Spring Framework

Spring AOP

❑ Separation of Concerns

- Separation of Concerns, 즉 "관심사의 분리"라는 명제를 가지고 프로그램에서는 많은 연구와 개발방법론이 등장합니다. 앞으로도 계속 새로운 기술들이 등장 할 것입니다. 이해 하기 쉽게 말하면 MVC(Model-View-Controller) 또한 관심사의 분리라고 보시면 됩니다. View는 디자이너, Controller, Model 은 개발자, 또는 5계층이라고 해서 Business/Logic Layer, Persistence Layer, Presentation Layer, Controller Layer, Domain Layer 또한 관심사의 분리하고 할 수 있습니다.
- AOP에서 concern 이란 프로그램 개발을 위해 상호 작용하는 객체들의 집합으로 보며 클래스를 주요 단위로 합니다. 간단하게 말하자면 어떠한 모듈단위라고 생각하면 됩니다. (비즈니스 로직, 로깅, 인증, 보안, 트랜잭션, 예외처리 등)
- 단어 concern :
  - 명사 : 1. (특히 많은 사람들이 공유하는) 우려[걱정]. 2. * 중요한 것, 관심사. 3. (책임이나 알 권리가 있는) 일
  - 동사: 1. (사람에게) 영향을 미치다[관련되다]. 2. (무엇에) 관한[관련된] 것이다.
    3. ...를 걱정스럽게[우려하게] 만들다. 4. ~에 흥미[관심]를 갖다

❑ AOP (Aspect Oriented Programming) 개요

- 객체지향 프로그래밍은 많은 장점에도 불구하고, 다수의 객체들에 분산되어 중복적으로 존재하는 공통 관심사가 존재합니다. 이들은 프로그램을 복잡하게 만들고, 코드의 변경을 어렵게 합니다. 관점 지향 프로그래밍(AOP, Apsect-oriented programming)은 이러한 객체지향 프로그래밍의 문제점을 보완하는 방법으로 핵심 관심사를 분리하여 프로그램 모듈화를 향상시키는 프로그래밍 스타일입니다.
- Core(Primary) concern(핵심 관심사)은 일반적으로 구현하려고 하는 핵심 비즈니스 기능을 말한다.
- Cross-cutting concern(횡단 관심사)은 보안, 로깅, 트랜잭션, 인증과 같이 시스템 관리를 위한 기능들을 말한다.
- AOP는 Cross-cutting concern를 어떻게 다룰 것인가에 대한 새로운 패러다임이라고 할 수 있다.
- 단어 Aspect :
  - 명사 : 1. 측면, 양상. 2. * (문제를 보는) 관점. 3. (건물・땅 등의) 방향; (건물의 특정 방향을 향한) 면
- AOP는 객체를 Core concern과 Cross-cutting concern으로 분리하고, 횡단 관심사를 관점(Aspect)이라는 모듈로 정의하고 핵심 관심사와 엮어서(Weaving) 처리할 수 있는 방법을 제공합니다.

❑ AOP 장점

- AOP는 각 concern들의 연관성을 최소화하여 cross-cutting concern들을 분리한다. concern들을 분리함으로써 얻는 장점은 다음과 같을 것이다.
- 중복 코드의 제거
  - 횡단 관심(CrossCutting Concerns)을 여러 모듈에 반복적으로 기술되는 현상을 방지
- 비즈니스 로직의 가독성 향상
  - 핵심기능 코드로부터 횡단 관심 코드를 분리함으로써 비즈니스 로직의 가독성 향상
- 생산성 향상
  - 비즈니스 로직의 독립으로 인한 개발의 집중력을 높임
- 재사용성 향상
  - 횡단 관심 코드는 여러 모듈에서 재사용될 수 있음
- 변경 용이성 증대
  - 횡단 관심 코드가 하나의 모듈로 관리되기 때문에 이에 대한 변경 발생시 용이하게 수행할 수 있음

