Spring Framework
- 스프링 AOP

학습 목표

- 트랜잭션처리에 AOP가 사용되기 때문에 한번은 보고 가야 할 AOP
- 스프링 프레임워크에서 어려운 개념 중 하나
- 현재 당장 이해가 어렵다면 당장 이해 보다는 예제에서 다시 보자.

CONTENTS

- AOP
- Spring에서의 AOP
- XML 기반의 POJO 클래스를 이용한 AOP 구현
- 어노테이션 기반의 AOP 구현
- 표현식

AOP

- **AOP[Aspect Oriented Programming]**
  - **개별 프로그래밍 언어는 프로그램 개발을 위해 고유한 관심사 분리 (Separation of Concerns) 패러다임을 갖는다.**
    - 예를 들면 절차적 프로그래밍은 상태 값을 갖지 않는 연속된 함수들의 실행을 프로그램으로 이해하고 모듈을 주요 분리 단위로 정의한다.
      객체지향 프로그래밍은 일련의 함수 실행이 아닌 상호작용하는 객체들의 집합으로 보며 클래스를 주요 단위로 한다.
    - 객체지향 프로그래밍은 많은 장점에도 불구하고, 다수의 객체들에 분산되어 중복적으로 존재하는 공통 관심사가 존재한다.
      이들은 프로그램을 복잡하게 만들고, 코드의 변경을 어렵게 한다.

AOP

- 관점 지향 프로그래밍(AOP, Apsect-oriented programming)은 이러한 객체지향 프로그래밍의 문제점을 보완하는 방법으로 핵심 관심사를 분리하여 프로그램 모듈화를 향상시키는 프로그래밍 스타일이다.

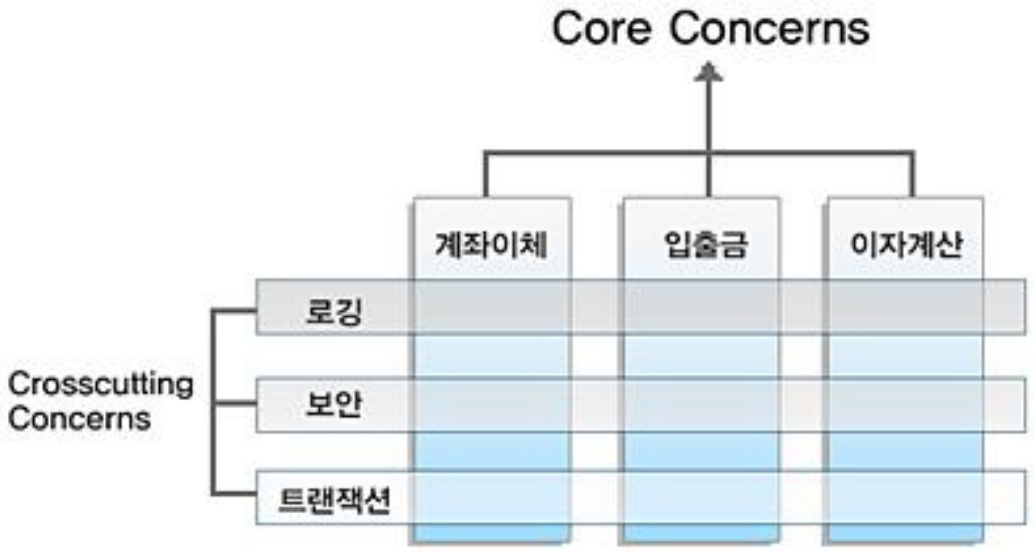
  AOP는 객체를 핵심 관심사와 횡단 관심사로 분리하고, 횡단 관심사를 관점(Aspect)이라는 모듈로 정의하고 핵심 관심사와 엮어서 처리할 수 있는 방법을 제공한다.

핵심 비즈니스 로직

계좌이체, 입출금, 이자계산

핵심 로직에 적용되는 공통 로직

로깅, 보안, 트랜잭션



AOP 사용

1. 간단한 메소드 성능 검사

개발 도중 특히 DB에 다량의 데이터를 넣고 빼는 등의 배치 작업에 대하여 시간을 측정해 보고 쿼리를 개선하는 작업은 매우 의미가 있다.

이 경우 매번 해당 메소드 처음과 끝에 System.currentTimeMills();를 사용하기는 매우 번거롭다.

이런 경우 해당 작업을 하는 코드를 밖에서 설정하고 해당 부분을 사용하는 편이 편리하다.

