

목차

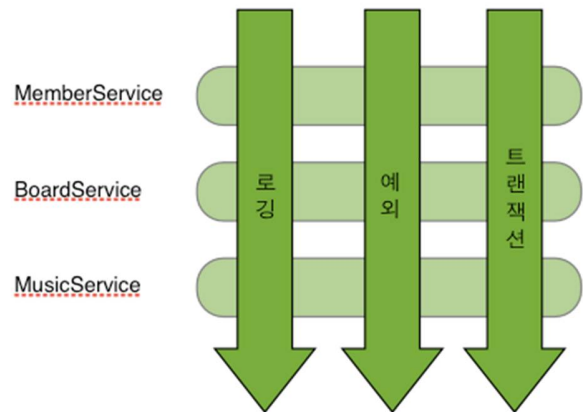
1. AOP(Aspect Oriented Programming) 란?	2
● 공통관심사항 (cross-cutting concern)	2
2. AOP 용어	4
● JoinPoint	4
● Pointcut	4
● Advice	5
Advice 타입	5
● Aspect	6
● Target	6
● proxy	6
3. AOP Weaving	7
● 컴파일시 Weaving → 컴파일시 코드 삽입	7
● 클래스 로딩시 Weaving → 로딩시 코드 삽입	7
● 런타임시 Weaving → 런타임 Proxy 생성	7
4. Spring AOP 냐? AspectJ 냐?	8
5. PointCut(어디에, 어떤 메서드) 정의 예제	9
6. 스프링에서의 AOP	11
● 스프링의 AOP 설정 방식	11
● XML 을 이용한 AOP 설정	11
● 어노테이션(@Aspect)을 이용한 AOP 설정	15
7. Reference	16
8. Intercept static methods using AOP	17
9. Load-time weaving with AspectJ in the Spring Framework	19

1. AOP(Aspect Oriented Programming) 란?

문제를 바라보는 관점을 기준으로 프로그래밍하는 기법

- 메서드 호출시의 **코드 삽입** 기법

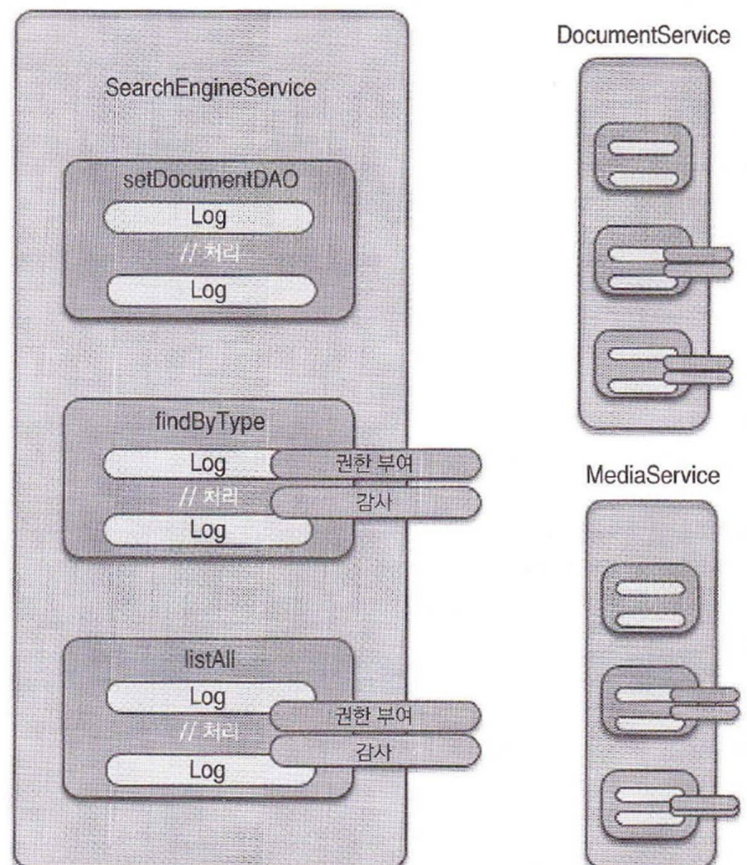
- 문제를 해결하기 위한 **핵심 관심 사항(비즈니스 로직)**과 전체에 적용되는 **공통 관심 사항**을 기준으로 프로그래밍 함으로써 공통 모듈을 여러 코드에 쉽게 적용
 - core concern(핵심 관심 사항)
 - 비즈니스 로직
 - cross-cutting concern(공통 관심 사항)
 - 로깅, 트랜잭션, 예외처리, 보안, 감사, 에러처리, 동기화