❑ AOP 주요 용어

- AOP에서 사용되는 용어중 다음에 등장하는 용어 및 의미를 잘 숙지하시기 바랍니다.
- Joinpoint
  - class의 인스턴스 생성 시점, 메소드를 호출하는 시점, Exception이 발생하는 시점과 같이 어플리케이션을 실행할 때 특정 작업이 실행되는 시점을 의미한다. 메소드, 예외, 변경 가능한 필드 등에 대해 joinpoint가 정의될 수 있다.
  - Joinpoint는 Aspect를 삽입하여 실행 가능한 어플리케이션의 특정 지점을 말한다.
  - Spring AOP는 메서드 실행 joinpoint 밖에 없다.
- Pointcut
  - 모든 Joinpoint에 Cross-cutting Concern 할 필요는 없다. 원하는 위치를 지정해야 한다.
  - 분리된 기능들을 결합시키기 위한 규칙이 필요하다. Target class와 Advice가 결합(Weaving)될 때 둘 사이의 결합 규칙을 정의하는 것이다. Advice가 어떤 Joinpoint에 적용돼야 하는지를 정의한다.
- Advice
  - Advice는 Aspect의 실제 구현체로 Joinpoint에 삽입되어 동작할 수 있는 코드이다. Advice는 Joinpoint와 결합하여 동작하는 시점에 따라 before advice, after advice, around advice 타입으로 구분된다.
    1. Before advice: joinpoint 전에 수행되는 advice
    2. After returning advice: joinpoint가 성공적으로 리턴된 후에 동작하는 advice
    3. After throwing advice: 예외가 발생하여 joinpoint가 빠져나갈때 수행되는 advice
    4. After (finally) advice: join point를 빠져나가는(정상적이거나 예외적인 반환) 방법에 상관없이 수행되는 advice
    5. Around advice: joinpoint 전, 후에 수행되는 advice
- Aspect
  - Aspect는 구현하고자 하는 Cross-cutting Concern이다.
  - Advice와 Pointcut을 합쳐서 하나의 Aspect라고 한다
- Advisor
  - Spring AOP에서만 사용되는 용어이다. Aspect와 유사한 의미이다.
- Introduction
  - Introduction은 새로운 메소드나 속성을 추가한다. Spring AOP는 충고(Advice)를 받는 대상 객체에 새로운 인터페이스를 추가할 수 있다.
- Target
  - Target 객체는 Advice를 받는 객체이다. 핵심 비즈니스 로직이 있는 객체라고 생각하면 된다.
- Weaving
  - Weaving은 Aspect를 Target 객체에 적용하여 새로운 프록시 객체를 생성하는 과정이다. Weaving은 다음과 같이 구분된다.
    1. 컴파일 시 엮기: 별도 컴파일러를 통해 핵심 관심사 모듈의 사이 사이에 관점(Aspect) 형태로 만들어진 횡단 관심사 코드들이 삽입되어 관점(Aspect)이 적용된 최종 바이너리가 만들어지는 방식이다. (ex. AspectJ, AspectWerkz, ...)
    2. 클래스 로딩 시 엮기: 별도의 Agent를 이용하여 JVM이 클래스를 로딩할 때 해당 클래스의 바이너리 정보를 변경한다. 즉, Agent가 횡단 관심사 코드가 삽입된 바이너리 코드를 제공함으로써 AOP를 지원하게 된다. (ex. AspectJ, AspectWerkz, ...)
    3. 런타임 엮기: 소스 코드나 바이너리 파일의 변경 없이 프록시를 이용하여 AOP를 지원하는 방식이다. 프록시를 통해 핵심 관심사를 구현한 객체에 접근하게 되는데, 프록시는 핵심 관심사 실행 전후에 횡단 관심사를 실행한다. 따라서 프록시 기반의 런타임 엮기의 경우 메소드 호출 시에만 AOP를 적용할 수 있다는 제한점이 있다. (ex. Spring AOP, JBossAOP, ..)
- Proxy
  - Proxy는 Target객체에 Advice가 적용된 후 생성되는 객체이다. Spring은 프록시를 통해 AOP를 지원한다.