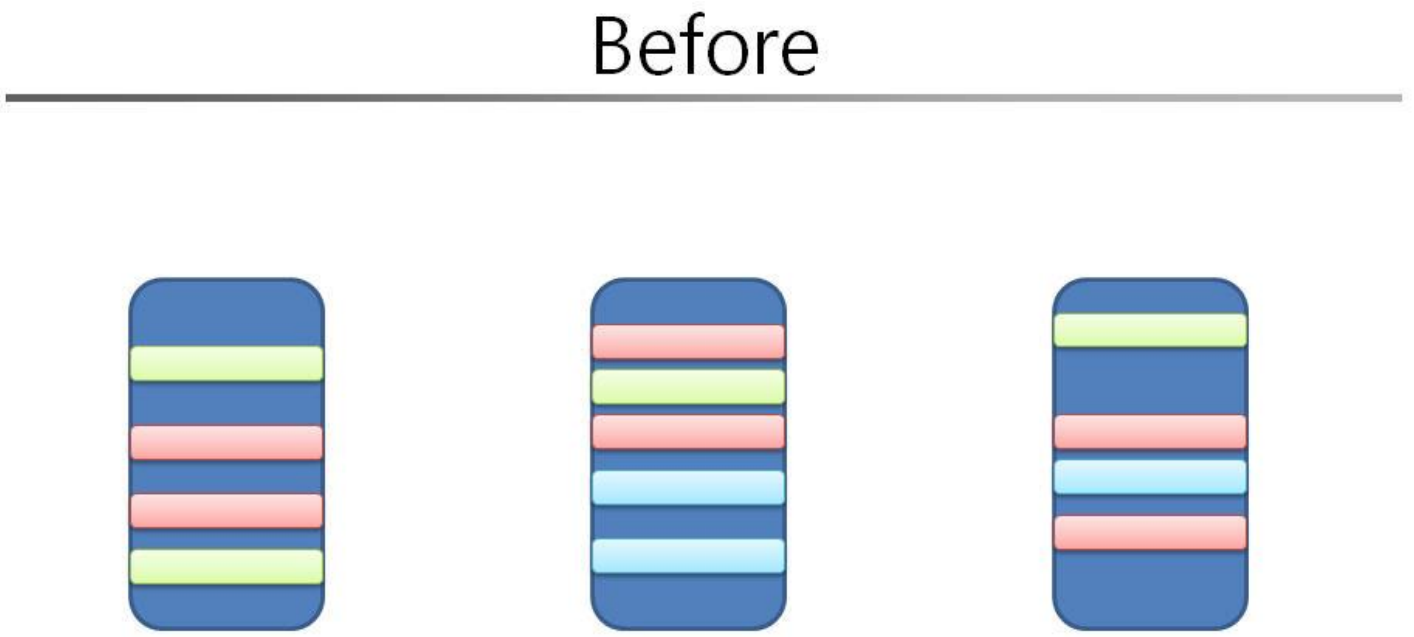
2. 트랜잭션 처리

트랜잭션의 경우 비지니스 로직의 전후에 설정된다. 하지만 매번 사용하는 트랜잭션(try~catch부분)의 코드는 번거롭고, 소스를 더욱 복잡하게 보여준다.