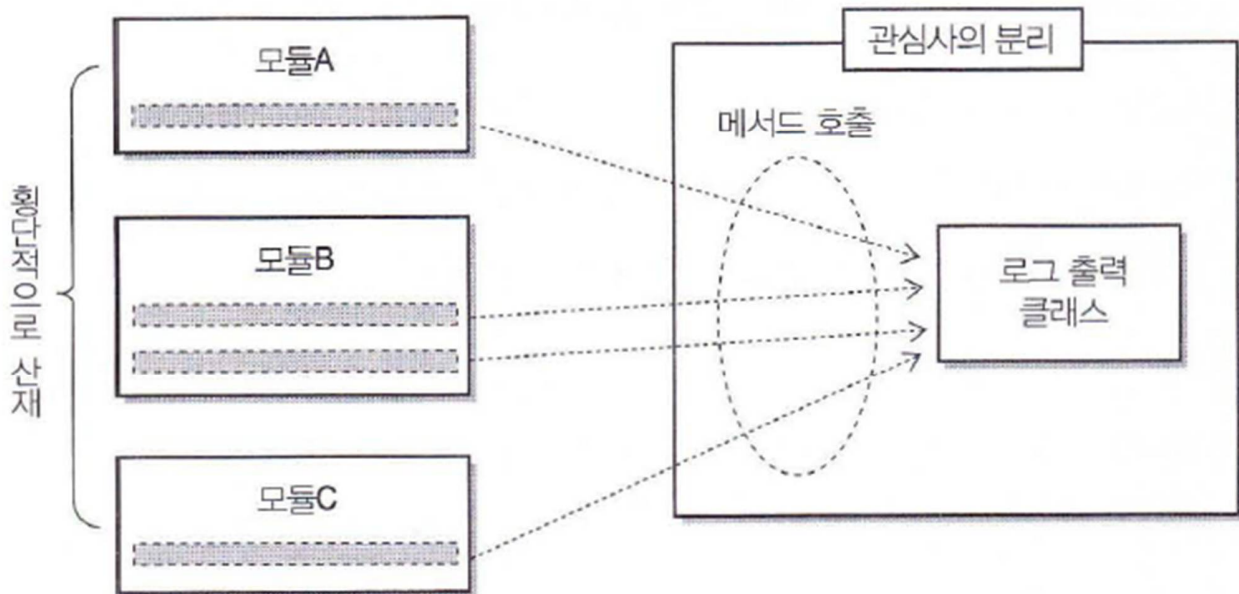
- **공통관심사항 (cross-cutting concern)**

어플리케이션에서 로깅과 같은 기본적인 기능에서부터 트랜잭션이나 보안과 같은 