

# SPRING AOP

**-Aspect Oriented Programming**  
**[관점지향 프로그래밍]**

**백 성 애**

---

# AOP-관점지향 프로그래밍

- × Aspect Oriented Programming
- × 문제를 바라보는 관점을 기준으로 프로그래밍하는 기법.
- × Application을 아래의 두가지 관점에 따라 구현
  - 핵심 관심 사항(core concern)
  - 공통 관심 사항 (cross-cutting concern)
- × 기존 OOP - 공통관심사항을 여러 모듈에서 적용하는데 불편함
- × AOP는 핵심 관심 사항과 공통관심 사항 분리하여 구현

# AOP

공통 기능 코드

```
public void biz(){  
  
    //핵심 로직  
  
    ....  
  
}
```

공통 기능 코드



```
public void biz(){
```

공통기능 코드

```
    //핵심 로직
```

공통기능 코드

```
}
```

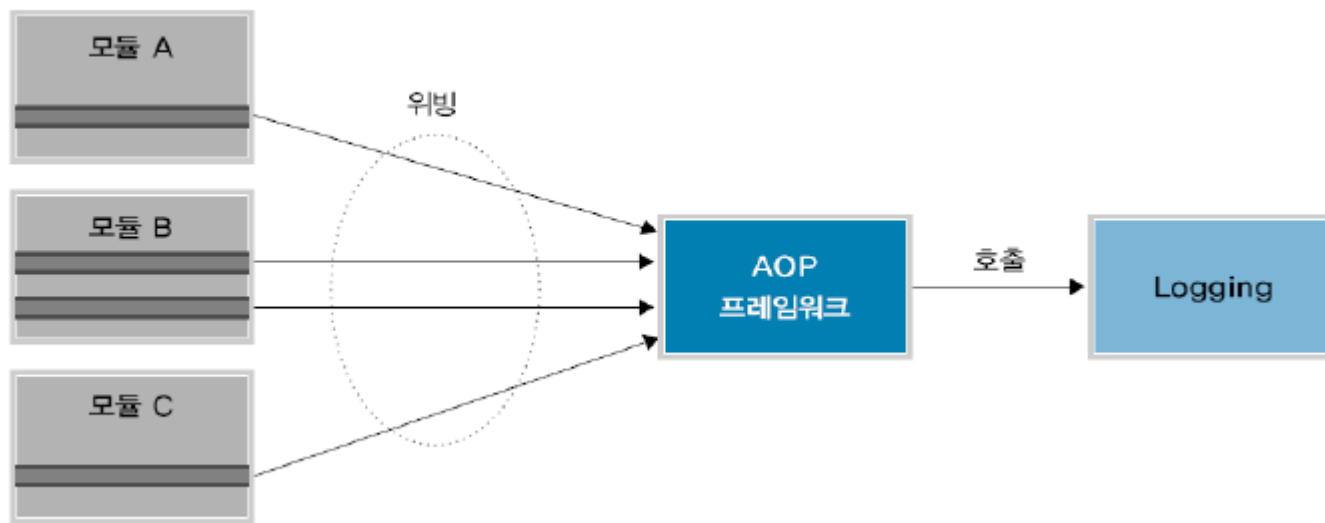
# AOP

- ✖ 핵심 로직을 구현한 코드에서 공통 기능을 직접 호출하는 것이 아니라,
- ✖ 핵심 로직 구현 클래스를 로딩하거나 객체를 생성할 때 AOP가 적용되어 핵심 로직 구현 코드 안에 공통 기능이 삽입된다.
- ✖ 개발자가 핵심 로직에 공통 기능을 삽입할 필요가 없다.
- ✖ Aop 라이브러리가 공통기능을 삽입해 줌



# AOP

- ✖ 핵심 로직을 구현한 코드를 컴파일 하거나,
- ✖ 클래스를 로딩하거나 또는 로딩한 클래스의 객체를 생성할때 핵심 로직 구현 코드 안에 공통기능이 삽입 한다.