3. 아키텍처 검증

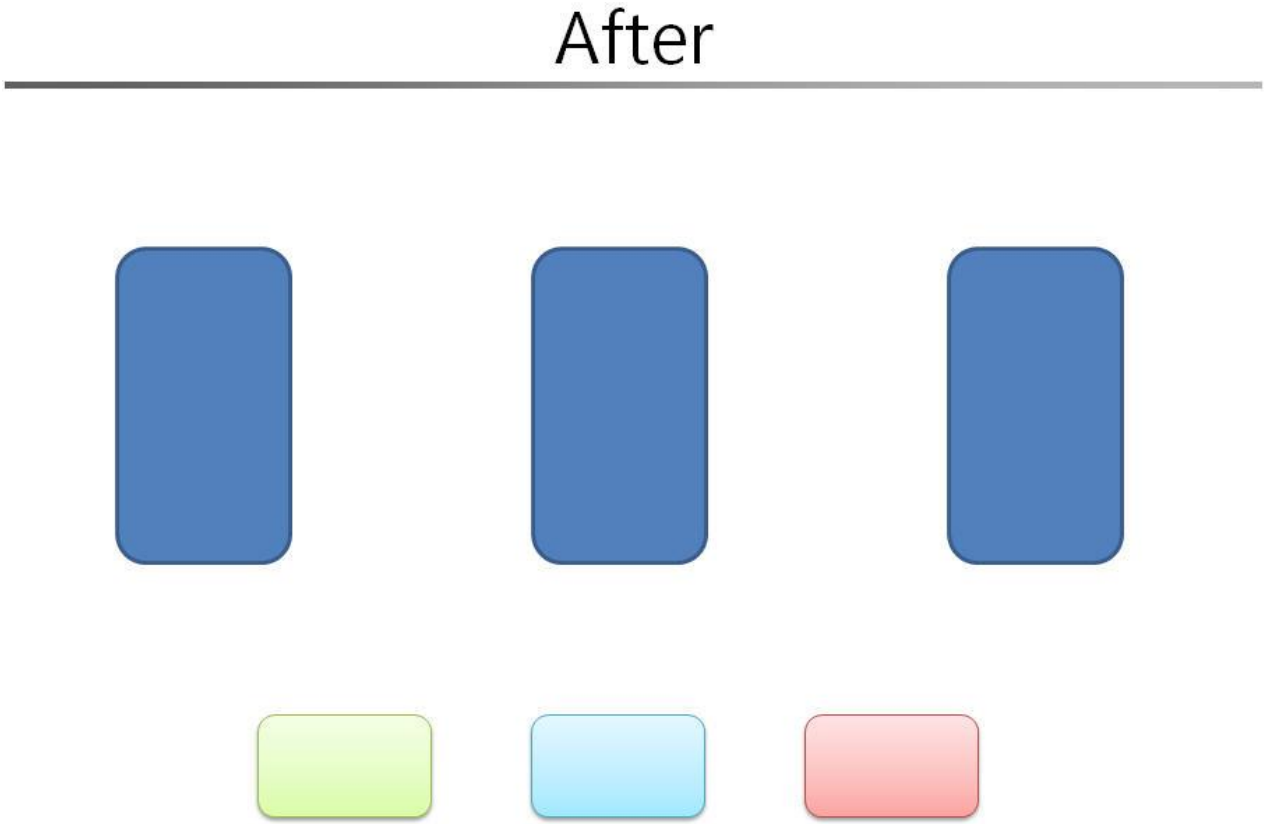
AOP

- 다음 그림은 객체지향 프로그래밍 개발에서 핵심 관심사와 횡단 관심사가 하나의 코드로 통합되어 개발된 사례를 보여준다.

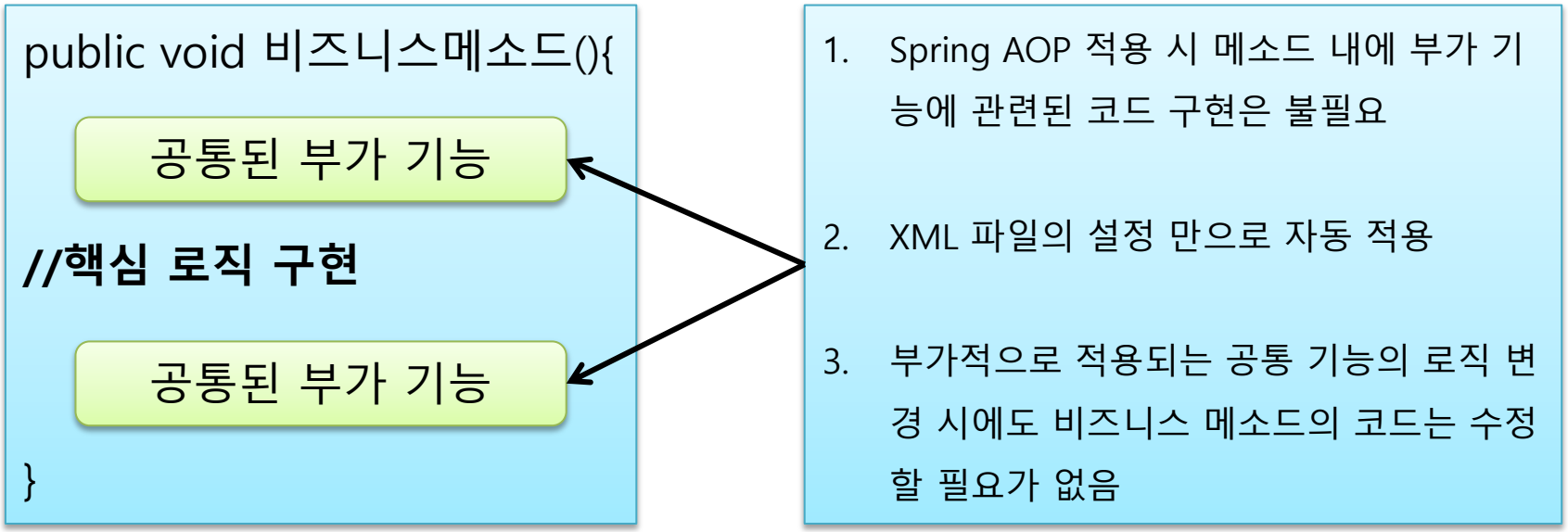


AOP

- 객체지향 프로그래밍 코드에 AOP를 적용하면 다음 그림처럼 각 코드에 분산되어 있던 횡단 관심사는 관점으로 분리되어 정의된다. AOP는 엮기(Weaving)라는 방식을 이용하여 분리된 관점을 핵심 관심사와 엮는다.



