

AOP

# AOP

- 스프링은 DI 컨테이너로써 뿐만 아니라 AOP 프레임워크로서의 기능도 제공하고 있다. AOP(Aspect Oriented Programming : 관점지향 프로그래밍)는 최근 각광받는 새로운 프로그래밍 기법에 대한 개념이다

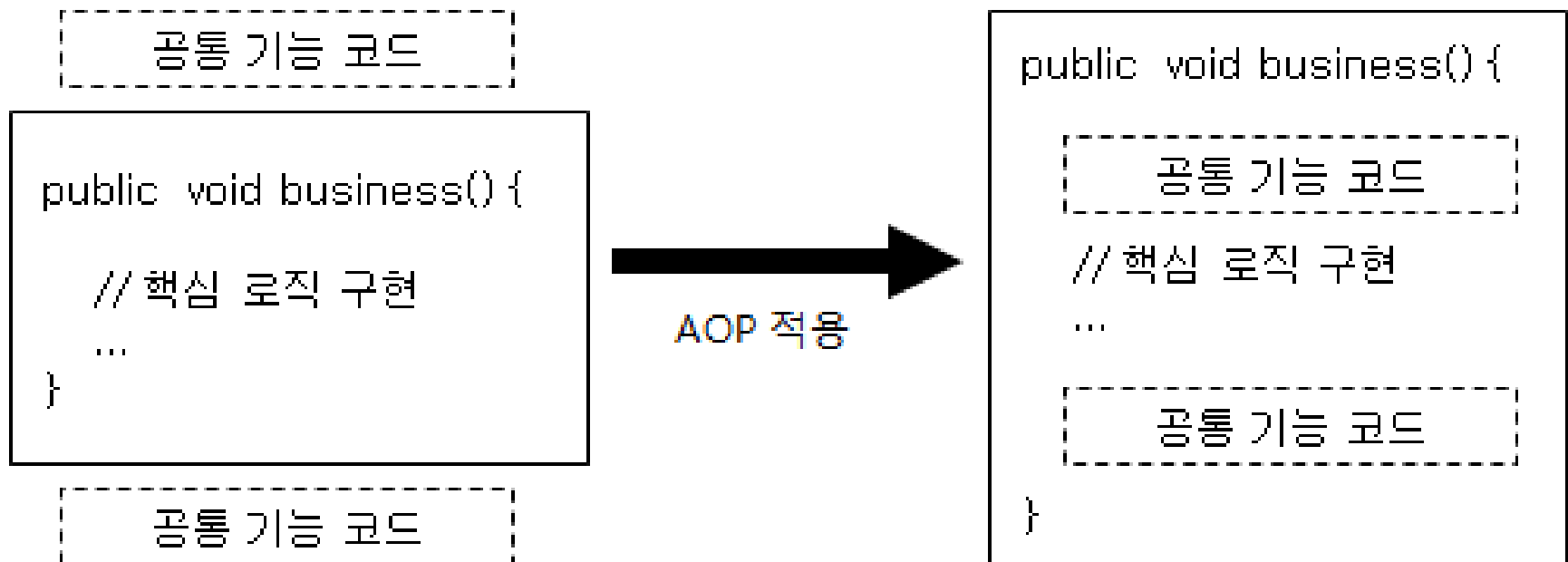
# AOP

- 관점 지향 프로그래밍(Aspect Oriented Programming, 이하 AOP)은 결국 객체 지향 프로그래밍(Object Oriented Programming)의 뒤를 잇는 또 하나의 프로그래밍 언어 구조라고 생각될 수 있다.
- OOP를 더욱 OOP답게 만들어 준다.

# AOP

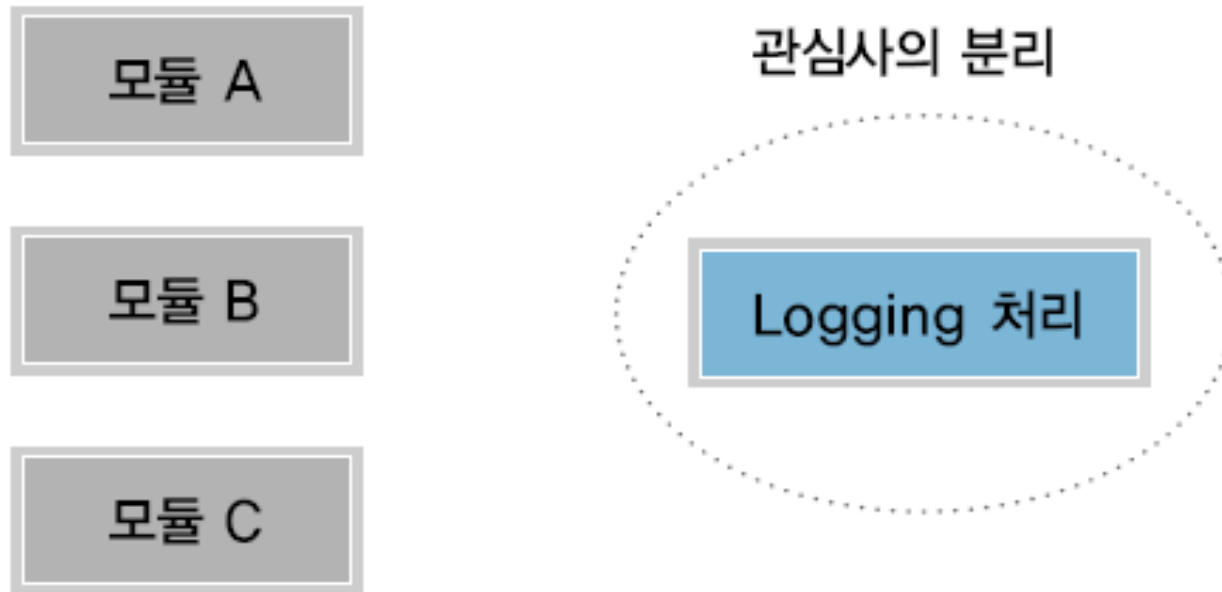
- Aspect 지향에서 중요한 개념은 「**횡단 관점의 분리(Separation of Cross-Cutting Concern)**」이다. 이에 대한 이해를 쉽게 하기 위해서 은행 업무를 처리하는 시스템을 예를 들어 보겠다.
- 은행 업무 중에서 계좌이체, 이자계산, 대출처리 등은 주된 업무(핵심 관점, 핵심 비즈니스 기능)로 볼 수 있다. 이러한 업무(핵심 관점)들을 처리하는데 있어서 「로그인», 「보안», 「트랜잭션」등의 처리는 **어플리케이션 전반에 걸쳐 필요한 기능**으로 핵심 비즈니스 기능과는 구분하기 위해서 **공통 관심 사항(Cross-Cutting Concern)**이라고 표현한다.

# 공통 관심 사항(cross-cutting concern)과 핵심 관심 사항(core concern)



# AOP

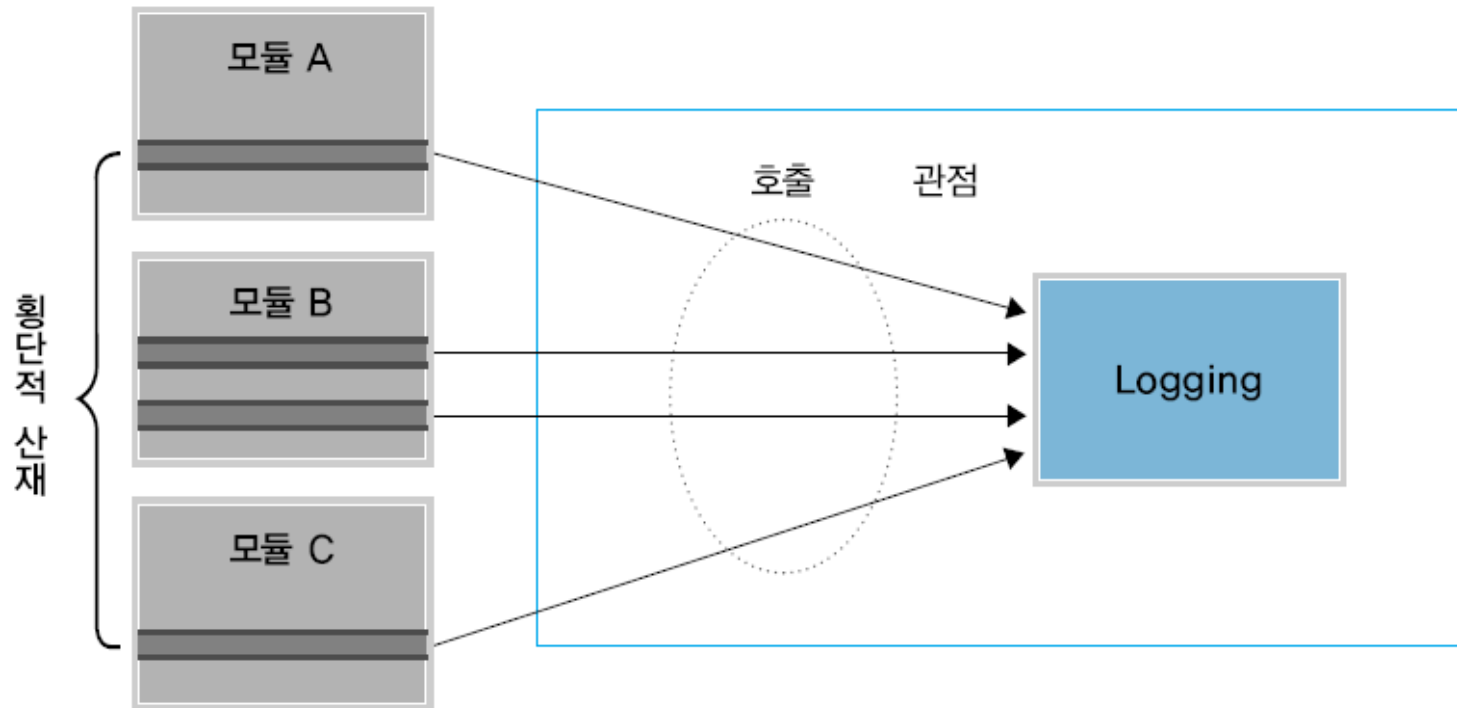
- 오브젝트 지향에서는 이들 업무들을 하나의 클래스라는 단위로 모으고 그것들을 모듈로부터 분리함으로써 재사용성과 보수성을 높이고 있다.