# AOP 용어

용어	의미
Joinpoint	공통관심사항이 적용 될 수 있는 지점 Ex] 메소드 호출 시, 객체생성시, 예외 발생 시 등 Advice들을 위빙 하는 포인트
Pointcut	여러 개의 <b>Joinpoint</b> 를 하나로 결합한(묶은) 것 JoinPoint 중 실제 공통사항이 적용될 대상을 지정.
Advice	조인포인트에 삽입되어져 동작할 수 있는 <b>공통관심 사항 코드</b> 로 실제로 그 기능을 구현한 객체 언제 공통 관심 기능을 핵심 로직에 적용할지를 정의함. Around Advice: Joinpoint 앞과 뒤에서 실행 Before Advice: Joinpoint 앞에서 실행 After returning Advice: Jointpoint 메서드 호출이 정상적으로 종료된 뒤에 실행 After Throwing Advice: 예외 발생시 실행

# AOP 용어

용어	의미
Weaving	Advice를 핵심 로직 코드에 적용하는 것을 weaving이라고 한다.
Aspect	공통 관심사항에 대한 추상적인 명칭으로 여러 객체에 공통으로 적용되는 기능을 Aspect라고 한다. 예를 들어 로깅이나 트랜잭션, 보안 등이 Aspect의 좋은 예
Target	핵심 로직을 구현하는 클래스
Advisor	Advice와 Pointcut를 하나로 묶어 취급한

Advice + Pointcut => ASPECT라고 한다 .  
무엇이 어디에 적용 될 것이냐 이것이 관점이다.

# ▣ XML 스키마 기반 AOP 예제

## ◆ 순서

1. 스프링 AOP를 사용하기 위한 의존 추가
2. 공통 기능을 제공한 클래스 구현
3. 핵심 로직을 갖는 Target클래스 구현
4. XML설정 파일에 <aop:config>를 이용해 Aspect를 설정한다. Advice를 어떤 Pointcut에 적용할지를 지정하게 된다.
5. TEST 클래스 작성



# 1. POM.XML에 두 개 의존을 추가

```
<!--AOP -->  
<dependency>  
  <groupId>org.springframework</groupId>  
  <artifactId>spring-aop</artifactId>  
  <version>4.0.4.RELEASE</version>  
</dependency>  
  
<dependency>  
  <groupId>org.aspectj</groupId>  
  <artifactId>aspectjweaver</artifactId>  
  <version>1.8.5</version>  
</dependency>
```

## 2. 공통기능 클래스 ADVICE 작성

```
public class LoggingAdvice {  
    //특정 조인포인트에서 실행될 공통 기능인 trace()메소드 구현.  
    public Object trace(ProceedingJoinPoint joinPoint) throws Throwable{  
        String signStr=joinPoint.getSignature().toShortString();  
        System.out.println(signStr+" 시작");  
        long start=System.currentTimeMillis();  
        try {  
            Object result=joinPoint.proceed();  
            //이 코드를 통해 target객체 메소드 실행  
            return result;  
        }finally{  
            long end=System.currentTimeMillis();  
            System.out.println(signStr+" 종료");  
            System.out.println(signStr+" 실행시간: "+(end-start)+"ms");  
        }  
    }  
}
```

## 2. 공통기능 클래스 LOGGINGADVICE

- ✖ LoggingAdvice 클래스는 Around Advice를 구현한 클래스로서,
- ✖ trace() 메소드는 Advice가 적용될 대상 객체를 호출 하기 전과 후에 시간을 구해서 대상 객체의 메소드 호출 실행 시간을 출력한다.

### 3. TARGET 클래스 작성

✕ 인터페이스 : MessageBean.java

```
1 package my.com;  
2  
3 public interface MessageBean {  
4     void sayHello();  
5 }
```

✕ 타겟 클래스: MessageBeanImpl.java





### 3. TARGET 클래스 작성

```

HelloControl...  hello.jsp  mvc-config.xml  *LoggingAdvi...  *MessageBean..
2  public class MessageBeanImpl implements MessageBean {
3      private String name;
4      private String greeting;
5      public MessageBeanImpl(String name) {
6          super();
7          this.name = name;
8      }
9      @Override
10     public void sayHello() {
11         String msg=greeting+" "+name+"~~";
12         System.out.println(msg);
13     }
14     public String getName() {
15         return name;
16     }
17     public void setName(String name) {
18         this.name = name;
19     }
20     public String getGreeting() {
21         return greeting;
22     }
23     public void setGreeting(String greeting) {
24         this.greeting = greeting;
25     }

```

# 4. XML에 ASPECT 설정

## SRC/MAIN/RESOURCES/LOGGINGADVICE.XML

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

  http://www.springframework.org/schema/bea
ns/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="loggingAdvice"
  class="my.com.LoggingAdvice" />
```

<!-- Aspect 설정: Advice를 어떤 Pointcut에 적용할 지  
설정 -->