❑ Spring의 AOP 구현

- 스프링은 프록시기반의 런타임 Weaving 방식만을 지원한다. 또한 스프링은 AOP 구현을 위해 다음 세가지 방식을 제공한다.
  1. Spring API를 이용한 AOP 구현
  2. XML Schema를 이용한 AOP 구현
  3. @AspectJ 어노테이션을 이용한 AOP 구현

- Spring API를 사용한 AOP 를 제공하지만 실제적으로 많이 사용되지는 않는다. 하위버전 호환성 등.
- Proxy 를 이용한 AOP 구현
  - 스프링은 대상(target)이 되는 객체에 프록시를 만들어 제공한다. 대상객체에 직접 접근하기보다는 프록시를 통해서 간접적으로 접근하게 된다. 이 과정에서 프록시는 대상객체의 메서드 실행 전.후에 실행하게 된다.
  - 프록시객체 생성 방식은 Interface구현 여부에 따라 다르다
    1. 대상객체가 인터페이스를 구현했다면 자바리플랙션(java.lang.reflect) API가 제공하는 Proxy 를 이용하여 생성한다. 해당 프록시객체는 target과 동일한 인터페이스를 구현, 필요한 메서드 호출, 즉 인터페이스에 정의되지 않은 메서드는 호출 불가능
    2. 대상객체가 인터페이스를 구현하지 않았다면 CGLIB를 이용하여 프록시 생성한다. CGLIB는 대상 클래스를 상속받아 프록시를 구현한다. 따라서 대상객체가 final인 경우 프록시를 생성할 수 없으며, 메서드가 final이라면 AOP 적용 불가능

❑ Dependency Library

- Spring에서 제공하는 API만으로도 AOP를 적용가능하지만 실제적으로 java AOP 프레임워크 중에 AspectJ가 구현, 기능면에서 앞서고 있기 때문에 AspectJ 에서 제공하는 인터페이스를 많이 사용합니다. 요즘에는 BEA가 지원하는 AspectWerkz 가 자바표준 클래스를 이용해서 AOP를 구현되며 기능도 AspectJ와 거의 동일하다.
- 또한 Spring AOP는 Interface에 대한 AOP프록시 객체만을 생성하기 때문에 인터페이스를 구현하지 않은 일반 클래스에 대한 프록시 생성을 할수 있는 CGLIB(Code Generation Library)도 필요합니다.
  - AOP Alliance : http://aopalliance.sourceforge.net/
  - AspectJ : http://www.eclipse.org/aspectj/
  - CGLIB : http://cglib.sourceforge.net/
  - 각 필요한 파일은 aopalliance.jar, aspectj-weaver.jar, cglig-nodep-2.x.x.jar

AspectJ : 원래 xerox의 PARC(Palo Alto Research Center)에서 개발된 AspectJ는 AOP 프로그래밍 기법을 Java에서 구현되도록 만든것이며 나중에 오픈소스가 되었고, 현재는 www.eclipse.org/aspectJ 에서 관리한다.
AOP에 대한 초기 글 중 하나인 "Aspect Oriented Programming"(PDF)글 에서 AspectJ 개발자들은 35,000 라인으로 만들었던 성능 최적화에 관해 기술된 이 코드를 AOP로 다시 작성한 결과 코드를 1,000 라인으로 줄일 수 있었다고 한다.

❑ Pontcut 표현식

- Pointcut은 JoinPoint에서 패턴매칭을 통해 필요한 지점을 선택하는 것이다.
- Pointcut 지정자
  - execution: 메소드 실행 결합점(join points)과 일치시키는데 사용된다.
  - within: 특정 타입에 속하는 결합점을 정의한다.
  - this: 빈 참조가 주어진 타입의 인스턴스를 갖는 결합점을 정의한다.
  - target: 대상 객체가 주어진 타입을 갖는 결합점을 정의한다.
  - args: 인자가 주어진 타입의 인스턴스인 결합점을 정의한다.
  - @target: 수행중인 객체의 클래스가 주어진 타입의 어노테이션을 갖는 결합점을 정의한다.
  - @args: 전달된 인자의 런타입 타입이 주어진 타입의 어노테이션을 갖는 결합점을 정의한다.
  - @within: 주어진 어노테이션을 갖는 타입 내 결합점을 정의한다.
  - @annotation: 결합점의 대상 객체가 주어진 어노테이션을 갖는 결합점을 정의한다.