기능에 이르기까지 어플리케이션 전반에 걸쳐 적용되는 공통 기능

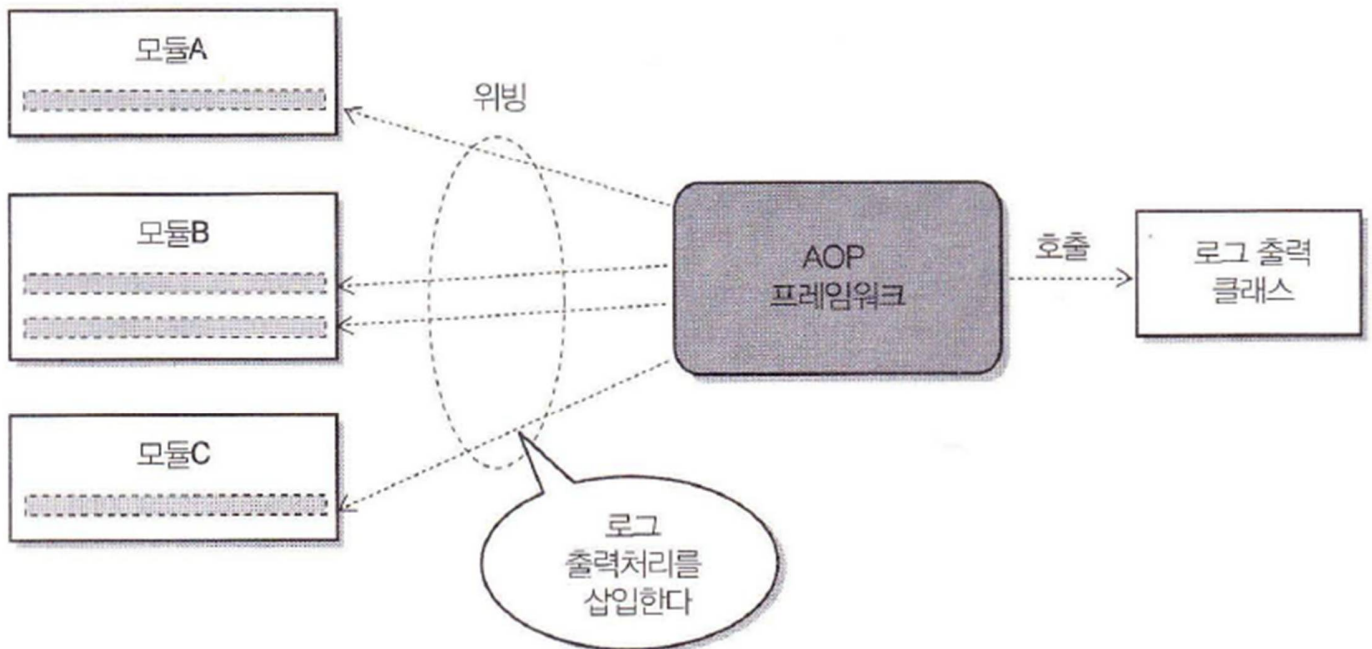
- 로깅(Logging)
- 트랜잭션 관리
- 예외처리
- 보안체크
- 감사체크