- **AOP 적용**
  - 개발자들은 횡단 관심 모듈을 각각 독립된 모듈로 중복 없이 작성하고 이를 적합한 위치에 XML 설정 파일에 설정하는 등의 작업을 통해 핵심 관심 모듈과 결합시키기 때문에 서로 독립성을 가진 다 차원의 모듈로 작성 할 수 있음
  - 재사용성과 보수성을 높일 수 있음



AOP의 주요 용어

- **Concern**
  - 애플리케이션을 개발하기 위하여 관심을 가지고 구현 해야 하는 각각의 기능들을 관심 사항(Concern)이라 함
- **core concern**
  - 핵심 관심 사항
  - 해당 애플리케이션이 제공하는 핵심이 되는 비즈니스 로직 의미
- **cross-cutting concern**
  - 횡단[공통] 관심 사항
  - 하나의 영역에서만 활용되는 고유한 관심 사항(Concern)이 아니라, 여러 클래스 혹은 여러 계층의 애플리케이션 전반에 걸쳐서 공통적으로 필요로 하는 기능들 의미

AOP의 주요 용어

- **Target**

  핵심 로직을 구현하는 클래스 공통 관심 사항을 적용 받게 되는 대상으로 어드바이스가 적용되는 객체

- **Aspect**

  여러 객체에 공통으로 적용되는 공통 관심 사항

- **Advice**

  조인 포인트에 삽입되어 동작할 수 있는 공통 관심 사항의 코드

  *동작시점 : Spring에서는 조인포인트 실행 전, 후로 before, after, after returning, after throwing, around로 구분

AOP의 주요 용어

- **joinpoint**

  「클래스의 인스턴스 생성 시점」,「메소드 호출 시점」 및「예외 발생 시점」과 같이 애플리케이션을 실행할 때 특정 작업이 시작되는 시점으로 Advice를 적용 가능한 지점, 즉 어드바이스가 적용될 수 있는 위치

- **Pointcut**

  여러 개의 Joinpoint를 하나로 결합한(묶은) 것

- **Advisor**

  Advice와 Pointcut를 하나로 묶어 취급한 것

- **weaving**

  공통 관심 사항의 코드인 Advice를 핵심 관심 사항의 로직에 적용 하는 것을 의미

Weaving방식- Advice를 위빙 하는 방식 3가지

- **컴파일 시 위빙하기**

  AOP가 적용된 새로운 클래스 파일이 생성됨

  AspectJ에서 사용 하는 방식

- **클래스 로딩 시 위빙하기**

  로딩한 바이트 코드를 AOP가 변경하여 사용

  AOP 라이브러리는 JVM이 클래스를 로딩할 때 클래스 정보를 변경할 수 있는 에이전트 제공 (원본 클래스 파일 변경 없이 클래스를 로딩할 때 JVM이 변경된 바이트 코드를 사용하도록 함)

- **런타임 시 위빙하기 (Spring Frlaework 에서 적용 방식)**

  소스 코드나 클래스 정보 자체를 변경하지 않음.

  프록시를 이용하여 AOP적용 .

  프록시 기반의 제한사항 : 메소드 호출할 경우에만 Advice를 적용

스프링에서의 AOP

- Spring에서는 자체적으로 런타임 시에 위빙하는 "**프록시 기반의 AOP**"를 지원
  - 프록시 기반의 AOP는 메소드 호출 조인포인트만 지원
  - **Spring에서 어떤 대상 객체에 대해 AOP를 적용할 지의 여부는 설정 파일을 통해서 지정**
  - Spring은 설정 정보를 이용하여 런타임에 대상 객체에 대한 프록시 객체를 생성 따라서, 대상 객체를 직접 접근하는 것이 아니라 **프록시를 통한 간접 접근을 하게 됨**

Proxy

- **의미**

  프록시의 단어 자체로는 '대리인'이라는 의미.

