(* divide(..))", throwing="error")
    public void myAfterThrowing(JoinPoint jointPoint, Throwable error) {
        System.out.println("=====MyAspect # myAfterThrowing======");
        System.out.println(">>> " + error.getMessage());
    }

    @After("pointcut()")
    public void myAfter(JoinPoint jointPoint) {
        System.out.println("=====MyAspect # myAfter======");
    }

}
```

Test.java

```
package com.example.demo.pointcut6.aopanno;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

    public static void main(String[] args) {

        ApplicationContext context = new ClassPathXmlApplicationContext(
                "pointcut6-aopanno.xml");

        First f = context.getBean(First.class);
        f.one();
        f.one2();
        f.two();
        f.add(2, 3);
```

```
        try {
            f.divide(4, 2);
            f.divide(4, 0);
        } catch (Exception ignore) {}

        System.out.println();

        Second s = context.getBean(Second.class);
        s.one();
        s.one2();
        s.two();
        s.add(2, 3);
    }

}
```

pointcut6-aopanno.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.example.demo.pointcut6.aopanno"/>

    <!-- AOP 설정을 애노테이션으로 한다고 통보합니다. 항상 CGLIB 방식의 프록시를 사용하라고 설정합니다. -->
    <aop:aspectj-autoproxy proxy-target-class="true"/>

    <!--
    <aop:config proxy-target-class="true">
        <aop:pointcut expression="execution(* two(..))" id="pointcut"/>

        <aop:aspect ref="myAspect">
            <aop:before method="myBefore" pointcut="execution(* add(..))"/>
            <aop:around method="myAround" pointcut-ref="pointcut"/>
            <aop:after-returning method="myAfterReturning" pointcut="execution(* divide(..))" returnir
            <aop:after-throwing method="myAfterThrowing" pointcut="execution(* divide(..))" throwing="
            <aop:after method="myAfter" pointcut-ref="pointcut"/>
        </aop:aspect>
    </aop:config>
     -->

</beans>
```

`<aop:aspectj-autoproxy>` 태그 대신 `@EnableAspectJAutoProxy` 애노테이션을 사용할 수 있다.

AOP Dependency

스프링 레가시 프로젝트에서 AOP 기술을 사용하기 위한 디펜던시 설정은 다음과 같다.

pom.xml

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.8.13</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.13</version>
</dependency>
```

aspectjrt는 런타임 프로그램이다. 이 라이브러리는 추가하면 AspectJ의 기능을 사용할 수 있게 된다. AspectJExression을 사용한다면 aspectjweaver가 필요하다.

AOP XML 설정 vs Annotation 설정

- 스프링 애플리케이션이 XML 기반이라면 aop 네임스페이스를 이용 하는것이 적절하다. 이렇게 하면 DI, AOP 설정 방식을 일관되게 유지할 수 있기 때문이다.
- 애플리케이션이 애노테이션 기반이라면 `@AspectJ` 애노테이션을 사용 한다. `@AspectJ` 애노테이션을 사용하는 경우 모듈 안에 어스펙트 관련 정보를 캡슐화 할 수 있기 때문에 유지보수가 용이하다.
- aop 네임스페이스와 `@AspectJ` 애노테이션의 차이
    - 포인트컷 구문이 조금 다르다. aop 네임스페이스는 `and`, `@AspectJ` 애노테이션에서는 `&&`를 사용한다.
    - aop 네임스페이스에서는 `싱글톤` 방식의 애스펙트 인스턴스화 모델만 지원한다.
    - `@AspectJ` 애노테이션 방식에서는 두 개의 포인트컷 정의 `(helloExec(intValue) && inMyDependency())`를 사전충고, 주변 충고에서 조합할 수 있지만 aop 네임스페이스에서는 조건을 조합한 포인트 컷을 새로 생성해야 한다.