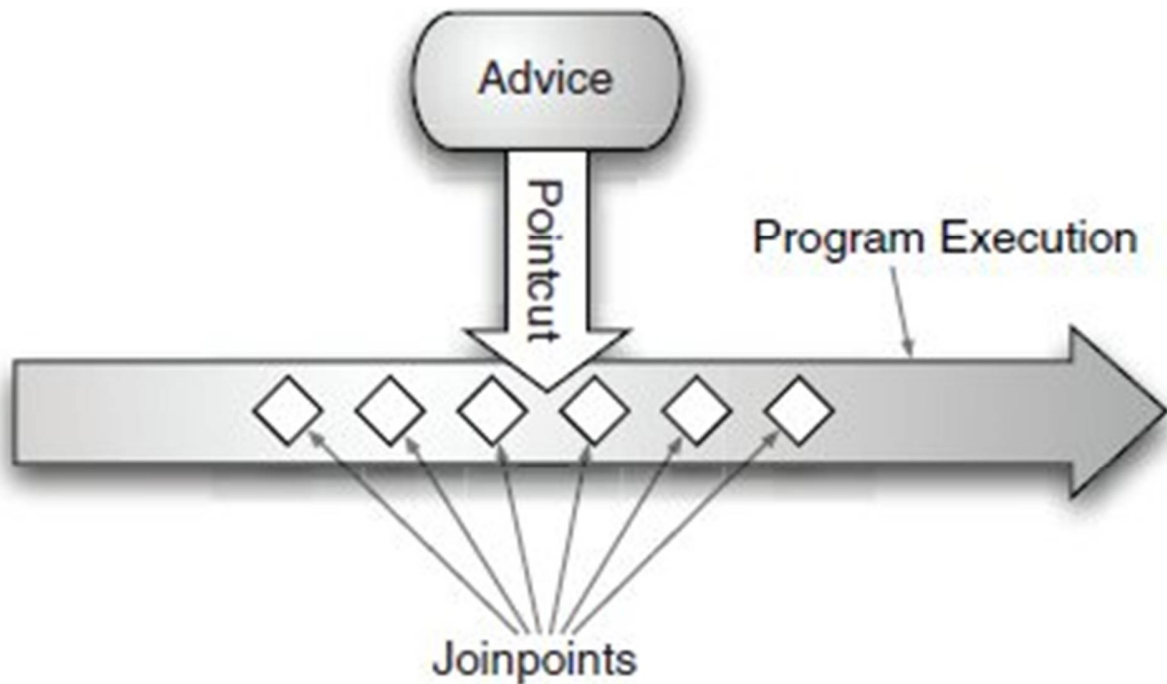
■ 횡단적으로 산재하는 기능 호출



■ AOP에서 횡단적 관심사의 분리와 위빙



2. AOP 용어



JointPoint 의 PointCut 의 Advice 를 Aspect 해~~

JoinPoint	동작 장소. 어디에 . 실행 지점
Advice(Advisor)	동작 코드. 무엇을 . 실행 액션
PointCut	동작 조건. 어떻게 . 결합 패턴(표현식 언어)
Aspect	Pointcut 과 Advice 의 집합

● JoinPoint

Advice 를 적용 가능한 지점을 의미한다. 인스턴스 생성 시점, '메소드 호출 시점', '예외 발생 시점'과 같이 어플리케이션을 실행할 때 특정 작업이 시작되는 시점을 '조인포인트'라고 한다. 구체적으로는 메서드 호출이나 예외발생이라는 포인트를 Joinpoint 라고 한다.

● Pointcut

Joinpoint 의 부분 집합으로서 실제로 Advice 가 적용되는 Jointpoint 를 나타낸다. 스프링에서는 정규 표현식이나 AspectJ 문법을 이용하여 Pointcut 을 정의할 수 있다.

하나 또는 복수의 Jointpoint 를 하나로 묶은 것을 Pointcut 이라고 한다. Advice 의 위빙 정의는 Pointcut 을 대상으로 설정한다. 하나의 Pointcut 에는 복수 Advice 를 연결할 수 있다. 반대로 하나의 Advice 를 복수 Pointcut 에 연결하는 것도 가능하다.

st05.스프링은 AOP 를 지원한다

Pointcut(교차점)은 JoinPoint(결합점)들을 선택하고 결합점의 환경정보를 수집하는 프로그램의 구조물이다.

● Advice

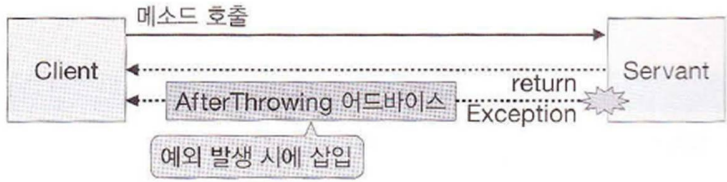
언제 공통 관심 기능을 핵심 로직에 적용할 지를 정의하고 있다.