▲ 기존의 객체 지향에서 관점의 분리

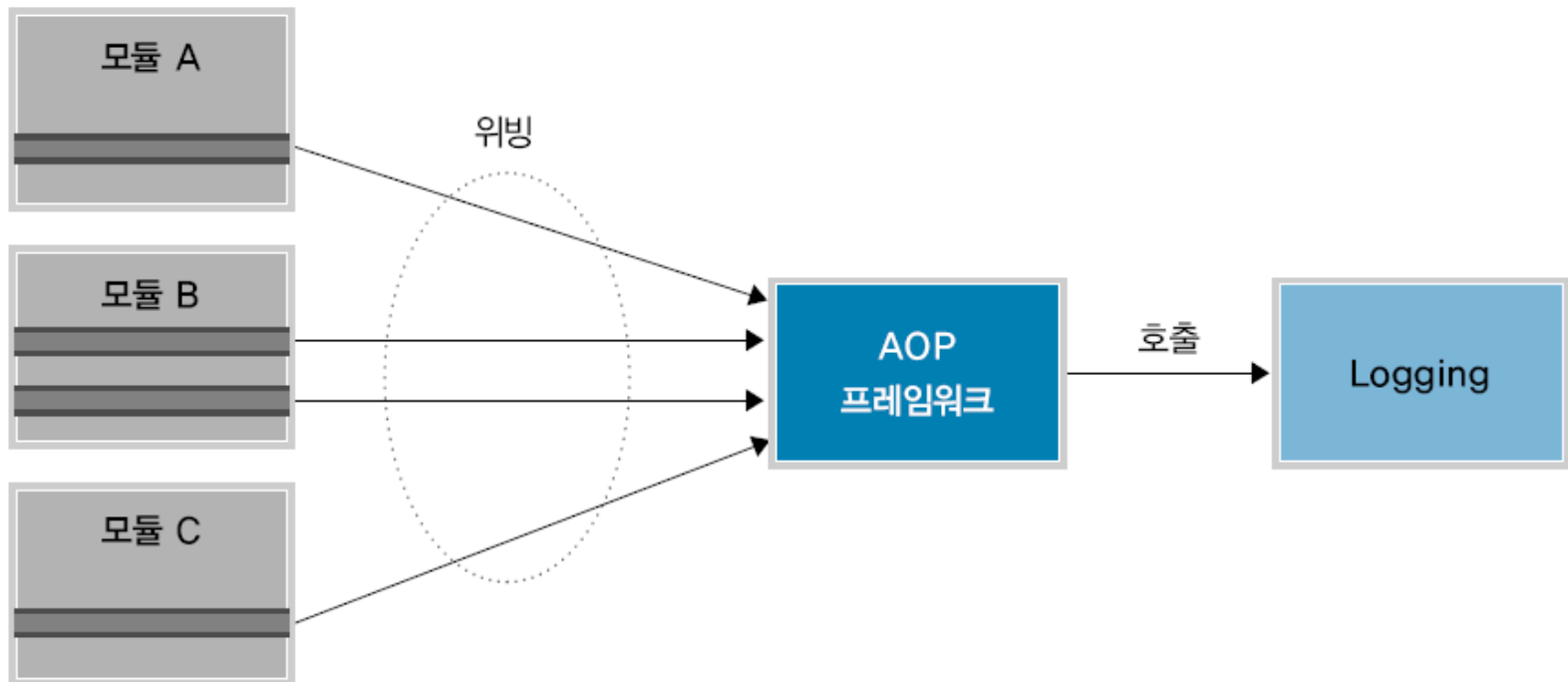
# AOP

- 오브젝트 지향에서는 로깅이라는 기능 및 관련하는 데이터 자체는 각 모듈로부터 분리하는 것으로 성공했지만 **그 기능을 사용하기 위해서 코드까지는 각 모듈로부터 분리할 수 없다.** 그렇기 때문에 분리한 기능을 이용하기 위해서 코드가 각 모듈에 횡단으로 산재하게 된다.



# AOP

- AOP에서는 분리한 기능의 호출도 포함하여 「관점」으로 다룬다.  
그리고 이러한 각 모듈로 산재한 관점을 「횡단 관점」라 부르고 있다.
- AOP에서는 이러한 「횡단 관점」까지 분리함으로써 **각 모듈로부터 관점에 관한 코드를 완전히 제거하는 것을 목표로 한다.**