  스프링 AOP에서의 프록시란 코드에 영향을 주지 않고 독립적으로 개발한 **부가 기능 모듈을** 다양한 핵심 기능의 객체에게 다이내믹하게 **적용해주기 위한 중요한 역할 담당**

- **스프링 AOP에서의 프록시**

  말 그대로 대리하여 업무를 처리하고, 메서드 호출자는 주요 업무가 아닌 보조 업무를 프록시에게 맡기고, 프록시는 내부적으로 이러한 보조 업무를 처리한다.

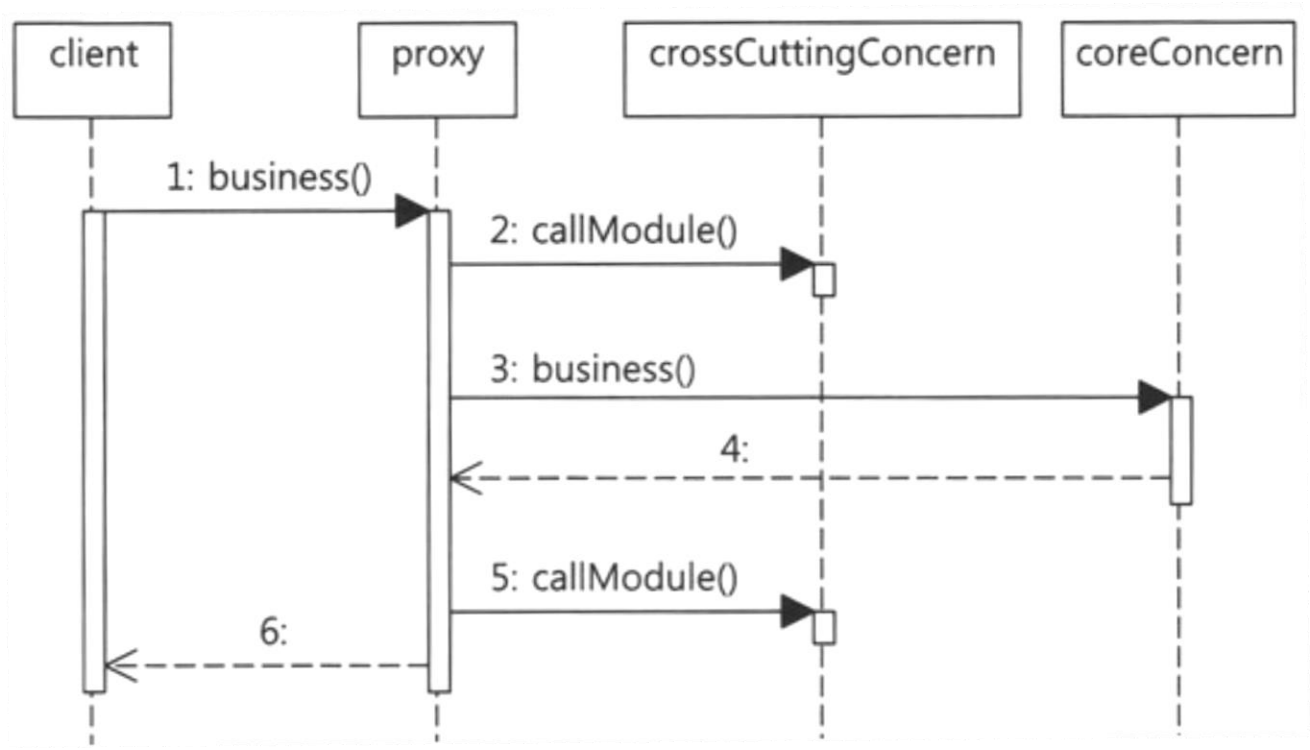
- **장점**

  Spring은 프록시 기술을 이용해 복잡한 빌드 과정이나 바이트코드 조작 기술 없이도 유용한 AOP를 적용할 수 있고, 변경 사항 발생시 애플리케이션 코드를 다시 컴파일 할 필요 없이 Aspect만 수정할 수 있어 편리하고 간단하게 사용 할 수 있음 .

  주 업무 코드는 보조 업무가 필요한 경우, 해당 Proxy 만 추가하면 되고, 필요 없게 되면 Proxy를 제거하면 됨.

  보조 업무의 탈 부착이 쉬워지고, 그리하여 주 업무 코드는 보조 업무 코드의 변경으로 인해서 발생하는 코드 수정 작업이 필요 없게 됨.

프록시 기반의 AOP 적용 process



- 런타임에 AOP를 적용할 때에는 소스 코드나 클래스 정보 자체를 변경하지 않고 프록시를 이용하여 AOP를 적용.
  프록시 기반의 AOP는 핵심 로직을 구현한 객체를 사용하기 위해서는 **프록시를 생성**하여 프록시를 통해서 핵심 로직을 구현한 객체를 활용할 수 있게 한다.