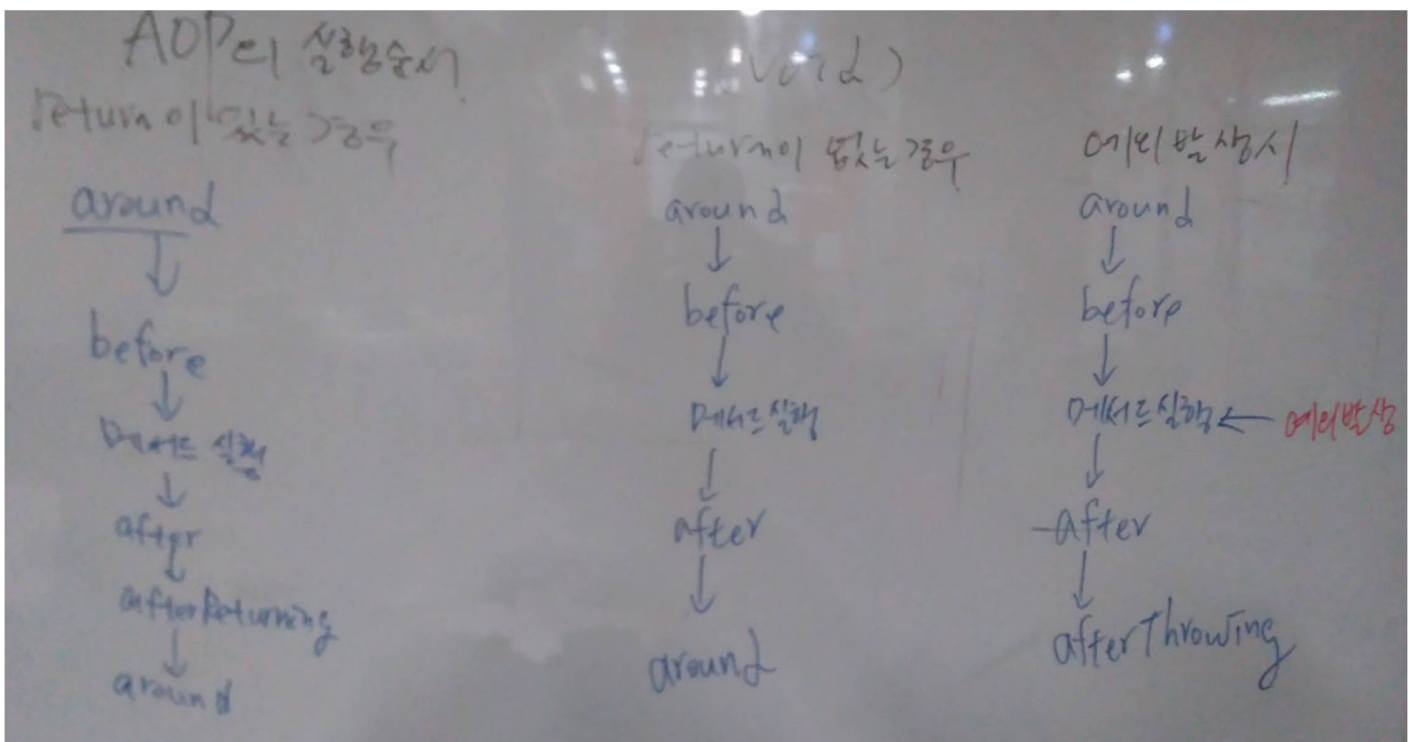
Advice 를 어디에서 위빙하는 지는 PointCut 이라는 단위로 정의한다.

Advice 타입

Advice 타입	설명	Annotation
Before	<p>Joinpoint 앞에서 실행할 Advice org.springframework.aop.MethodBeforeAdvice</p>	@Before
After	<p>Joinpoint 뒤에서 실행할 Advice org.springframework.aop.AfterAdvice</p>	@After
Around	<p>Joinpoint 앞과 뒤에서 실행되는 Advice org.springframework.aop.MethodInterceptor</p>	@Around
After Returning	<p>Joinpoint 가 정상 종료한 다음에 실행되는 Advice org.springframework.aop.AfterReturningAdvice</p>	@AfterReturning