```
<aop:config>
```

```
<aop:aspect id="traceAspect" ref="loggingAdvice">
```

```
<aop:pointcut id="publicMethod"
```

```
expression="execution(public * my.com..*(..))" />
```

```
<aop:around pointcut-ref="publicMethod"
  method="trace" />
```

```
</aop:aspect>
```

```
</aop:config>
```

<!-- 타겟 클래스를 빈으로 등록 -->

```
<bean id="msgBean"
```

```
class="my.com.MessageBeanImpl">
```

```
<constructor-arg value="홍길동"/>
```

```
<property name="greeting" value="Bonjour~~"/>
```

```
</bean>
```

```
</beans>
```

## AOP설정에 대한 설명

```
<!-- Aspect 설정: Advice를 어떤 Pointcut에 적용할 시 설정 -->
<aop:config>
  <aop:aspect id="traceAspect" ref="loggingAdvice">
    <aop:pointcut id="publicMethod"
      expression="execution(public * my.com..*(..))" />
    <aop:around pointcut-ref="publicMethod" method="trace" />
  </aop:aspect>
</aop:config>
```

my.com 패키지 및 하위 패키지에 있는 모든 public 메소드를 Pointcut으로 설정한다.

이들 Pointcut에 Around Advice로 loggingAdvice의 trace()메소드를 적용

MessageBean의 public메소드가 호출되면 LoggingAdvice클래스의 trace()메소드가 Around Advice로 적용된다.

◎ xml 각 엘리먼트 의미

<aop:config>: AOP의 설정 정보임을 나타냄

<aop:aspect>Aspect를 설정한다.

<aop:pointcut>: Pointcut을 설정. pointcut설정 시 expression 속성은 pointcut을 정의하는 AspectJ의 표현식을 입력하는데, 이는 뒤에 살펴보자.

<aop:around> : Around Advice를 설정

## 5. TEST클래스 작성-AOPTEST.JAVA

```
package my.com;
import
org.springframework.context.support.GenericXmlApplicationContext;

public class AOPTest {

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

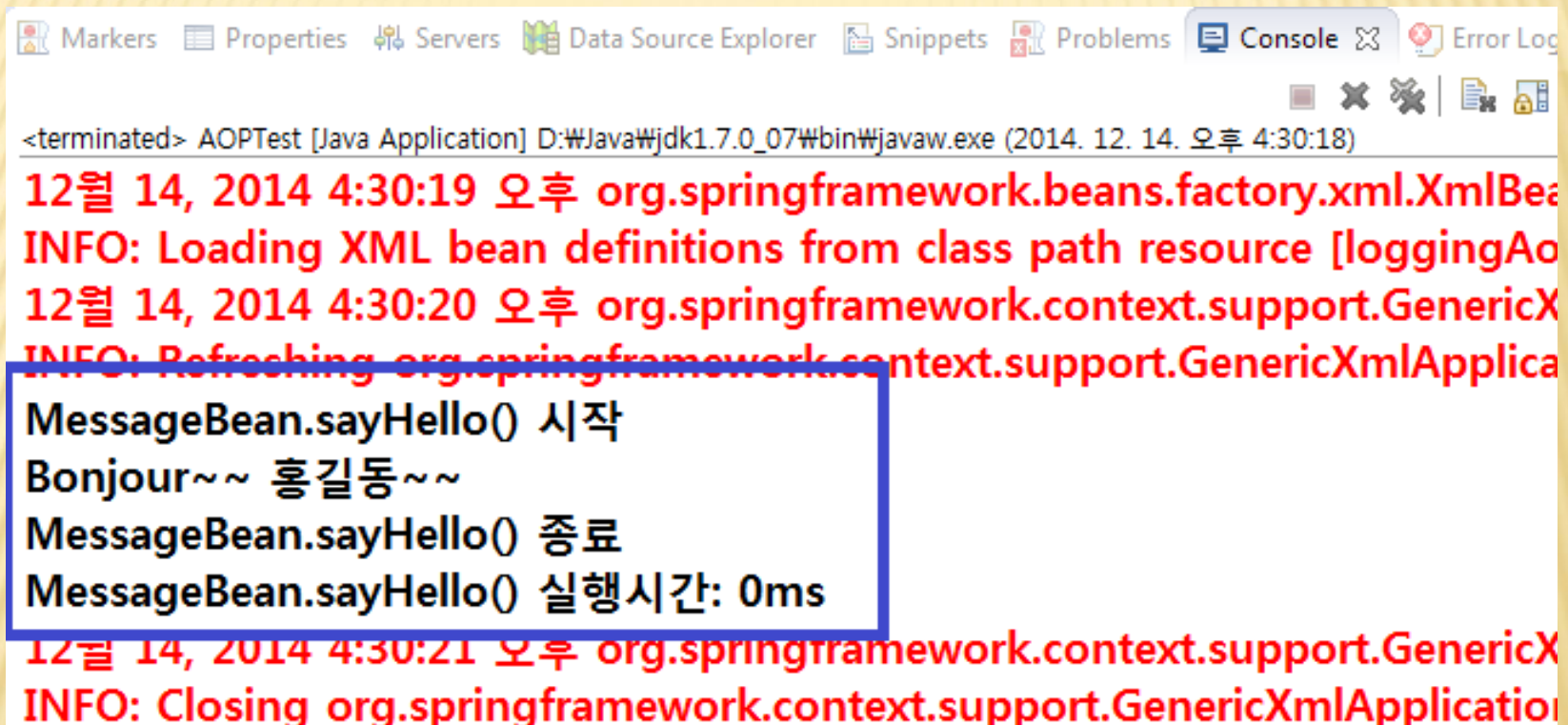
        MessageBean mb=ctx.getBean("msgBean",MessageBean.class);
        mb.sayHello();
        ctx.close();