- Pointcut 표현식 조합
  - '&&„ : anyPublicOperation() && inTrading()
  - '||' : bean(*dataSource) || bean(*DataSource)
  - '!' : !bean(accountRepository)
  - XML 설정파일에서는 "&&"를 "&&" 또는 and, or 사용 가능

- execution 을 이용한 pointcut 을 많이 사용하며 문법은다음과 같다

```
execution( [접근제한자] 리턴타입 [클래스이름.]메서드이름(파라미터타입|..)
```

  - 대괄호는 옵션이다.
  - "*" 모든 문자 매칭
  - ".." 하위폴더, 여러 파라미터 인자

❑ Pointcut 예시

- Spring AOP에서 자주 사용되는 포인트컷 표현식의 예를 살펴본다. (전자정부)

Pointcut	선택된 Joinpoints
execution(public * *(..))	public 메소드 실행
execution(* set*(..))	이름이 set으로 시작하는 모든 메소드명 실행
execution(* com.xyz.service.AccountService.*(..))	AccountService 인터페이스의 모든 메소드 실행
execution(* com.xyz.service.*.*(..))	service 패키지의 모든 메소드 실행
execution(* com.xyz.service..*.*(..))	service 패키지와 하위 패키지의 모든 메소드 실행
within(com.xyz.service.*)	service 패키지 내의 모든 결합점
within(com.xyz.service..*)	service 패키지 및 하위 패키지의 모든 결합점
this(com.xyz.service.AccountService)	AccountService 인터페이스를 구현하는 프록시 개체의 모든 결합점
target(com.xyz.service.AccountService)	AccountService 인터페이스를 구현하는 대상 객체의 모든 결합점
args(java.io.Serializable)	하나의 파라미터를 갖고 전달된 인자가 Serializable인 모든 결합점
@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
bean(accountRepository)	"accountRepository" 빈
!bean(accountRepository)	"accountRepository" 빈을 제외한 모든 빈
bean(*)	모든 빈
bean(account*)	이름이 'account'로 시작되는 모든 빈
bean(*Repository)	이름이 "Repository"로 끝나는 모든 빈
bean(accounting/*)	이름이 "accounting/"로 시작하는 모든 빈
bean(*dataSource) || bean(*DataSource)	이름이 "dataSource" 나 "DataSource" 으로 끝나는 모든 빈

❑ Advice 시점 (정의)

- Advice는 Pointcut에 매칭되는 지점에 동작하는 모듈이다. 이때 핵심 모듈 실행 전,후 등의 시점을 선택할 수 있다.
- before advice
  - Before advice는 target 메서드 실행 전에 동작한다.
  - Advice 메서드에서 에러가 나는 경우 target 메소드가 호출되지 않는다.
  - <aop:before /> 사용
  - @Before 어노테이션을 사용
- After returning advice
  - After returing advice 는 정상적으로 target 메소드가 성공적으로 return 된 후 수행된다.
  - <aop:after-returning /> 사용
  - @AfterReturing 어노테이션을 사용
- After throwing advice
  - After throwing advice 는 target 메소드가 수행 중 예외사항을 반환하고 종료하는 경우 수행된다.
  - <aop:after-throwing/> 사용
  - @AfterThrowing 어노테이션을 사용
- After (finally) advice
  - After (finally) advice 는 target 메소드가 수행 후 무조건 수행된다.
  - After advice 는 정상 종료와 예외 발생 경우를 모두 처리해야 하는 경우에 사용된다 (예:리소스 해제 작업)
  - 실제적으로 잘 사용되지는 않는다.
  - <aop:after /> 사용
  - @After 어노테이션을 사용
- Around advice
  - Around advice 는 메소드 수행 전후에 수행된다.
  - target 메서드 실행 전, 후에 실행 되며, 입력값, 리턴값, target 에 대한 변경이 가능
  - 가장 강력하며 많이 사용된다.
  - <aop:around /> 사용
  - @Around 어노테이션을 사용