▲ AOP의 횡단 관점의 분리와 위빙

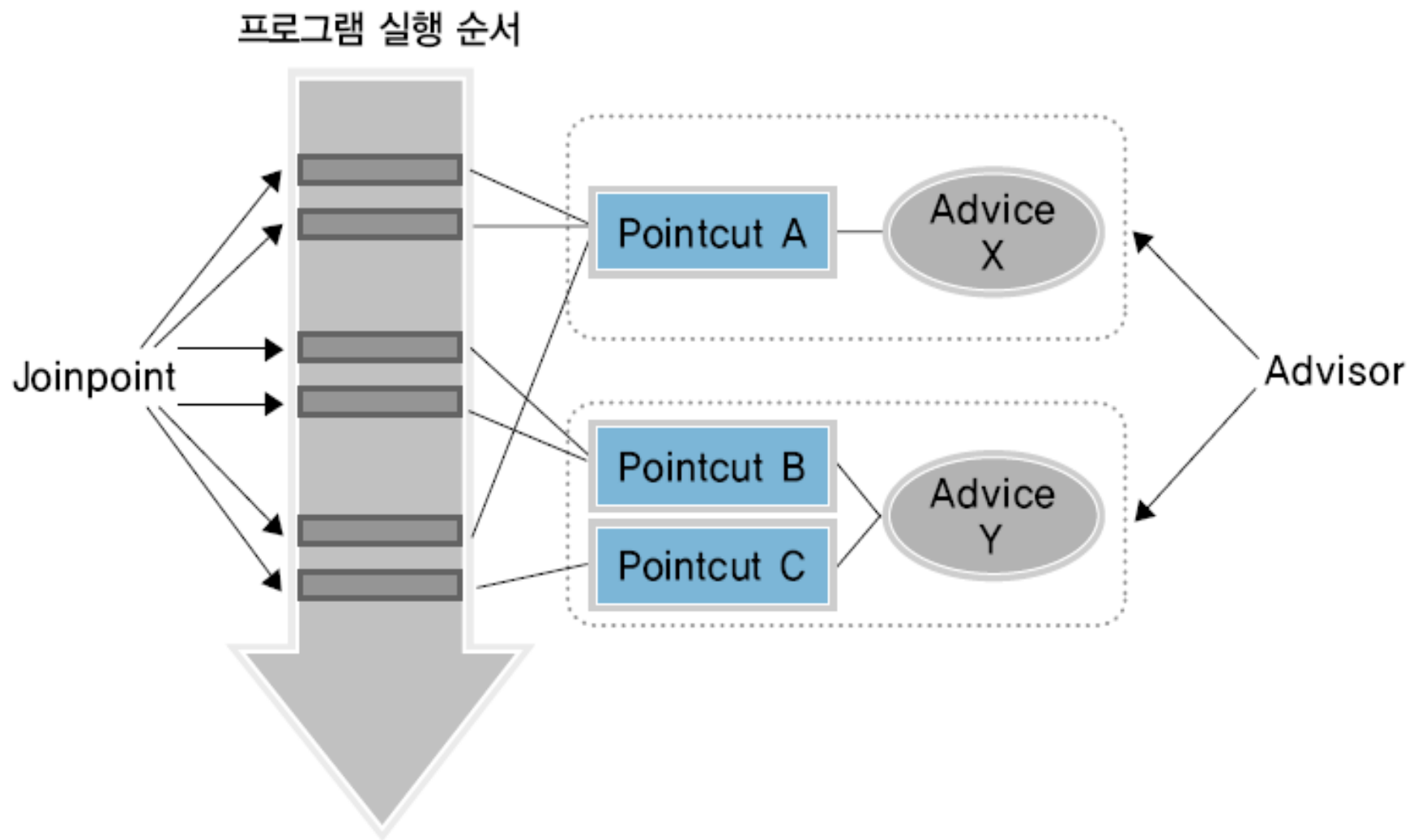


# 스프링 AOP 용어

- **Joinpoint** - 「클래스의 인스턴스 생성 시점」, 「메소드 호출 시점」 및 「예외 발생 시점」과 같이 애플리케이션을 실행할 때 특정 작업이 시작되는 시점을 Joinpoint라 한다. 즉, 내가 원하는 위치에 특정 공통모듈을 삽입하기 위해 시작점을 알아야 하는데 그러한 시작 기준점을 뜻함.
- **Pointcut** - 여러 개의 Joinpoint를 하나로 결합한(묶은) 것을 Pointcut이라고 부른다.
- **Advice** - Joinpoint에 삽입되어져 동작할 수 있는 코드를 Advice라 한다.(실제적인 코딩부분)

# 스프링 AOP 용어

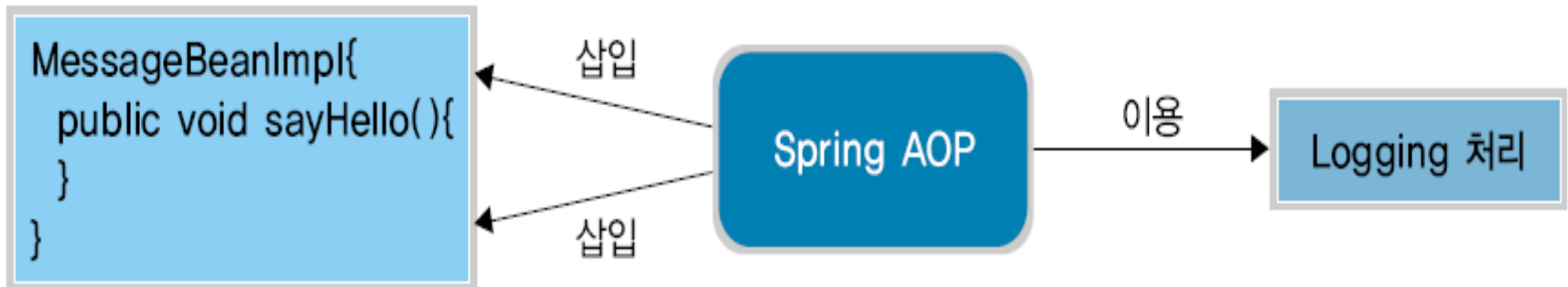
- **Advisor** - Advice와 Pointcut를 하나로 묶어 취급한 것을 Advisor라 부른다.
- **Weaving** - Advice를 핵심 로직 코드에 삽입하는 것을 Weaving이라 부른다.(AOP프레임웍에서 직접 호출해서 전달 할 수 있도록하는 것을 위빙이라 함.)
- **Target** - 핵심 로직을 구현하는 클래스를 말한다.
- **Aspect** - 여러 객체에 공통으로 적용되는 공통 관점 사항을 Aspect라 부른다.



▲ 스프링 AOP에서의 용어와 개념

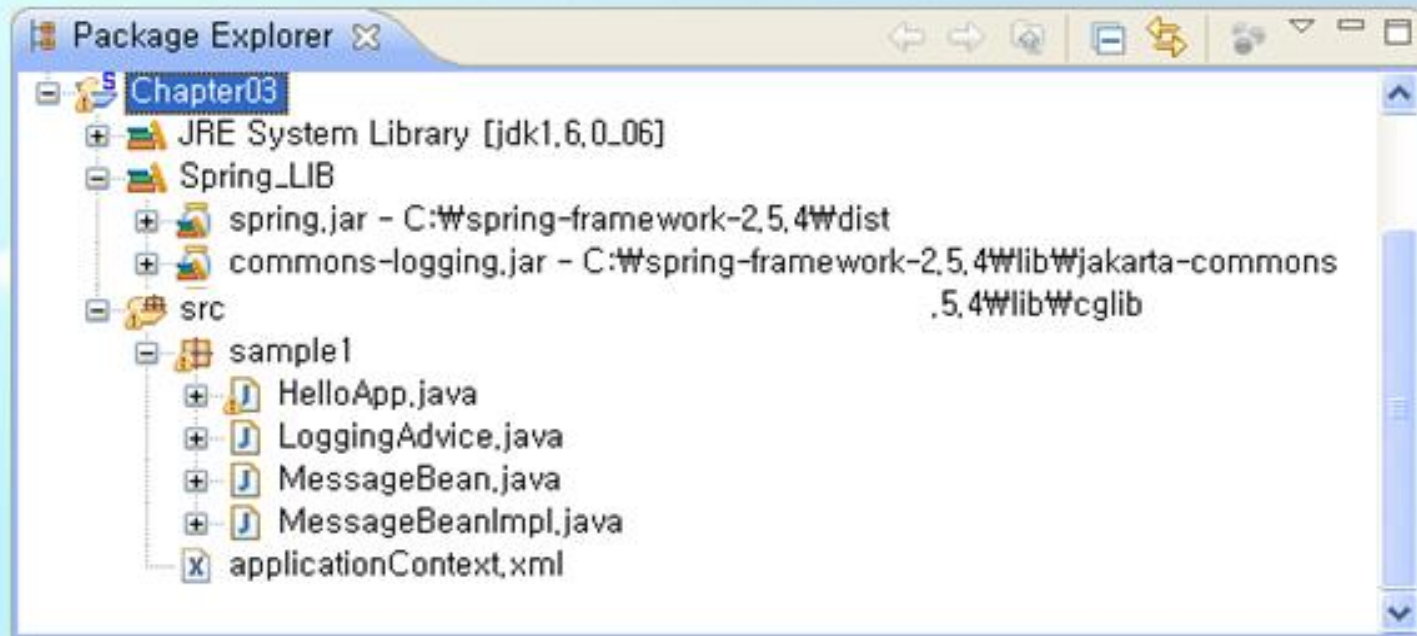
# AOP를 이용한 logging 구현 예제

- 이 예제에서는 AOP 구조를 활용하여 메소드 트레이스 정보의 Logging 처리를 MessageBeanImpl의 sayHello() 메소드 호출 전 후에 삽입한다.
- 로깅 처리 자체 및 그 호출은 MessageBeanImpl에는 기술하지 않는다.
- 스프링이 제공하는 기능인 「스프링 AOP」가 그 역할을 담당한다.



▲ 예제 개요 그림

## ▼ 파일 구성



**sayHello() 메소드가 핵심 로직이고 이 메소드를 멤버로 갖는 MessageBeanImpl는 타겟 클래스가 된다.**

**LoggingAdvice 클래스가 로깅처리를 담당하고 있게 된다.**

# 스프링에서 AOP를 구현하는 방법

1. 각 Advice 타입에 대응하는 인터페이스 구현
2. AspectJ 스타일 AOP의 이용
3. 어노테이션으로 AOP 설정

# 각 Advice 타입에 대응하는 인터페이스 구현

1. Advice 클래스를 작성한다. – 특정 인터페이스구현
2. 설정 파일에 Pointcut을 설정한다.
3. 설정 파일에 Advice와 Pointcut을 묶어 놓는 Advisor를 설정한다.
4. 설정 파일에 ProxyFactoryBean 클래스를 이용하여 대상 객체에 Advisor를 적용한다.
5. `getBean( )` 메소드로 빈 객체를 가져와 사용한다.