    }

}
```



## 6. 실행 결과



The screenshot shows an IDE console window with the following content:

```
<terminated> AOPTest [Java Application] D:\Java\jdk1.7.0_07\bin\javaw.exe (2014. 12. 14. 오후 4:30:18)
12월 14, 2014 4:30:19 오후 org.springframework.beans.factory.xml.XmlBea
INFO: Loading XML bean definitions from class path resource [loggingAo
12월 14, 2014 4:30:20 오후 org.springframework.context.support.GenericX
INFO: Refreshing org.springframework.context.support.GenericXmlApplica
MessageBean.sayHello() 시작
Bonjour~~ 홍길동~~
MessageBean.sayHello() 종료
MessageBean.sayHello() 실행시간: 0ms
12월 14, 2014 4:30:21 오후 org.springframework.context.support.GenericX
INFO: Closing org.springframework.context.support.GenericXmlApplication
```

The output text is as follows:

<terminated> AOPTest [Java Application] D:\Java\jdk1.7.0\_07\bin\javaw.exe (2014. 12. 14. 오후 4:30:18)

12월 14, 2014 4:30:19 오후 org.springframework.beans.factory.xml.XmlBea

INFO: Loading XML bean definitions from class path resource [loggingAo

12월 14, 2014 4:30:20 오후 org.springframework.context.support.GenericX

INFO: Refreshing org.springframework.context.support.GenericXmlApplica

MessageBean.sayHello() 시작

Bonjour~~ 홍길동~~

MessageBean.sayHello() 종료

MessageBean.sayHello() 실행시간: 0ms

12월 14, 2014 4:30:21 오후 org.springframework.context.support.GenericX

INFO: Closing org.springframework.context.support.GenericXmlApplication

## [참고] ADVICE정의 관련 태그

태그	설명
<aop:before>	메소드 실행 전에 적용되는 Advice를 정의한다.
<aop:after-returning>	메소드가 정상적으로 실행된 후에 적용되는 Advice를 정의
<aop:after-throwing>	메소드가 예외를 발생시킬 때 적용되는 Advice를 정의한다. Try~catch블럭에서 catch블럭과 비슷함
<aop:after>	메소드가 정상적으로 실행되는지 또는 예외를 발생시키는지 여부에 상관없이 적용되는 Advice를 정의한다. Try~catch~finally블럭에서 finally블럭과 비슷함
<aop:around>	메소드 호출 이전, 이후, 예외 발생 등 모든 시점에 적용 가능한 Advice를 정의한다.

# ■ @ASPECT어노테이션 기반 AOP 예제

## ◆ 순서

1. @Aspect를 이용해서 Aspect클래스를 구현  
이 때 Advice를 구현한 메소드와 Pointcut을 포함한다.
2. XML설정에서 **<aop:aspectj-autoproxy />**를 설정한다.

Cf>(스프링은 객체를 생성할 때 해당 객체와 연관된 것을 연결시켜주기 위해서 프락시 패턴을 사용한다.)

@Aspect어노테이션을 이용할 경우 XML설정 파일에서 Pointcut을 설정하는 것이 아니라 **클래스에서 Pointcut을 정의한다.**

3. Test클래스 작성

# 1. @ASPCET 적용한 LOGGINGASPECT2 클래스

그림