■ 플록시 예제

```
public interface Calculator {

        // 구현 해야 하는 메서드 선언 : 클래스의 구현 정의
        public long factorial(long num);

}
```

■ 플록시 예제

```
public class RecCalculator implements Calculator {

        // 인터페이스에서 정의되었던 추상메서드
        @Override
        public long factorial(long num) {
                if (num == 0)
                        return 1;
                else
                        return num * factorial(num - 1);
        }

}
```

■ 플록시 예제

```
public class ImpeCalculator implements Calculator {

        // 인터페이스에서 정의되었던 추상메서드
        @Override
        public long factorial(long num) {
                long result = 1;
                for (long i = 1; i <= num; i++) {
                        result *= i;
                }
                return result;
        }
}
```

플록시 예제

- 클래스의 실행 시간을 출력하는 프로그램을 작성하자.

■ 플록시 예제 :

```
public class ImpeCalculator implements Calculator {

        @Override
        public long factorial(long num) {
                // 현재 시간 기준에서 1970년 10월 1일 0시 0분 0초를 기준으로
밀리 초 값을 long타입으로 계산해서 반환
                long start = System.currentTimeMillis();
                long result = 1;
                for (long i = 1; i <= num; i++) {
                        result *= i;
                }
                long end = System.currentTimeMillis();
                // 실행 시간은 end-start 값으로 계산
                System.out.println("factorial("+num+") 실행시간 :"+(end-start));
                return result;
        }
}
```

■ 플록시 예제

```
public class RecCalculator implements Calculator {
        @Override
        public long factorial(long num) {
                // 현재 시간 기준에서 1970년 10월 1일 0시 0분 0초를 기준으로
밀리 초 값을 long타입으로 계산해서 반환
                long start = System.currentTimeMillis();
                try{
                if (num == 0)
                        return 1;
                else
                        return num * factorial(num - 1);
                } finally {
                long end = System.currentTimeMillis();
                // 실행 시간은 end-start 값으로 계산
                System.out.println("factorial("+num+") 실행시간 :"+(end-start));
                }
        }
}
```

■ 플록시 예제

```
// 플록시 구현
public class ExeTimeCalculator implements Calculator {

        private Calculator delegate;
        public ExeTimeCalculator(Calculator delegate) {
                this.delegate = delegate;
        }
        @Override
        public long factorial(long num) {
                // 가상머신의 기준 시점에서 경과 시간을 측정하는데 사용(ns 단위)
                // 시스템 시간과 무관
                long start = System.nanoTime();
                long result = delegate.factorial(num);
                long end = System.nanoTime();
                System.out.println("factorial("+num+") 실행시간 :"+(end-start));
                return result;
        }
}
```

Advice 개요

- **Advice란?**

  조인 포인트에 삽입되어 동작할 수 있는 공통 관심 사항의 코드

  동작시점 : Spring에서는 조인포인트 실행 전, 후로

  before, after, after returning, after throwing, around로 구분

Advice 개요

- **구현 가능한 Advice 종류**
  - **Before advice**

    joinpoint 전에 수행되는 advice
  - **After returning advice**

    joinpoint가 성공적으로 리턴 된 후에 동작하는 advice
  - **After throwing advice**

    예외가 발생하여 joinpoint가 빠져나갈 때 수행되는 advice
  - **After (finally) advice**

    join point를 빠져나가는(정상적이거나 예외적인 반환) 방법에 상관없이 수행되는 advice
  - **Around advice**

    joinpoint 전, 후에 수행되는 advice

Advice 개요

Advice 종 류	XML 스키마 기반의 POJO 클래스를 이용	@Aspect 어노테이션 기반	설 명
Before	<aop:before>	@Before	target 객체의 메소드 호출시 호출 전에 실행
AfterReturning	<aop:after-returning>	@AfterRetuning	target 객체의 메소드가 예외 없이 실행된 후 호출
AfterThrowing	<aop:after-throwing>	@AfterThrowing	target 객체의 메소드가 실행하는중 예외가 발생한 경우에 실행
After	<aop:after>	@After	target 객체의 메소드를 정상 또는 예외 발생 유무와 상관없이 실행 try의 finally와 흡사
Around	<aop:around>	@Around	target 객체의 메소드 실행 전, 후 또는 예외 발생 시점에 모두 실행해야 할 로직을 담아야 할 경우