# 각 Advice 타입에 대응하는 인터페이스 구현

## Advice 클래스를 작성한다

(JoinPoint 앞뒤에서 실행되는 Advice)

```
public class LoggingAdvice implements MethodInterceptor {  
  
    public Object invoke(MethodInvocation invocation) throws Throwable {  
  
        String methodName = invocation.getMethod().getName();  
        Stopwatch sw = new Stopwatch();  
  
        sw.start(methodName);  
  
        System.out.println("[LOG] METHOD: " + methodName + " is calling.");  
        Object rtnObj = invocation.proceed();  
  
        sw.stop();  
  
        System.out.println("[LOG] METHOD: " + methodName + " was called.");  
        System.out.println("[LOG] 처리시간 " + sw.getTotalTimeMillis() / 1000 + "초");  
  
        return rtnObj;  
    }  
}
```



# 각 Advice 타입에 대응하는 인터페이스 구현

## 설정 파일 작성하기

```
<bean id="loggingAdvice" class="sample1.LoggingAdvice" />
```

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="targetBean"/>
  <property name="interceptorNames">
    <list>
      <value>advisor</value>
    </list>
  </property>
</bean>
```

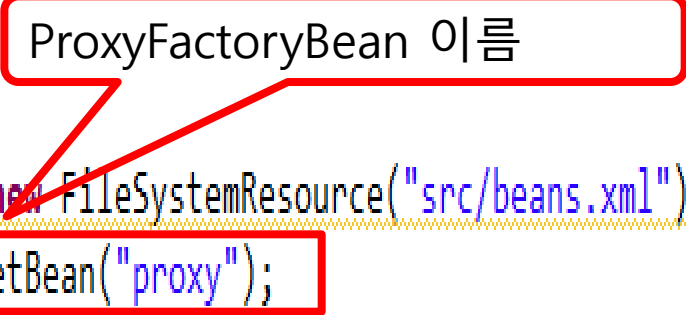
Advice와 Pointcut를 하나로 묶어 취급한 것

```
<bean id="advisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
  <property name="advice" ref="loggingAdvice"/>
  <property name="pointcut">
    <bean class="org.springframework.aop.support.JdkRegexpMethodPointcut">
      <property name="pattern">
        <value>.*sayHello.*</value>
      </property>
    </bean>
  </property>
</bean>
```

# 각 Advice 타입에 대응하는 인터페이스 구현

getBean( ) 메소드로 빈 객체를 가져와 사용  
한다

```
public class HelloApp {  
    public static void main(String[] args) {  
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource("src/beans.xml"));  
        MessageBean bean = (MessageBean)factory.getBean("proxy");  
  
        /*ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");  
        MessageBean bean = (MessageBean)context.getBean("proxy");*/  
  
        bean.sayHello();  
    }  
}
```



# AspectJ 스타일 AOP의 이용

1. AspectJ는 제록스 팔로알토 연구소에서 개발했던 AOP 구현으로 현재 [eclipse.org](http://eclipse.org) 개발프로젝트에서 관리하고 있음.
2. 특정 클래스나 인터페이스를 상속 / 구현 하지 않고 POJO 방식코딩
3. Xml 설정파일에서 AspectJ스타일로 Aop설정한다.

# AspectJ 스타일 AOP의 이용

```
<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"
  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">
```

```
<bean id="loggingSample" class="sample1.LoggingSample" />
```

Aspect설정: Advice를 어떤  
Pointcut에 적용할지 설정

```
<aop:config>
  <aop:aspect id="logAspect" ref="loggingSample">
    <aop:pointcut expression="execution(* sayHello())" id="logPointCut"/>
    <aop:around pointcut-ref="logPointCut" method="logAround"/> <!--LoggingSample의 logAround메소드 호출 -->
  </aop:aspect>
</aop:config>
```

# AspectJ ~~스타일~~ AOP의 이용

```
public class LoggingSample {  
    public Object logAround(ProceedingJoinPoint pjp) throws Throwable {  
        String methodName = pjp.getKind();  
        Stopwatch sw = new Stopwatch();  
  
        sw.start(methodName);  
  
        System.out.println("[LOG] METHOD: " + methodName + " is calling.");  
        Object rtnObj = pjp.proceed();  
  
        sw.stop();  
  
        System.out.println("[LOG] METHOD: " + methodName + " was called.");  
        System.out.println("[LOG] 처리시간 " + sw.getTotalTimeMillis() / 1000 + "초");  
  
        return rtnObj;  
    }  
}
```

}

# AspectJ 스타일 AOP의 이용

```
public class HelloApp {  
    public static void main(String[] args) {  
        ApplicationContext factory = new FileSystemXmlApplicationContext("src/beans.xml");  
        MessageBean bean = (MessageBean)factory.getBean("targetBean");  
  
        bean.sayHello();  
    }  
}
```

# 어노테이션으로 AOP 설정

- @AspectJ 어노테이션을 Advice로 사용하는 클래스에 기술한다.
- Advice처리를 맡는 메서드에 @Around 어노테이션을 기술한다.
- @Around 어노테이션 값에 Pointcut 정의를 작성한다.

- 어노테이션을 사용해 AOP를 설정하는 경우 설정파일에 `<aop:aspectj-autoproxy/>` 반드시 기술한다.
- `<aop:config>` 요소 내용을 어노테이션으로 설정하므로 `<aop:config>` 요소는 삭제한다.

# 어노테이션으로 AOP 설정

```
@Aspect
public class LoggingSample {
    @Around("execution(* sayHello())")
    public Object logAround(ProceedingJoinPoint pjp) throws Throwable {
        String methodName = pjp.getKind();
        Stopwatch sw = new Stopwatch();

        sw.start(methodName);

        System.out.println("[LOG] METHOD: " + methodName + " is 호출중입니다..");
        Object rtnObj = pjp.proceed();

        sw.stop();

        System.out.println("[LOG] METHOD: " + methodName + " 완료되었습니다..");
        System.out.println("[LOG] 처리시간 " + sw.getTotalTimeMillis() / 1000 + "초");

        return rtnObj;
    }
}
```

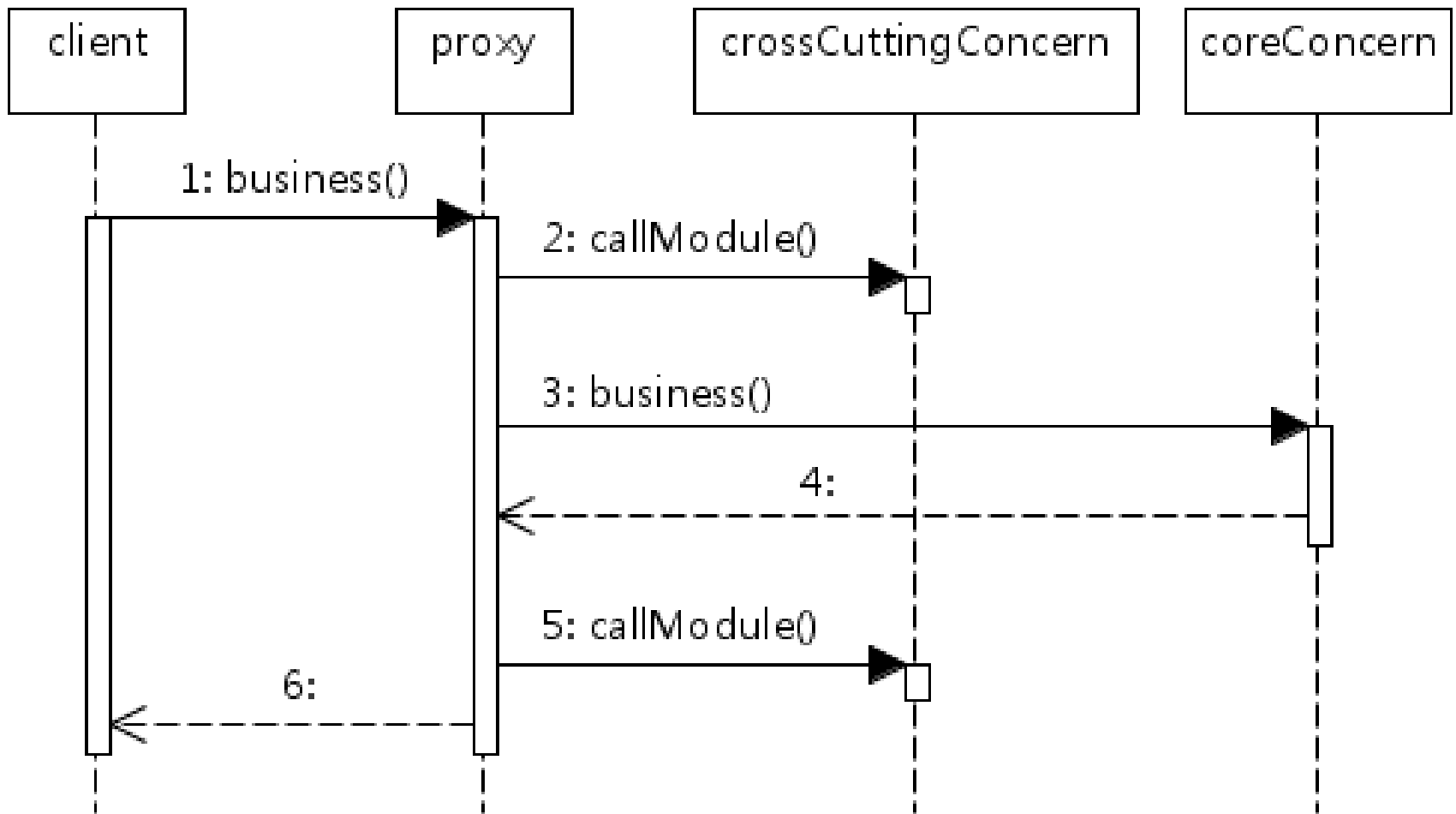


# 어노테이션으로 AOP 설정