❑ Advice 클래스에서 JoinPoint 사용

- Advice에서 target class 정보/입력값등을 확인하기 위해 org.aspectj.lang.JoinPoint 를 사용한다.
- 단, JoinPoint를 파라미터로 전달 받을 때는 반드시 첫번째 파라미터로 지정해야 한다.
- JoinPoint 인터페이스는 호출되는 대상객체, 메서드, 그리고 전달되는 파라미터 목록에 접근할 수 있는 메서드를 제공합니다.
  - Signature getSignature() - 호출되는 메서드에 대한 정보를 구한다.
  - Object getTarget() - 대상 객체를 구한다.
  - Object[] getArgs() - 파라미터 목록을 구한다.
- Signature 인터페이스는 메서드와 관련된 정보를 제공하기 위해 다음과 같은 메서드를 정의하고 있다
  - String getName() - 메서드의 이름을 구한다.
  - String toLongString() - 완전하게 표현된 메서드이름을 구한다
  - String toShortString() - 간단하게 표현된 메서드이름을 구한다.(이름만)
- Around Advice 의 경우 ProceedingJoinPoint를 첫번째 파라미터로 사용하면 된다. ProceedingJoinPoing는 JoinPoint 인터페이스를 상속하고있으므로 JoinPoint의 메서드를 사용할 수 있으며, 대상 객체를 호출 할 수 있는 메서드도 있다.
  - Object proceed() - 대상 객체의 메서드를 실행한다.

❑ 주의사항

- 프로젝트 개발 초기에 AOP를 위한 정책 수립
- 명명규칙을 확립하고 개발자들이 명명규칙에 따르도록 교육
- 하나의 Target 클래스에 너무 많은 Aspect를 적용되지 않도록 한다. .

❑ XML 스키마 기반 AOP

- Spring 2.0 이상에서 제공하는 XML 스키마 기반의 AOP를 사용할 수 있다.
- Java 5 버전을 사용할 수 없거나, XML 기반 설정을 선호하는 경우 사용한다.
- AOP 선언을 한 눈에 파악할 수 있다.

- 테스트를 위한 Target 클래스
  - ddit.test.aop. TargetObject.java

```
package ddit.test.aop;

public class TargetObject {

    public String total(int maxValue) {
      System.out.println("=== total 실행 ===");
      if(maxValue < 1 ){
        throw new RuntimeException( maxValue + "는 범위를 벗어났습니다.");
      }
      int sum = 0;
      for(int i = 1; i <= maxValue; i++){
        sum += i;
      }
      return "결과값은 = [" + sum + "]";
    } // total
 } //class
```