스프링에서의 AOP

- **AOP 구현을 위한 방법**
  - **XML** 스키마 기반의 POJO 클래스를 이용한 AOP 구현
    - Spring API를 사용하지 않은 POJO 클래스를 이용하여 어드바이스를 개발하고 적용할 수 있는 방법이 추가
    - XML 확장을 통해 설정 파일도 보다 쉽게 작성 할 수 있음
  - **어노테이션** 기반의 AOP 구현

XML 기반의 POJO 클래스를 이용한 AOP 구현

XML 기반의 POJO 클래스를 이용한 AOP 구현

- **구현과정**

1. **관련 jar 파일을 클래스패스에 추가한다.**
   (이클립스의 경우는 레퍼런스로 추가하거나 마빈 프로젝트로 실행)
2. **Advice 클래스를 구현**
3. aop 네임스페이스가 있는 XML 설정 파일에 **Advice클래스를 빈으로 등록**
4. <aop:config> 태그를 이용해서 **Advice를 어떤 포인트컷에 적용할지 지정**

pom.xml

```
<dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.8.2</version>
</dependency>
```

XML 기반의 POJO 클래스를 이용한 AOP 구현

aspect.ExeTimeAspect.java

```
package aspect;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
//step2. Advice 클래스 구현
public class ExeTimeAspect {
        public Object measure(ProceedingJoinPoint joinPoint) throws Throwable {
                long start = System.nanoTime();
                try {
                        Object result=joinPoint.proceed();  //Advice가 적용될 객체 호출
                        return result;
                } finally {
                        long finish = System.nanoTime();
                        // 호출되는 메서드 정보를 구함.
                        Signature sig = joinPoint.getSignature();
                        System.out.printf("%s.%s 실행 시간 : %d ns₩n",
        joinPoint.getTarget().getClass().getSimpleName(), sig.getName(), (finish - start));
                }
        }
}
```

JoinPoint 사용

- Around Advice를 제외한 나머지 Advice를 구현한 메서드는 JoinPoint 객체를 선택적으로 전달 받을 수 있다.
- **Signature getSignature() – 호출되는 메서드에 대한 정보를 구한다.**
  - **String getName() – 메서드의 이름을 구함**
  - **String toLongString() – 메서드를 완전히 표현한 문장을 구한다.**
  - **String toShortString() – 메서드를 축약해서 표현한 문장을 구한다.**
- **Object getTarget() – 대상객체를 구한다.**
- **Object[] getArgs() – 파라미터 목록을 구한다.**

XML 기반의 POJO 클래스를 이용한 AOP 구현

aopPojo.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       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">

       <!-- 공통 기능을 제공할 클래스를 빈으로 등록 -->
       <bean id="exeTimeAspect" class="aspect.ExeTimeAspect" />
```

XML 기반의 POJO 클래스를 이용한 AOP 구현

aopPojo.xml

```
        <!-- Aspect 설정: Advice를 어떤 Pointcut에 적용할 지 설정 -->
        <aop:config>
                <!– Aspect 빈을 정의 ref="exeTimeAspect" -->
                <aop:aspect id="measureAspect" ref="exeTimeAspect">
                        <aop:pointcut id="publicMethod"
                                expression="execution(public * test..*(..))" />
        <!– test 페키지 및 그 하위 페키지 안에 있는 모든 public 메서드 -->
                        <!– 메서드 지정  -->
                        <aop:around pointcut-ref="publicMethod"
method="measure" />

                </aop:aspect>
        </aop:config>

        <bean id="impeCal" class="test.ImpeCalculator"> </bean>

        <bean id="recCal" class="test.RecCalculator"> </bean>

</beans>
```

XML 기반의 POJO 클래스를 이용한 AOP 구현

aopPojo.xml

```
package main;
import org.springframework.context.support.GenericXmlApplicationContext;
import test.Calculator;
public class MainXmlPojo {

	public static void main(String[] args) {
		GenericXmlApplicationContext ctx =
					new
GenericXmlApplicationContext("classpath:aopPojo.xml");

		Calculator impeCal = ctx.getBean("impeCal", Calculator.class);
		long fiveFact1 = impeCal.factorial(5);
		System.out.println("impeCal.factorial(5) = " + fiveFact1);