```
3 <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xmlns:aop="http://www.springframework.org/schema/aop"
6     xsi:schemaLocation="http://www.springframework.org/schema/beans
7     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8     http://www.springframework.org/schema/aop
9     http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
10
11     <bean id="loggingSample" class="sample1.LoggingSample" />
12
13     <aop:aspectj-autoproxy/>
14
15     <bean id="targetBean" class="sample1.MessageBeanImpl">
16         <property name="name">
17             <value>Spring</value>
18         </property>
19     </bean>
20 </beans>
```

# 세 가지 Weaving 방식

- 컴파일 시에 Weaving
  - AspectJ에서 사용하는 방식
- 클래스로딩 시에 Weaving
  - AspectJ 5/6 버전이 컴파일 방식과 더불어 제공
- 런타임 시에 Weaving
  - 프록시를 이용하여 AOP를 적용한다.
  - 핵심 객체에 직접 접근하지 않고 중간에 프록시를 통하여 핵심 로직을 구현한 객체에 접근하게 된다.



\*\* 메서드가 호출될 때에만 Advice를 적용할 수 있기 때문에 필드 값 변경과 같은 Joinpoint에 대해서는 적용할 수 없다.

# 스프링에서의 AOP

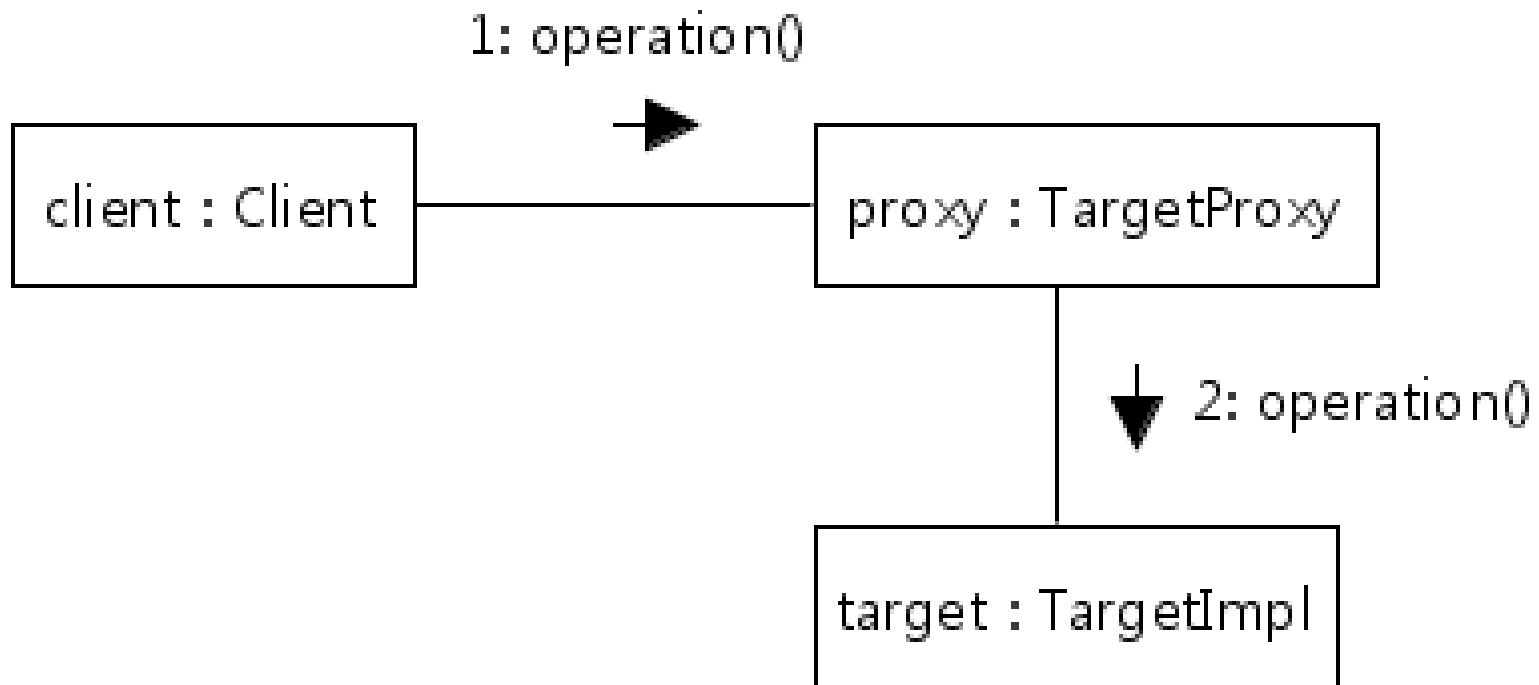
- 스프링 AOP는 메서드 호출 Joinpoint만을 지원한다.
- 필드 값 변경과 같은 Joinpoint를 사용 하려면 AspectJ 같은 풍부한 기능은 지원하는 AOP 도구를 사용해야 한다.

# 스프링 AOP 구현 방식

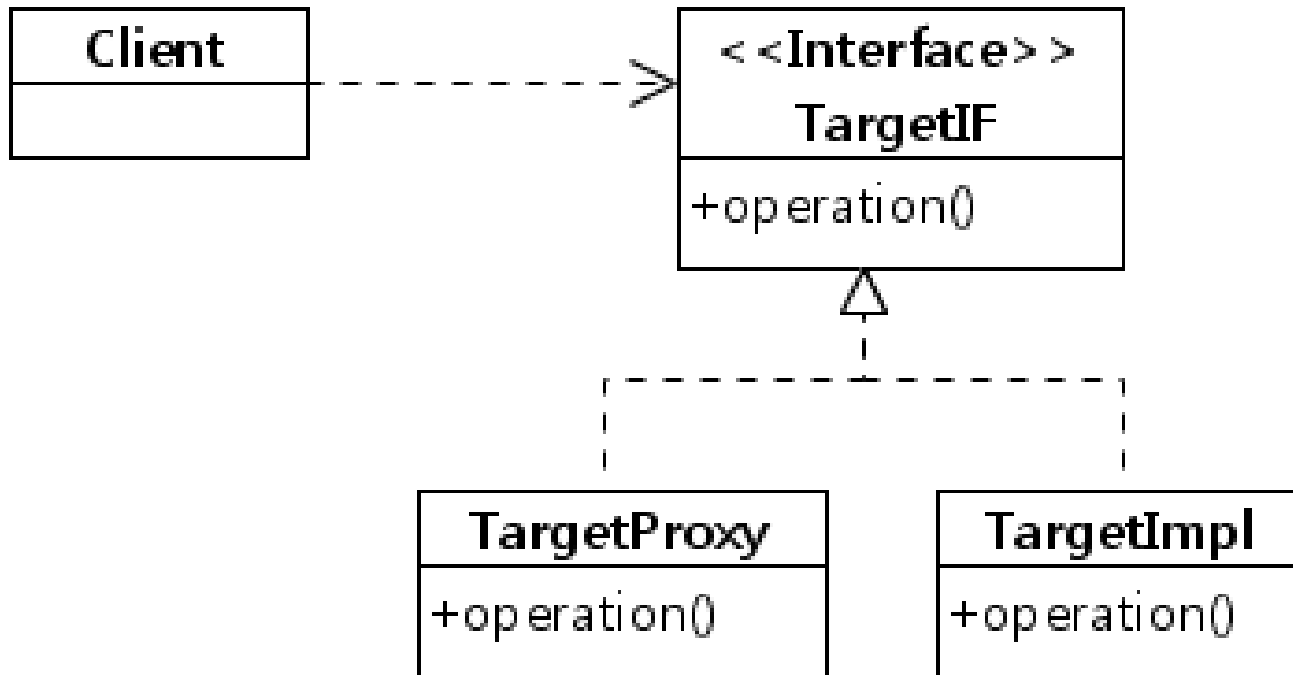
- XML 스키마 기반의 POJO 클래스를 이용한 AOP 구현
- AspectJ 5/6에서 정의한 @AspectJ 어노테이션 기반의 AOP 구현
- 스프링 API를 이용한 AOP 구현

\*\* 어떤 방식을 사용하더라도 내부적으로는 프록시를 이용하여 AOP가 구현되므로 메서드 호출에 대해서만 AOP를 적용할 수 있다.

# 프록시를 이용한 AOP 구현



스프링은 Aspect의 적용대상이 되는 객체에 대한 프록시를 만들어 제공하며, client는 프록시를 통하여 간접적으로 대상객체에 접근하게 된다.



- \*\* 대상객체가 인터페이스를 구현
  - > `java.lang.reflect.Proxy`를 이용하여 프록시 객체 생성
- \*\* 대상객체가 인터페이스를 구현하고 있지 않을때
  - > CGLIB을 이용하여 프록시 객체를 생성
  - > 대상객체 및 메서드가 `final`이 될 수 없다.

# 구현 가능한 Advice의 종류

종류	설명
Before Advice	대상 객체의 메서드 호출 전에 공통 기능을 실행한다.
After Returning Advice	대상 객체의 메서드가 예외 없이 실행한 이후에 공통 기능을 실행한다.
After Throwing Advice	대상 객체의 메서드를 실행하는 도중 예외가 발생한 경우에 공통기능을 실행한다.
After Advice	대상 객체의 메서드를 실행하는 도중에 예외가 발생했는지의 여부와 상관없이 메서드 실행 후 공통 기능을 실행한다.
Around Advice	대상 객체의 메서드 실행 전, 후 또는 예외 발생 시점에 공통 기능을 실행한다.

**\*\* 대상 객체의 메서드의 실행하기 전/후에 원하는 기능을 삽입 할 수 있기 때문에 Around Advice를 범용적으로 사용함.**



## XML 스키마 기반의 POJO 클래스를 이용한 AOP 구현

- 스프링 2 버전 부터 스프링 API를 사용하지 않은 POJO 클래스를 이용하여 Advice를 적용하는 방법이 추가됨.

# XML 스키마 이용 AOP 구현 과정

- 관련 jar를 클래스 패스에 추가한다.
- 공통기능을 제공하는 Advice 클래스를 구현한다.
- XML 설정파일에서 <aop:config>를 이용하여 Aspect를 설정한다.
- Advice를 어떤 Pointcut에 적용할지를 지정하게 된다.

# 공통 기능을 제공할 Advice클래스 작성

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

- \*\* 특정 Joinpoint에서 실행될 trace()메서드를구현
- \*\* 매개변수로 전달받은 ProceedingJoinPoint를 통해 AroundAdvice를 구현 할 수 있음.

# Advice 객체의 생성