❑ Advice 작성

- 용어확인 : Advice는 Aspect의 실제 구현체로 Joinpoint에 삽입되어 동작할 수 있는 코드이다. Advice는 Joinpoint와 결합하여 동작하는 시점에 따라 before advice, after advice, around advice 타입으로 구분된다.
- 여기서 사용되는 메서드이름은 테스트를 위한 것이므로 메소드명은 상관이 없습니다.
  - ddit.test.aop.Advide4XML.java

```
package ddit.test.aop;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.ddit.exception.ServiceException;

public class Advice4XML {
  // 타겟 메서드 실행전 수행
  public void beforeMethod(JoinPoint joinPoint)  {
    String methodName = joinPoint.getSignature().getName();
    String className = joinPoint.getTarget().getClass().getSimpleName();
    System.out.println("beforMethod 실행 " + className + "." + methodName);
  }

  // 정상적으로 동작이 된 경우 실행됨
  public void afterReturningMethod(JoinPoint joinPoint, Object retVal)  {
    String methodName = joinPoint.getSignature().getName();
    String className = joinPoint.getTarget().getClass().getSimpleName();
    Object param = joinPoint.getArgs()[0];
    System.out.println("afterReturningMethod 실행 " + className + "." + methodName
                       + ", return Value=" + retVal);
  }

  // 예외가 발생하고, 종료하는 경우 수행됨
  public void afterThrowingMethod(JoinPoint joinPoint, Exception ex) throws Throwable{
    String methodName = joinPoint.getSignature().getName();
    String className = joinPoint.getTarget().getClass().getSimpleName();
    System.out.println("afterThrowingMethod 실행, " + className + "." + methodName);
    System.err.println("예외발생:" + ex.getMessage());
  }

  // 타겟 메서드 수행후 무조건 수행됨
  public void afterMethod(JoinPoint joinPoint) throws Exception  {
    String methodName = joinPoint.getSignature().getName();
    String className = joinPoint.getTarget().getClass().getSimpleName();
    System.out.println("afterMethod 실행 " + className + "." + methodName);
  }

  // 실행전 처리, 타겟 메서드 직접호출, 예외처리 가능
  public Object aroundMethod(ProceedingJoinPoint joinPoint) throws Throwable {
    String methodName = joinPoint.getSignature().getName();
    String className = joinPoint.getTarget().getClass().getSimpleName();
    System.out.println("aroundMethod실행 - 1");
    long startTime = System.currentTimeMillis();
    Object retVal =  joinPoint.proceed();
    System.out.println("aroundMethod 실행 - 2 " + className + "." + methodName
                       +", lead time =" + ( System.currentTimeMillis()  - startTime));
    return retVal;
  }
} //class
```

❑ Aspect 정의

- 용어확인 : Advice와 Pointcut을 합쳐서 하나의 Aspect라고 한다.
  - ddit/test/aop/aopTestXML.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"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/aop
            http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-3.0.xsd
            http://www.springframework.org/schema/tx
            http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

  <!-- // ========== Advice 정의 ============ -->
  <bean id="testAdvice" class="ddit.test.aop.Advice4XML" />

  <!-- // ========== Aspect   정의 ============ -->
  <aop:config>
    <aop:pointcut id="targetMethod" expression="execution(* total(..))" />
    <aop:aspect ref="testAdvice">
      <aop:before          pointcut-ref="targetMethod" method="beforeMethod" />
      <aop:after-returning pointcut-ref="targetMethod" method="afterReturningMethod"
                           returning="retVal" />
      <aop:after-throwing  pointcut-ref="targetMethod" method="afterThrowingMethod"
                           throwing="ex" />
      <aop:after           pointcut-ref="targetMethod" method="afterMethod" />
      <aop:around          pointcut-ref="targetMethod" method="aroundMethod" />
    </aop:aspect>
  </aop:config>

 <!-- // ========== 대상 빈 정의 ============ -->
  <bean id="testTarget" class="ddit.test.aop.TargetObject" />

</beans>
```

- <aop:config> : AOP 설정 정보를 나타내는 루트 태그
- <aop:pointcut> : Pointcut을 설정
- <aop:aspect> : Aspect를 설정한다.
- <aop:advisor> : Spring API 를 사용한 Advice 이용 할 때

❑ Aspect Test 클래스 작성

- ddit.test.aop.AopTest4XML.java

```
package ddit.test.aop;

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

public class AopTest4XML {
  public static void main(String[] args) throws Exception {
    // TODO Auto-generated method stub
      ApplicationContext context =
        new ClassPathXmlApplicationContext("ddit/test/aop/aopTestXML.xml");

        TargetObject testTarget = (TargetObject)context.getBean("testTarget");
        String ret = testTarget.total(50);
        System.out.println("main :" + ret);
  } // main