st05.스프링은 AOP 를 지원한다

After Throwing	<p>Jointpoint 에서 예외가 발생했을 때 실행되는 Advice org.springframework.aop.ThrowsAdvice</p>  <pre> sequenceDiagram participant Client participant Servant Client->>Servant: 메소드 호출 Servant-->>Client: return Servant-->>Client: Exception Note over Servant: AfterThrowing 어드바이스 Note over Servant: 예외 발생 시에 삽입 </pre>	@AfterThrowing
Introduction	<p>클래스에 인터페이스와 구현을 추가하는 특수한 Advice org.springframework.aop.IntroductionInterceptor</p>	



● Aspect

여러 객체에 공통으로 적용되는 공통 관심 사항을 Aspect 라고 한다. 로깅, 트랜잭션이나 보안 등이 Aspect 의 좋은 예이다.

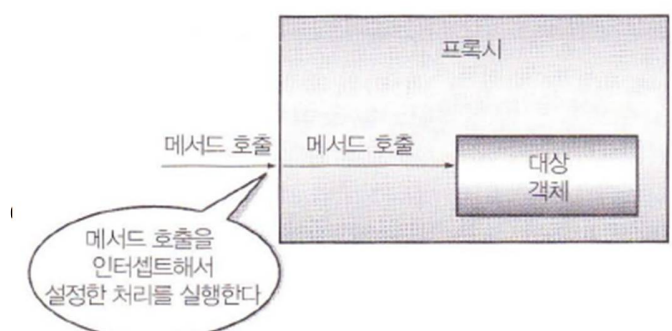
Aspect 는 AOP 의 중심 단위로 Advice 와 PointCut 을 합친 것이다. Aspect 를 Advisor 라고도 한다.

● Target

핵심 로직을 구현하는 클래스를 말한다. 충고를 받는 클래스를 대상(target)이라고 한다. 대상은 여러분이 작성한 클래스는 물론, 별도의 기능을 추가하고자 하는 써드 파티 클래스가 될 수 있다.

● proxy

대상 객체에 Advice 가 적용된 후 생성된 객체



3. AOP Weaving

Weaving 이란 코드가 삽입되는 행위를 Weaving 이라고 한다.

Weaving 방식에는

- 컴파일시 코드 Weaving
- 로딩시 코드 Weaving : 프로그램 실행시 weaving
- 런타임시 코드 Weaving : 메서드 호출시 weaving. 스프링에서 사용하는 방식.

- **컴파일시 Weaving → 컴파일시 코드 삽입**

대표적인 AOP 지원툴인 **AspectJ** 가 사용하는 방식이다. 컴파일시 핵심로직을 구현한 소스에 알맞은 위치에 공통 기능 코드를 삽입하게 된다.

- **클래스 로딩시 Weaving → 로딩시 코드 삽입**

JVM 이 클래스를 로딩할 때 클래스 정보를 변경할수 있는 에이전트를 제공함으로써 클래스의 바이너리 정보를 변경하여 알맞은 위치에 공통 코드를 삽입한 새로운 클래스 바이너리 코드를 사용하도록 한다. 즉 원본 클래스 파일은 변경하지 않고 클래스를 로딩할 때 JVM 이 변경된 바이트 코드를 사용한다. **AspectWerkz** 가 이에 해당한다.

- **런타임시 Weaving → 런타임 Proxy 생성**

런타임시 Weaving 은 프록시를 이용해 AOP 를 적용한다. 즉 핵심 로직을 구현한 객체에 직접 접근하지 않고 중간에 공통코드가 적용된 **프록시**를 생성하여 **프록시**를 통해 핵심 로직에 접근하게 된다. **Spring AOP** 가 이에 해당한다.