```
<!-- Advice 클래스를 빈으로 등록 -->  
<bean id="performanceTraceAdvice"  
      class="madvirus.spring.chap05.aop.pojo.ProfilingAdvice" />
```

# Aop의 설정

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

<!-- Aspect 설정: Advice를 어떤 Pointcut에 적용할 지 설정 -->
<aop:config>
    <aop:aspect id="traceAspect1" ref="performanceTraceAdvice">
        <aop:pointcut id="publicMethod"
            expression="execution(public * madvirus.spring.chap05.board..*(..))" />
        <aop:around pointcut-ref="publicMethod" method="trace" />
    </aop:aspect>
</aop:config>
```

madvirus.spring.chap05 패키지의 모든 public 메서드를 pointcut으로 설정

# Target이 되는 객체의 생성

```
<bean id="writeArticleService"
      class="madvirus.spring.chap05.board.service.WriteArticleServiceImpl">
  <constructor-arg>
    <ref bean="articleDao" />
  </constructor-arg>
</bean>

<bean id="articleDao"
      class="madvirus.spring.chap05.board.dao.MySQLArticleDao" />

<bean id="memberService"
      class="madvirus.spring.chap05.member.service.MemberServiceImpl" />
```

```

public class MainQuickStart {

    public static void main(String[] args) {
        String[] configLocations = new String[] { "acQuickStart.xml" };
        ApplicationContext context = new ClassPathXmlApplicationContext(
            configLocations);

        WriteArticleService articleService = (WriteArticleService) context
            .getBean("writeArticleService");
        articleService.write(new Article());

        MemberService memberService = context.getBean("memberService",
            MemberService.class);
        memberService.regist(new Member());
    }
}

```

# AOP 관련 정보 설정

<b>&lt;aop:config&gt;</b>	<b>AOP 설정 정보임을 나타냄</b>
<b>&lt;aop:aspect&gt;</b>	<b>Aspect를 설정</b>
<b>&lt;aop:pointcut&gt;</b>	<b>Pointcut을 설정</b>
<b>&lt;aop:around&gt;</b>	<b>Around Advice를 설정</b>



# Advice 정의 관련 태그

태그	설명
<aop:before>	메서드 실행 전에 적용되는 Advice
<aop:after-returning>	메서드가 정상적으로 실행 된 후에 적용되는 Advice
<aop:after-throwing>	메서드가 예외를 발생 시킬때 적용되는 Advice
<aop:after>	메서드가 정상적으로 실행되는지 예외를 발생시키는지 여부에 상관없이 적용되는 Advice
<aop:around>	메서드 호출 이전, 이후, 예외발생 등 모든 시점에 적용가능한 Advice

# <aop:around>에 pointcut을 직접 설정

```
<aop:config>
  <aop:aspect id="traceAspect1" ref="performanceTraceAdvice">
    <aop:pointcut id="publicMethod"
      expression="execution(public * madvirus.spring.chap05.board..*(..))" />
    <aop:around pointcut-ref="publicMethod" method="trace" />
  </aop:aspect>

  <aop:aspect id="traceAspect2" ref="performanceTraceAdvice">
    <aop:around pointcut="execution(public * madvirus.spring.chap05.member..*(..))"
      method="trace" />
  </aop:aspect>
</aop:config>
```

# Advice 타입 별 클래스 작성

```
<aop:aspect id="traceAspect2" ref="performanceTraceAdvice">  
  <aop:around pointcut="execution(public * madvirus.spring.chap05.member..*(..))"  
    method="trace" />  
</aop:aspect>
```

AroundAdvice가 제공

# Before Advice

```
<aop:config>
  <aop:aspect id="loggingAspect" ref="loggingAdvice"
    <aop:pointcut id="publicMethod"
      expression="execution(public * madvirus.spr
    <aop:before method="before" pointcut-ref="publi
    <aop:after-returning method="afterReturning" pc
    <aop:after-throwing method="afterThrowing" poin
    <aop:after method="afterFinally" pointcut-ref='
  </aop:aspect>
```

```
public void before()
{
    // 대상 객체의 메서드 실행 이전에 적용할 기능 구현
}
```

```
public void before(JoinPoint joinPoint)
// 대상 객체 및 호출되는 메서드에 대한 정보가 필요할 경우에 사용
{..
}
```

\*\* Before Advice에서 예외를 발생시키면 대상 객체의 메서드가 호출되지 않기 때문에 메서드를 실행하기 전에 접근 권한을 검사해서 권한이 없을 경우 예외를 발생 시키도록 하는 것이 적합하다.

# After Returning Advice

- 대상 객체의 메서드가 정상적으로 실행 된 후 공통 기능을 적용하고자 할 때 사용

```
<bean id="logging" class="madvirus.spring.chap05.aop.pojo.LoggingAdvice" />
<aop:config>
    <aop:aspect id="loggingAspect" ref="logging">
        <aop:pointcut
            expression="execution(public * madvirus.spring.chap05.board..*(..))"
            id="publicMethod"/>
        <aop:after-returning method="afterReturning"
            pointcut-ref="publicMethod" />
    </aop:aspect>
</aop:config>
```

```
public void afterReturning() {
    System.out.println("[LA] 메서드 실행 후 후처리 수행");
}
```

# 리턴 값을 사용하고자 할때

```
<aop:after-returning method="afterReturning"  
    pointcut-ref="publicMethod" returning="ret"/>
```

전달받을 파라미터 이름을 명시

```
public void afterReturning(Object ret) {  
    // returnning 속성에 명시한 이름을 갖는 파라미터를 이용해서  
    // 값을 전달 받는다.  
}  
  
public void afterReturning(Article ret) {  
    //리턴 객체의 특정 타입인 경우에 한해서 처리하고자 할때  
}  
  
public void afterReturning(JoinPoint joinPoint, Article ret) {  
    //대상객체 및 호출되는 메서드에 대한 정보나 전달되는 파라미터에  
    //에 대한 정보가 필요할때  
}
```

# After Throwing Advice

- 대상 객체의 메서드가 예외를 발생시킨 경우에 적용
- <aop:after-throwing>태그를 이용

```
<bean id="logging" class="madvirus.spring.chap05.aop.pojo.LoggingAdvice" />
<aop:config>
    <aop:aspect id="loggingAspect" ref="logging">
        <aop:pointcut
            expression="execution(public * madvirus.spring.chap05.board..*(..))"
            id="publicMethod"/>
        <aop:after-throwing method="afterThrowing"
            pointcut-ref="publicMethod"/>
    </aop:aspect>
</aop:config>
```

```
public void afterThrowing() {
    System.out.println("[LA] 메서드 실행 중 예외 발생");
}
```

## 대상 객체의 메서드가 발생시킨 예외 객체가 필요한 경우

```
<aop:after-throwing method="afterThrowing"  
    pointcut-ref="publicMethod" throwing="ex"/>
```

예외를 받을 파라미터 이름을 명시한다.

```
public void afterThrowing(Throwable ex) {  
    System.out.println("[LA] 메서드 실행 중 예외 발생, 예외=" + ex.getMessage());  
}
```

```
public void afterThrowing(ArticleNotFoundException ex) {  
  
}
```

```
public void afterThrowing(JoinPoint joinPoint, Exception ex) {  
  
}
```