}
```

❑ @AspectJ 어노테이션을 이용한 AOP

- @AspectJ는 Java 5 어노테이션을 사용하여 일반 Java 클래스로 관점(Aspect)을 정의하는 방식이다.
- @AspectJ 방식은 AspectJ 5 버전에서 소개되었으며, Spring은 2.0 버전부터 AspectJ 5 어노테이션을 지원한다.
- Spring AOP 실행환경은 AspectJ 컴파일러나 Weaver에 대한 의존성이 없이 @AspectJ 어노테이션을 지원한다.
- @AspectJ 어노테이션을 사용한다고 해서 AspectJ 와 관련 있는것이 아니라 단지 어노테이션만을 이용하는 것이다.
  - ddit.test aop.Advide4AspectJ.java

```
package ddit.test.aop;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;

@Aspect
public class Advice4AspectJ {
  // 메서드의 내용은 이전과 동일하므로  "..." 처리함

  @Pointcut("execution(* total(..))")
  private void dummyPointcut() {
    // pointcut을 생성하기 위한 dummy method
    // 반환은 void, 접근제한자는 public 보다는 private으로 하죠~~
  }

  // 타켓 메서드 실행 전 수행됨
  @Before(value="dummyPointcut()")
  public void beforeMethod(JoinPoint joinPoint)  {
    ...
  }

  // 정상적으로 동작이 될 경우 실행됨
  @AfterReturning(pointcut="execution(* total(..))" , returning= "retVal")
  public void afterReturningMethod(JoinPoint joinPoint, Object retVal)  {
    ...
  }

  // 예외가 발생하고, 종료하는 경우 수행됨
  @AfterThrowing(pointcut="dummyPointcut()", throwing="ex" )
  public void afterThrowingMethod(JoinPoint joinPoint, Exception ex) throws Throwable  {
    ...
  }

  // 메서드 수행 종료후 무조건 수행됨
  @After(value="execution(* total(..))")
  public void afterMethod(JoinPoint joinPoint) throws Exception  {
    ...
  }

  // 실행전 처리, 타겟 메서드 직접호출, 예외처리 가능
  @Around("dummyPointcut()")
  public Object aroundMethod(ProceedingJoinPoint joinPoint) throws Throwable {
    ...
  }
} //class
```

- @AspectJ 설정하기
  - @AspectJ를 사용하기 위해서 다음 코드를 Spring 설정에 추가한다.
  - <aop:aspectj-autoproxy/>

- 관점(Aspect) 정의하기
  - 클래스에 @Aspect 어노테이션을 추가하여 Aspect를 생성한다. @Aspect 설정이 되어 있는 경우 Spring은 자동적으로 @Aspect 어노테이션을 포함한 클래스를 검색하여 Spring AOP 설정에 반영한다.

- 포인트컷(Pointcut) 정의하기
  - 포인트컷은 결합점(Join points)을 지정하여 충고(Advice)가 언제 실행될지를 지정하는데 사용된다. Spring AOP는 Spring빈에 대한 메소드 실행 결합점만을 지원하므로, Spring에서 포인트컷은 빈의 메소드 실행점을 지정하는 것으로 생각할 수 있다.

  - ddit/test/aop/aopAdviceAspectJ.xml

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

 <!-- // ===== @AspectJ 검색, Aspect정의  ======= -->
 <aop:aspectj-autoproxy />

 <!-- // ===== Advice 정의(@AspectJ 사용된 )   ======= -->
 <bean id="testAdvice" class="ddit.test.aop.Advice4AspectJ" />

 <!-- // ========== 대상 빈 정의 ============ -->
  <bean id="testTarget" class="ddit.test.aop.TargetObject" />

</beans>
```

  - ddit.test.aop.aopTest4AspectJ.java

```java
package ddit.test.aop;

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

public class aopTest4XML {
  public static void main(String[] args) throws Exception {
    // TODO Auto-generated method stub
      ApplicationContext context =
        new ClassPathXmlApplicationContext("ddit/test/aop/aopTestAspectJ.xml");

        TargetObject testTarget = (TargetObject)context.getBean("testTarget");
        String ret = testTarget.total(50);
        System.out.println("main :" + ret);
  } // main

}
```