4. Spring AOP 냐? AspectJ 냐?

작업 할 수 있는 가장 간단한 것을 사용해라. 스프링 AOP 는 개발이나 빌드 프로세스에 AspectJ 컴파일러/위버(weaver)를 도입해야 하는 요구사항이 없으므로 완전한 AspectJ 를 사용하는 것보다 간단하다. 스프링 빈에서 작업의 실행을 어드바이즈 하는 것이 필요한 것의 전부라면 스프링 AOP 가 좋은 선택이다. 스프링 컨테이너가 관리하지 않는 객체(보통 도메인 객체 같은)를 어드바이즈 해야 한다면 AspectJ 를 사용해야 할 것이다. 간단한 메서드 실행외에 조인포인트를 어드바이즈해야 한다면 마찬가지로 AspectJ 를 사용해야 할 것이다.(예를 들면 필드를 가져오거나 조인포인트를 설정하는 등)

AspectJ 를 사용하는 경우 AspectJ 언어 문법("code style"이라고도 알려진)과 @AspectJ 어노테이션 방식의 선택권이 있다. 알기 쉽게 자바 5 이상을 사용하지 않는다면 code style 을 사용해야 한다. 설계상 관점이 커다란 역할을 하고 AspectJ Development Tools (AJDT) 이클립스 플러그인을 사용할 수 있다면 AspectJ 언어의 문법을 사용하는 것이 더 바람직하다. AspectJ 언어가 관점을 작성하기 위한 목적으로 설계된 언어이기 때문에 더 깔끔하고 간단하다. 이클립스를 사용하지 않거나 어플리케이션에서 주요 역할을 하지 않는 약간의 관점만 가지고 있다면 IDE 에서 일반적인 자바 컴파일과 @AspectJ 방식을 사용하고 빌드 스크립트에 관점을 위빙하는 단계를 추가하는 것을 고려해라.

5. PointCut(어디에, 어떤 메서드) 정의 예제

Spring AOP 에서 자주 사용되는 PointCut 표현식의 예를 살펴본다.

PointCut	선택된 JoinPoints
<code>execution(public * *(..))</code>	public 으로 시작하는 모든 메소드에 실행
<code>execution(* set*(..))</code>	set 로 시작하는 모든 메소드에 실행
<code>execution(* com.xyz.service.AccountService.*(..))</code>	AccountService 의 모든 메소드 실행
<code>execution(* com.xyz.service.*.*(..))</code>	service 패키지의 모든 메소드 실행
<code>execution(* com.xyz.service..*.*(..))</code>	service 패키지와 하위 패키지의 모든 메소드 실행
<code>within(com.xyz.service.*)</code>	service 패키지 내의 모든 결합점
<code>within(com.xyz.service..*)</code>	service 패키지 및 하위 패키지의 모든 결합점
<code>this(com.xyz.service.AccountService)</code>	AccountService 인터페이스를 구현하는 프록시 객체의 모든 결합점
<code>target(com.xyz.service.AccountService)</code>	AccountService 인터페이스를 구현하는 대상 객체의 모든 결합점
<code>args(java.io.Serializable)</code>	하나의 파라미터를 갖고 전달된 인자가 Serializable 인 모든 결합점

Pointcut	선택된 Joinpoints
<code>bean(accountRepository)</code>	"accountRepository" 빈
<code>!bean(accountRepository)</code>	"accountRepository" 빈을 제외한 모든 빈
<code>bean(*)</code>	모든 빈
<code>bean(account*)</code>	이름이 'account'로 시작되는 모든 빈
<code>bean(*Repository)</code>	이름이 "Repository"로 끝나는 모든 빈
<code>bean(accounting/*)</code>	이름이 "accounting/"로 시작하는 모든 빈
<code>bean(*dataSource) bean(*DataSource)</code>	이름이 "dataSource" 나 "DataSource" 으로 끝나는 모든 빈

st05.스프링은 AOP 를 지원한다

Pointcut	선택된 Joinpoints
@target(org.springframework.transaction.annotation.Transactional)	대상 객체가 @Transactional 어노테이션을 갖는 모든 결합점
@within(org.springframework.transaction.annotation.Transactional)	대상 객체의 선언 타입이 @Transactional 어노테이션을 갖는 모든 결합점
@annotation(org.springframework.transaction.annotation.Transactional)	실행 메소드가 @Transactional 어노테이션을 갖는 모든 결합점
@args(com.xyz.security.Classified)	단일 파라미터를 받고, 전달된 인자 타입이 @Classified 어노테이션을 갖는 모든 결합점