```
8 @Aspect
9 public class LoggingAspect2 {
10
11     @Pointcut("execution(public * my.com..*(..))")
12     private void loggingTarget(){
13         // @Pointcut을 이용하여 pointcut을 정의하면, Advice 관련 어노테이션에서
14         // 해당 메소드 이름을 이용해서 pointcut을 사용할 수 있다.
15         // *주의] @Pointcut이 적용된 메소드는 리턴 타입이 void여야 한다.
16     }
17
18     @Around("loggingTarget()")
19     public Object trace(ProceedingJoinPoint joinPoint) throws Throwable{
20         String signStr=joinPoint.getSignature().toShortString();
21         System.out.println(signStr+" 시작");
22         long start=System.currentTimeMillis();
23         try {
24             Object result=joinPoint.proceed();
25             return result;
26         }finally{
27             long end=System.currentTimeMillis();
28             System.out.println(signStr+" 종료");
29             System.out.println(signStr+" 실행 시간: "+(end-start)+"ms");
30         }
31     }
32 }
```



# 1. @ASPECT 적용 클래스 설명

- ✖ @Pontcut을 이용하여 pointcut을 정의하면, Advice관련 어노테이션에서 해당 메소드 이름을 이용해서 pointcut을 사용할 수 있다.
- ✖ [주의] @Pontcut이 적용된 메소드는 리턴 타입이 void여야 한다.
- ✖ @Around를 이용해서 Around Advice를 구현하는데,
- ✖ 이 때 @Around값으로 @Pointcut을 적용한 메소드 이름을 지정한 것을 확인할 수 있다.
- ✖ trace()메소드는 Around Advice를 구현하게 되며 loggingTarget() 메소드에 정의된 Pointcut에 Advice를 적용하게 된다.

## 2. . XML설정에서 ASPECT클래스를 빈으로 등록