# After Advice

- 대상 객체의 메서드가 정상적으로 실행 되었는지 예외를 발생 시켰는지의 여부에 상관없이 적용

```
<aop:config>
  <aop:aspect id="loggingAspect" ref="logging">
    <aop:pointcut
      expression="execution(public * madvirus.spring.chap05.board..*(..))"
      id="publicMethod"/>
    <aop:after method="afterFinally"
      pointcut-ref="publicMethod" />
  </aop:aspect>
</aop:config>
```

```
public void afterFinally() {
```

```
}
```

```
public void afterFinally(JoinPoint joinPoint) {
```

```
}
```

# Around Advice

- Before, After, Returning, After Throwing, After Advice를 모두 구현할 수 있음.
- <aop:around> 태그를 이용하여 설정

```
<bean id="cache" class="madvirus.spring.chap05.aop.pojo.ArticleCacheAdvice" />
<aop:config>
    <aop:aspect id="cacheAspect" ref="cache">
        <aop:around method="cache"
            pointcut="execution(public * *..ReadArticleService.*(..))"/>
        </aop:aspect>
    </aop:config>
```

# Around Advice의 사용 예

```
package madvirus.spring.chap05.aop.pojo;

import java.util.HashMap;

public class ArticleCacheAdvice {

    private Map<Integer, Article> cache = new HashMap<Integer, Article>();

    public Article cache(ProceedingJoinPoint joinPoint) throws Throwable {
        Integer id = (Integer) joinPoint.getArgs()[0];
        Article article = cache.get(id);
        if (article != null) {
            System.out.println("[ACA] 캐시에서 Article[" + id + "] 구함");
            return article;
        }
        Article ret = (Article) joinPoint.proceed();
        if (ret != null) {
            cache.put(id, ret);
            System.out.println("[ACA] 캐시에 Article[" + id + "] 추가함");
        }
        return ret;
    }
}
```

# @Aspect 어노테이션을 이용한 AOP

- AspectJ 5 버전에 추가된 어노테이션
- Xml 파일에 Advice 및 Pointcut 설정을 하지 않고 자동으로 Advice를 적용
- 스프링 2 버전 부터 @Aspect 어노테이션을 지원

# XML 스키마 기반의 AOP와 차이점

- `@Aspect` 어노테이션을 이용해서 Aspect 클래스를 구현한다.
- Aspect클래스는 Advice를 구현한 메서드와 Pointcut을 포함한다.
- XML 설정에서 `<aop:aspectj-autoproxy/>`를 설정한다.

```

@Aspect
public class ProfilingAspect {

    @Pointcut("execution(public * madvirus.spring.chap05.board..*(..))")
    private void profileTarget() {}

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

```

@Pointcut이 적용된 메소드의 리턴값은 void 여야 한다.

```
<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-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

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

```
<!-- Aspect 클래스를 빈으로 등록 -->
```

```
<bean id="performanceTraceAspect"
  class="madvirus.spring.chap05.aop.annot.ProfilingAspect" />
```

```
<bean id="writeArticleService"
  class="madvirus.spring.chap05.board.service.WriteArticleServiceImpl">
  <constructor-arg>
    <ref bean="articleDao" />
  </constructor-arg>
</bean>
```

```
<bean id="articleDao"
  class="madvirus.spring.chap05.board.dao.MySQLArticleDao" />
```

```
<bean id="memberService"
  class="madvirus.spring.chap05.member.service.MemberServiceImpl" />
```

```
</beans>
```

```
public class MainQuickStart2 {  
  
    public static void main(String[] args) {  
        String[] configLocations = new String[] { "acQuickStart2.xml" };  
        ApplicationContext context = new ClassPathXmlApplicationContext(  
            configLocations);  
  
        WriteArticleService articleService = (WriteArticleService) context  
            .getBean("writeArticleService");  
        articleService.write(new Article());  
  
        MemberService memberService = context.getBean("memberService",  
            MemberService.class);  
        memberService.regist(new Member());  
    }  
}
```



# Advice 타입 별 클래스 작성

- Before Advice
- After Returning Advice
- After Throwing Advice
- After Advice
- Around Advice

# Before Advice

- @Before 어노테이션을 사용한다.

```
@Aspect
public class LoggingAspect {

    @Before("execution(public * madvirus.spring.chap05..*(..))")
    public void before() {
        System.out.println("[LA] 메서드 실행 전 전처리 수행");
    }
}
```

@Before 어노테이션 값으로는 AspectJ의 Pointcut 표현식이나 @PointCut 어노테이션이 적용된 메서드이름이 올 수 있다.

Before Advice 구현 메서드는 madvirus.spring.chap05 패키지 또는 그 하위에 있는 모든 public 메서드가 호출되기 전에 호출된다.

# After Returning Advice

- @AfterReturning 어노테이션을 구현 메서드에 적용

```
@Aspect
public class LoggingAspect {

    @AfterReturning("madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()")
    public void afterReturning() {
        System.out.println("[LA] 메서드 실행 후 후처리 수행");
    }
}
```

## 대상 객체가 리턴 한 값을 사용 하고자 할 때

```
@Pointcut("execution(public * exam.*(..))")  
private void loggingTarget() {}
```

```
@AfterReturning(pointcut="loggingTarget()",  
                returning="ret")  
public void afterReturning(JoinPoint joinPoint, Object ret  
{  
    //System.out.println( joinPoint.getSignature().getName()  
    System.out.println(joinPoint.getSignature().getName() + "  
}
```

# 대상객체의 반환 값이 특정 타입인 경우에 한해서 메서드를 실행

```
@Aspect
public class LoggingAspect {

    @AfterReturning(
        pointcut="madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()",
        returning="ret")
    public void afterReturning(Article ret) {
        System.out.println("[LA] 메서드 실행 후 후처리 수행, 리턴값="+ret);
    }
}
```

## 대상 객체 및 호출되는 메서드의 정보가 필요한 경우

```
@Aspect
public class LoggingAspect {

    @AfterReturning(
        pointcut="madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()",
        returning="ret")
    public void afterReturning(JoinPoint joinPoint, | Article ret) {
        System.out.println("[LA] 메서드 실행 후 후처리 수행, 리턴값="+ret);
    }
}
```

# After Throwing Advice

- @AfterThrowing 어노테이션 사용

```
@Aspect
public class LoggingAspect {
    @AfterThrowing("madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()")
    public void afterThrowing() {
        System.out.println("[LA] 메서드 실행 중 예외 발생");
    }
}
```

# 대상객체의 메서드가 발생시킨 예외에 접근

```
@Aspect
public class LoggingAspect {
    @AfterThrowing(
        pointcut="madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()",
        throwing="ex")
    public void afterThrowing(Throwable ex) {
        System.out.println("[LA] 메서드 실행 중 예외 발생, 예외 =" + ex.getMessage());
    }
}
```



# 특정 타입의 예외에 대해서만 처리

```
@Aspect
public class LoggingAspect {
    @AfterThrowing(
        pointcut="madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()",
        throwing="ex")
    public void afterThrowing(FileNotFoundException ex) {
        System.out.println("[LA] 메서드 실행 중 예외 발생, 예외 =" + ex.getMessage());
    }
}
```

# 대상 객체 및 호출 메서드에 대한 정보가 필요한 경우

```
@Aspect
public class LoggingAspect {
    @AfterThrowing(
        pointcut="madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()",
        throwing="ex")
    public void afterThrowing(JoinPoint joinPoint, Exception ex) {
        System.out.println("[LA] 메서드 실행 중 예외 발생, 예외 =" + ex.getMessage());
    }
}
```

# After Advice

- @After 어노테이션 사용

```
@Aspect
public class LoggingAspect {

    @After("madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod() ")
    public void afterFinally() {
        System.out.println("[LA] 메서드 실행 완료");
    }
}
```

```
@Aspect
public class LoggingAspect {

    @After("madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod() ")
    public void afterFinally(JoinPoint joinPoint) {
        System.out.println("[LA] 메서드 실행 완료");
    }
}
```

# Around Advice의 사용 예

```
@Aspect
public class ArticleCacheAspect {

    private Map<Integer, Article> cache = new HashMap<Integer, Article>();

    @Around("execution(public * *..ReadArticleService.*(..))")
    public Article cache(ProceedingJoinPoint joinPoint) throws Throwable {
        Integer id = (Integer) joinPoint.getArgs()[0];
        Article article = cache.get(id);
        if (article != null) {
            System.out.println("[ACA] 캐시에서 Article[" + id + "] 구함");
            return article;
        }
        Article ret = (Article) joinPoint.proceed();
        if (ret != null) {
            cache.put(id, ret);
            System.out.println("[ACA] 캐시에 Article[" + id + "] 추가함");
        }
        return ret;
    }
}
```

# @Pointcut 어노테이션을 이용한 Pointcut설정

```
<aop:config>
  <aop:aspect id="traceAspect1" ref="performanceTraceAdvice">
    <aop:pointcut id="publicMethod"
      expression="execution(public * madvirus.spring.chap05.board..*(..))" />
    <aop:around pointcut-ref="publicMethod" method="trace" />
  </aop:aspect>
</aop:config>
```