6. 스프링에서의 AOP

스프링은 자체적으로 프록시 기반의 AOP 를 지원한다. 따라서 스프링 AOP 는 메서드 호출 Joinpoint 만을 지원하고 필드값 변경 같은 Joinpoint 를 사용하기 위해서는 AspectJ 와 같이 풍부한 기능을 지원하는 AOP 도구를 사용해야 한다.

● 스프링의 AOP 설정 방식

- XML 을 이용한 AOP 설정
- 어노테이션(@Aspect)을 이용한 AOP 설정
- API 를 이용한 AOP 설정(많이 사용하지는 않음)

● XML 을 이용한 AOP 설정

<aop:config />	aop 설정을 포함.
<aop:aspect />	Aspect 를 설정.
<aop:pointcut />	Pointcut 을 설정.
<aop:before />	메서드 실행전에 적용되는 Advice 를 정의
<aop:after-returning />	메서드가 정상적으로 실행된 후에 적용되는 Advice 를 정의
<aop:after-throwing />	예외를 발생시킬 때 적용되는 Advice 를 정의 (catch 블록과 유사)
<aop:after />	예외 여부와 상관없는 Advice 를 정의한다 (finally 블록과 유사)
<aop:around />	메서드 호출 이전, 이후, 예외 발생 등 모든 시점에서 적용 가능한 Advice 를 정의

```

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

  xmlns:aop="http://www.springframework.org/schema/aop"

  xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.xsd

  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd">

  <!--핵심기능 -->
  <beans:bean name="myBean" class="com.lecture.spring.bean.MyBean" />

  <!-- 공통관심사항 -->
  <beans:bean id="aspectBean" class="com.lecture.spring.aspect.MyAspect" />

  <aop:config>
    <aop:aspect id="myAspect" ref="aspectBean">
      <aop:before pointcut="execution(* print*(..))" method="beforeMessage" />
      <aop:after-returning pointcut="execution(* print*(..))"
method="afterMessage" />
    </aop:aspect>
  </aop:config>

</ beans: beans>

```

```

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

  xmlns:aop="http://www.springframework.org/schema/aop"

  xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
  http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd

  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd">

  <beans:bean id="myFirstAOP" class="aop01.aop.FirstAspect" />

  <aop:config>
    <aop:aspect id="myAspect" ref="myFirstAOP">

      <!-- get 으로 시작하는 모든 메서드에 적용하겠다. -->
      <aop:pointcut id="pc" expression="execution( * get*(*))" />

      <aop:after          method="after"          pointcut-ref="pc" />
      <aop:after-returning method="afterReturning" pointcut-ref="pc"
returning="product" />
      <aop:after-throwing  method="afterThrowing"  pointcut-ref="pc"
throwing="e" />
      <aop:around          method="around"         pointcut-ref="pc" />
      <aop:before          method="before"         pointcut-ref="pc" />
    </aop:aspect>
  </aop:config>

</beans:beans>

```

```
public class ProfilingAdvice {  
    public Object trace(ProceedingJoinPoint joinPoint) throws Throwable{  
  
        // 전처리  
        String signatureString = joinPoint.getSignature().toShortString();  
        System.out.println(signatureString + "시작");  
        long start = System.currentTimeMillis();  
  
        try{  
            Object result = joinPoint.proceed();  
            return result;  
        }finally{  
            // 후처리  
            long finish = System.currentTimeMillis();  
            System.out.println(signatureString + "종료");  
            System.out.println(signatureString + "실행 시간 : " + (finish - start) + "ms");  
        }  
    }  
}
```