		Calculator recCal = ctx.getBean("recCal", Calculator.class);
		long fiveFact2 = recCal.factorial(5);
		System.out.println("recCal.factorial(5) = " + fiveFact2);
	}
}
```

AspectJ의 Pointcut 표현식

① 리턴 타입 지정

리턴타입 지정에서 가장 기본적인 방법은 '*'를 이용

표현식	설명
*	모든 리턴타입 허용
void	리턴타입이 void인 메서드만 허용
!void	리턴타입이 void가 아닌 메서드만 허용

② 패키지 지정

패키지 경로를 지정할 때는 '*', '..' 를 이용

표현식	설명
com.spring.service	정확하게 **com.spring.service** 패키지만 선택
com.spring.service..	**com.spring.service** 패키지로 시작하는 모든 패키지 선택
com.spring..impl	**com.spring.service** 패키지로 시작하면서 마지막 패키지 이름이 impl로 끝나는 패키지 선택

AspectJ의 Pointcut 표현식

③ 클래스 지정

클래스 이름을 지정할 때는 '*','+' 를 이용

표현식	설명
ListServiceImpl	정확하게 **ListServiceImpl** 패키지만 선택
***Impl**	클래스 이름이 **Impl**로 끝나는 클래스만 선택
ListService+	클래스 이름 뒤에 '+'가 붙으면해당 클래스로부터 파생되는 모든 하위 클래스 선택.

④ 메서드 지정

메서드를 지정할 때는 '*'를 주로 이용, 매개변수를 지정할 때는 '..' 를 이용

표현식	설명
***(..)**	가장 기본 설정으로 모든 메서드 선택
get*(..)	메서드 이름이 get으로 시작하는 모든 메서드 선택

AspectJ의 Pointcut 표현식

⑤ 매개변수 지정

매개변수를 지정할 때는 '..', '*' 를 이용 하거나 정확한 타입을 지정

표현식	설명
(..)	가장 기본적으로 '..'은 매개변수의 개수와 타입의 제약이 없음을 의미
(*)	반드시 1개의 매개변수를 가지는 메서드만 선택
(com.spring.member.MemberVo)	매개변수로 MemberVo를 가지는 메서드만 선택. 이때 패키지의 경로가 반드시 포함되어야 한다.
(!com.spring.member.MemberVo)	매개변수로 MemberVo를 가지지 않는 메서드만 선택
(Integer, ..)	한 개 이상의 매개변수를 가지되, 첫 번째 매개변수의 타입은 Integer인 메서드만 선택
(Integer, *)	반드시 두 개의 매개변수를 가지되, 첫 번째 매개변수 타입은 Integer인 메서드만 선택

AspectJ의 Pointcut 표현식

- **execution() 표현식**
  - 가장 대표적인 강력한 포인트컷의 지시자
  - 메소드의 signature를 이루는 접근 제한자, 리턴타입, 메소드명, 파라미터 타입, 예외타입 조건을 조합해서 메소드 단위까지 선택 가능한 가장 정교한 포인트컷 구성 가능.

- execution([**접근 제한자 패턴**] **타입패턴** [**타입패턴**.] **이름패턴** (타입패턴 | "..", ...) [throws 예외 패턴])
  - public과 같은 접근 제한자
  - 리턴 값의 타입
  - 패키지와 클래스 이름에 대한 패턴 사용할 때 '.'을 두어서 연결해야 함
  - 파라미터의 타입 패턴을 순서대로 넣을 수 있고, 와일드카드를 이용해 파라미터 개수에 상관없는 패턴을 만들 수 있음

AspectJ의 Pointcut 표현식

- **execution() 표현식 예시**

예	설명
execution(* hello(..))	리턴 타입과 파라미터는 상관 없이 메소드명이 hello인 모든 메소드를 선정하는 포인트컷 표현식
execution(* hello(int, int))	리턴 타입은 상관없이 메소드 이름이 hello 이며, 두개의 int 타입의 파라미터를 가진 모든 메소드를 선정하는 포인트컷 표현식
execution(* *(..))	리턴 타입과 파라미터의 종류, 개수에 상관없이 모든 메소드를 선정하는 포인트컷 표현식

AspectJ의 Pointcut 표현식

- **within() 표현식**
  - 특정 타입에 속하는 메소드를 포인트컷으로 설정할 때 사용
  - execution()의 여러 조건 중에서 타입 패턴만을 적용하기 때문에 execution 표현식 문법보다 간단
  - 단, 타깃 클래스의 타입에만 적용되며 조인포인트는 target 클래스 안에서 선언된 것만 선정됨
  - 선택된 타입의 모든 메소드가 AOP 적용 대상이 됨

AspectJ의 Pointcut 표현식

- **within() 표현식**

예	설명
within(spring.study.aop...*)	spring.study.aop 및 모든 서브패키지가 포함하고 있는 모든 메소드
within(spring.study.aop.*)	spring.study.aop 패키지 밑의 인터페이스와 클래스에 있는 모든 메소드
within(spring.study.aop.Hello)	spring.study.aop 패키지의 Hello클래스의 모든 메소드