```
@Aspect
public class ProfilingAspect {

  @Pointcut("execution(public * madvirus.spring.chap05.board..*(..))")
  private void profileTarget() {}

  @Around("profileTarget()")
  public Object trace(ProceedingJoinPoint joinPoint) throws Throwable {

  }
}
```

Pointcut 어노테이션이 적용된 메소드는 리턴값이 void 여야 하고, 메소드 몸체에 코드를 갖지 않으며, 코드를 가져도 의미가 없다.

# @Pointcut의 참조

- 같은 클래스에서
  - 메서드 이름만 입력
- 같은 패키지에서
  - 클래스이름.메서드이름
- 다른 패키지에서
  - 완전한클래스이름.메서드이름

```
package madvirus.spring.chap05.aop.annot;

import org.aspectj.lang.annotation.Pointcut;

public class PublicPointcut {

    @Pointcut("execution(public * madvirus.spring.chap05..*(..))")
    public void publicMethod() { }

}
```

@Aspect

```
public class LoggingAspect {

    @Before("PublicPointcut.publicMethod()")
    public void before() {
        System.out.println("[LA] 메서드 실행 전 전처리 수행");
    }

    @AfterReturning(
        pointcut = "madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()",
        returning = "ret")
    public void afterReturning(Object ret) {
        System.out.println("[LA] 메서드 실행 후 후처리 수행, 리턴값=" + ret);
    }

}
```

# JoinPoint 사용

- Around Advice를 제외한 나머지 Advice 타입을 구현한 메서드는 JoinPoint 객체를 선택적으로 전달 받을 수 있다.

```
public void afterLogging(Object refVal, JoinPoint joinPoint) {  
}
```

JoinPoint를 첫 번째 매개변수로 받아야 한다. 그렇지 않으면 예외를 발생시킨다



# JoinPoint의 메소드

- Signature getSignature()
  - 호출되는 메서드의 정보
- Object getTarget()
  - 대상 객체를 반환
- Object[] getArgs()
  - 파라미터 목록을 반환

# Signature의 메서드

- `String getName()`
  - 메서드 이름을 반환
- `String toLongString()`
  - 메서드의 리턴타입, 파라미터 타입의 정보를 반환
- `String toShortString()`
  - 메서드의 이름을 반환

# 타입을 이용한 파라미터 접근

- JoinPoint를 사용하지 않고 Advice 메서드에서 직접 파라미터 이용해서 메서드 호출시 사용된 인자에 접근할 수 있다.
1. Advice 구현 메서드에 인자를 전달 받을 파라미터를 명시한다.
  2. Pointcut 표현식에서 args() 명시자를 사용해서 인자 목록을 지정한다.

Advice 구현 메서드에 인자를 전달 받을 파라미터를 명시한다.

```
public class UpdateMemberInfoTraceAspect {  
  
    public void traceReturn(String memberId, UpdateInfo info) {  
        System.out.println("[TA] 정보 수정: 대상회원-" + memberId  
            + ", 수정정보=" + info);  
    }  
}
```

**Pointcut 표현식에서 args() 명시자를 사용해서 인자 목록을 지정한다.**

```
<bean id="traceAdvice"
      class="madvirus.spring.chap05.aop.pojo.UpdateMemberInfoTraceAdvice"/>

<aop:config>
  <aop:aspect id="traceAspect" ref="traceAdvice">
    <aop:after-returning method="traceReturn"
      pointcut="args(memberId, info)"/>
  </aop:aspect>
</aop:config>
```

## Advice가 적용될 메서드

```
public interface MemberService {  
    boolean update(String memberId, UpdateInfo info);  
}
```

# args() 명시자의 타입과 다를 때

```
public interface MemberService {  
    boolean update(String memberId, Object info);  
}
```

메서드 선언에 사용된 객체의 타입이 args() 명시자의 타입과 다르다 하더라도 실제로 메서드에 전달되는 인자의 타입이 args()에서 지정한 것과 동일하다면 Advice가 적용된다.

```
MemberService service = context.getBean("memberService", MemberService.class);  
UpdateInfo updateInfo = new UpdateInfo();
```

```
//실제로 전달되는 객체의 타입이 UpdateInfo이므로 적용됨  
service.update("홍길동", updateInfo);
```

```

@Aspect
public class UpdateMemberInfoTraceAspect {

    @AfterReturning(pointcut="args(memberId,info)", returning="result")
    public void traceReturn(JoinPoint joinPoint, Boolean result,
                           String memberId, UpdateInfo info) {

```

@Aspect 어노테이션을 사용하는 경우에는 Pointcut 표현식에 args()명시자를 사용한다.



# 인자의 이름 매핑 처리

```
@Aspect
public class UpdateMemberInfoTraceAspect {

    @AfterReturning(pointcut="args(memberId, info)", argNames="memberId, info")
    public void traceReturn(String memberId, UpdateInfo info) {
        //
    }
}
```

argNames는 파라미터 이름을 순서대로 표시해서 Pointcut 표현식에서 사용된 이름이 몇번째 파라미터인지 검색할 수 있도록 한다.

@Aspect

```
public class UpdateMemberInfoTraceAspect {
```

```
    @AfterReturning(pointcut="args(memberId,info)", argNames="joinPoint.memberId,info")
```

```
    public void traceReturn(JoinPoint joinPoint, String memberId, UpdateInfo info) {
```

```
        //
```

```
    }
```

첫 번째 파라미터의 타입이 JoinPoint나 ProceedingJoinPoint라면 첫번째 파라미터를 제외한 나머지 파라미터의 이름을 argNames 속성에 입력한다.

```
<aop:after-returning
```

```
    method="traceReturn"
```

```
    pointcut="args(memberId, info)"
```

```
    returning="result" arg-names="joinPoint,memberId,Info"/>
```

**\*\* Pointcut 표현식에서 사용된 파라미터 개수와 실제 구현 메서드의 파라미터 개수가 다르다면 예외 발생.**

# AspectJ의 Pointcut 표현식

```
<bean id="cacheAdvice" class="madvirus.spring.chap05.aop.pojo.ArticleCacheAdvice" />
<aop:config>
    <aop:aspect id="cacheAspect" ref="cacheAdvice">
        <aop:around method="cache"
            pointcut="execution(public * *..ReadArticleService.*(..))"/>
    </aop:aspect>
</aop:config>
```

<aop:태그>를 이용하여 Aspect를 설정하는 경우  
Execution 명시자를 이용하여 Advice가 적용될 Pointcut를 설정함

AspectJ는 Pointcut를 명시할 수 있는 다양한 명시자를 제공하지만  
스프링은 메서드 호출과 관련된 명시자만을 지원함.

# Pointcut 명시자의 종류

- execution
- within
- bean

# execution

## Advice를 적용할 때 메서드를 명시할 때 사용

형식 :

execution(접근명시자 리턴타입 클래스이름?메소드이름(파라미터))

1. 접근명시자 패턴은 생략 가능
2. 각 패턴은 \*을 사용하여 모든 값을 표현할 수 있다.
3. ..을 이용하여 0개 이상이라는 의미를 표현한다.

# execution의 사용 예

- `execution(* madvirus.spring.chap05.*.*())`
  - `madvirus.spring.chap05` 패키지의 파라미터가 없는 모든 메서드 호출
- `execution(* madvirus.spring.chap05..*.*(..))`
  - `madvirus.spring.chap05` 패키지 및 하위 패키지에 있는 파라미터가 0개 이상인 메서드
- `execution(Integer madvirus.spring.chap05..WriteArticleService.write(..))`
  - 리턴 타입이 `Integer`인 `WriteArticleService` 인터페이스의 `write` 메서드 호출
- `execution(* get*(*))`
  - 이름이 `get`으로 시작하고 1개의 파라미터를 갖는 메서드 호출
- `execution(* get*(*,*))`
  - 이름이 `get`으로 시작하고 2개의 파라미터를 갖는 메서드 호출
- `execution(* read*(Integer, ..))`
  - 메서드 이름이 `read`로 시작하고 첫 번째 파라미터 타입이 `Integer`이며 1개 이상의 파라미터를 갖는 메서드 호출

# within 명시자

- 메서드가 아닌 특정 타입에 속하는 메서드를 Pointcut으로 설정 할 때 사용
- **within(madvirus.spring.chap05.board.service.WriteArticleService)**  
WriteArticleService 인터페이스의 모든 메서드 호출
- **within(madvirus.spring.chap05.board.service.\*)**  
madvirus.spring.chap05.board.service 패키지에 있는 모든 메서드 호출
- **within(madvirus.spring.chap05.board..\*)**  
madvirus.spring.chap05.board 패키지 및 하위 패키지에 있는 모든 메서드 호출

# bean 명시자

- 스프링 2.5 버전 부터 추가
- bean 이름을 이용하여 Pointcut를 정의
- **bean(writeArticleService)**  
이름이 writeArticleService인 빈의 메서드 호출
- **bean(\*ArticleService)**  
이름이 ArticleService로 끝나는 빈의 메서드 호출