```
<?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">

<aop:aspectj-autoproxy/>
<!-- Aspect클래스를 빈으로 등록 -->
<bean id="loggingAspect" class="my.com.LoggingAspect2" />
<!-- 타겟 클래스를 빈으로 등록 -->
<bean id="msgBean" class="my.com.MessageBeanImpl">
<constructor-arg value="홍길동"/>
<property name="greeting" value="Bonjour~~"/>
</bean>
</beans>
```

### 3. TEST클래스 작성-AOPTEST2.JAVA

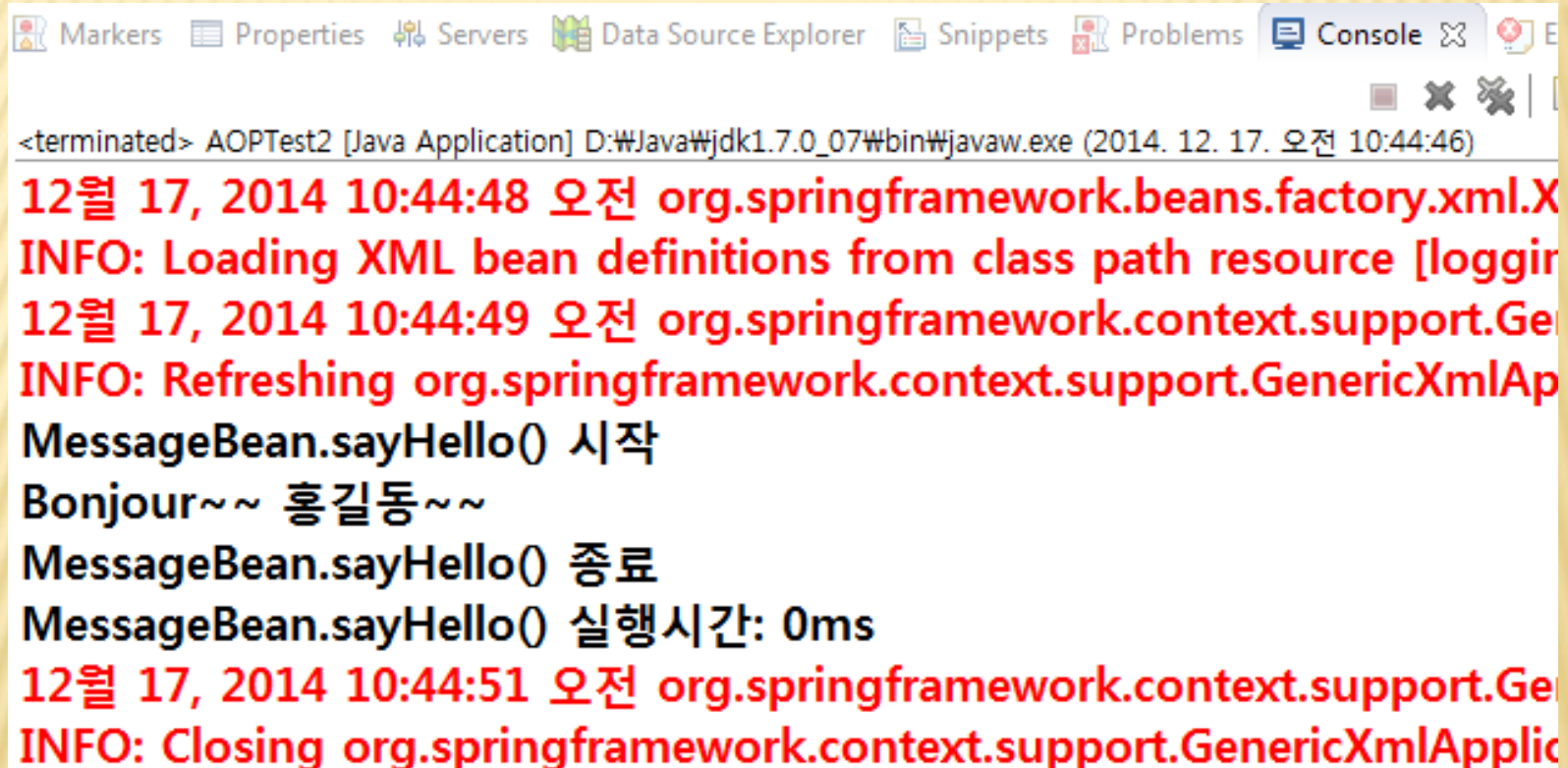
```
package my.com;
import org.springframework.context.support.GenericXmlApplicationContext;

public class AOPTest2 {

    public static void main(String[] args) {
        GenericXmlApplicationContext ctx
        =new GenericXmlApplicationContext("classpath:loggingAop2.xml");

        MessageBean mb=ctx.getBean("msgBean",MessageBean.class);
        mb.sayHello();
        ctx.close();
    }
}
```

## 4. 실행 결과



The screenshot shows an IDE console window with the following content:

```
<terminated> AOPTest2 [Java Application] D:\Java\jdk1.7.0_07\bin\javaw.exe (2014. 12. 17. 오전 10:44:46)
12월 17, 2014 10:44:48 오전 org.springframework.beans.factory.xml.XmlBeanDefinitionReader
INFO: Loading XML bean definitions from class path resource [logging.properties]
12월 17, 2014 10:44:49 오전 org.springframework.context.support.GenericXmlApplicationContext
INFO: Refreshing org.springframework.context.support.GenericXmlApplicationContext
MessageBean.sayHello() 시작
Bonjour~~ 홍길동~~
MessageBean.sayHello() 종료
MessageBean.sayHello() 실행시간: 0ms
12월 17, 2014 10:44:51 오전 org.springframework.context.support.GenericXmlApplicationContext
INFO: Closing org.springframework.context.support.GenericXmlApplicationContext
```



# 현재 JOINPOINT에 접근하기

- ✖ 모든 어드바이스는 `org.aspectj.lang.JoinPoint` 타입의 파라미터를 어드바이스의 첫 파라미터로 선언할 수 있다.
- ✖ `JoinPoint`를 파라미터로 전달받을 경우 반드시 첫 번째 파라미터로 지정해야 함(그 외는 예외 발생)
- ✖ `Around Advice`는 `JoinPoint`의 하위클래스인 `ProceedingJoinPoint` 타입의 파라미터를 필수적으로 첫 파라미터로 선언해야 한다.
- ✖

# JOINPOINT 주요 메소드

메소드	설명
getArgs()	메소드의 파라미터 (아규먼트)를 반환함
getThis()	프록시 객체를 반환
getTarget()	타겟 객체를 반환
getSignature()	호출되는 메서드에 대한 정보를 구함
proceed()	프록시 대상 객체를 호출하는 메소드

Signature를 이용하여 타겟 객체, 메서드 및 전달되는 파라미터에 대한 정보를 구할 수 있음

# ASPECTJ의 POINTCUT 표현식

## □ AspectJ의 Pointcut 표현식

- AspectJ는 Pointcut을 명시할 수 있는 다양한 명시자를 제공.
- 스프링은 메서드 호출과 관련된 명시자만을 지원.

### ■ execution 명시자

- Advice를 적용할 메서드를 명시할 때 사용.

#### ○ 기본 형식

execution(수식어패턴? 리턴타입패턴 패키지패턴?이름패턴(파라미터패턴))

#### ● 수식어 패턴

- 생략가능한 부분.
- public, protected 등이 옴.

#### ● 리턴타입패턴

- 리턴 타입을 명시

#### ● 클래스이름 패턴, 이름패턴

- 클래스 이름 및 메서드 이름을 패턴으로 명시.

#### ● 파라미터패턴

- 매칭될 파라미터에 대해서 명시.

#### ○ 특징

- 각 패턴은 '\*'를 이용하여 모든 값을 표현.
- '..'를 이용하여 0개 이상이라는 의미를 표현.

# ASPECTJ의 POINTCUT 표현식 예

## ○ 설정 예

- `execution(public void set*(..))`
  - 리턴 타입이 `void`이고 메서드 이름이 `set`으로 시작하고, 파라미터가 0개 이상인 메서드 호출.
- `execution(* kame.spring.chap03.core.*,*())`
  - `kame.spring.chap03.core` 패키지의 파라미터가 없는 모든 메서드 호출.
- `execution(*,kame.spring.chap03.core,*,*())`
  - `kame.spring.chap03.core` 패키지 및 하위 패키지에 있는 파라미터가 0개 이상인 메서드 호출.
- `execution(Integer kame.spring.chap03.core.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.

<http://blog.naver.com/PostView.nhn?blogId=chocolleto&logNo=30086024618&categoryNo=29&viewDate=&currentPage=1&listtype=0>



# ASPECTJ의 POINTCUT 표현식

## ■ within 명시자

- 메서드가 아닌 특정 타입에 속하는 메서드를 Pointcut으로 설정할 때 사용.

### ○ 설정 예

- `within(kame.spring.chap03.core, WriteArticleService)`
  - `WriteArticleService` 인터페이스의 모든 메서드 호출.
- `within(kame.spring.chap03.core, *)`
  - `kame.spring.chap03.core` 패키지에 있는 모든 메서드 호출.
- `within(kame.spring.chap03.core, ..*)`
  - `kame.spring.chap03.core` 패키지 및 그 하위 패키지에 있는 모든 메서드 호출.

## ■ bean 명시자

- 스프링 2.5 버전부터 스프링에서 추가적으로 제공하는 명시자.
- 스프링 빈 이름을 이용하여 Pointcut을 정의.
- 빈 이름의 패턴을 갖는다.

### ○ 설정 예

- `bean(writeArticleService)`
  - 이름이 `writeArticleService`인 빈의 메서드 호출.
- `bean(*ArticleService)`
  - 이름이 `ArticleService`로 끝나는 빈의 메서드 호출.

# ASPECTJ의 POINTCUT 표현식 연결

## ■ 표현식 연결

- 각각의 표현식은 '&&' 나 '||' 연산자를 이용하여 연결 가능.
- @Aspect 어노테이션을 이용하는 경우
  - '&&' 연산자를 사용하여 두 표현식을 모두 만족하는 Joinpoint에만 Advice가 적용.

```
@AfterThrowing(  
    pointcut = "execution(public * get*()) && execution(public void set*(..))"  
    public void throwingLogging() {  
        ...  
    }
```

- XML 스키마를 이용하여 Aspect를 설정하는 경우
  - '&&'나 '||' 연산자를 사용.

```
<aop:pointcut id="propertyMethod"  
    expression="execution(public * get*()) && execution(public void set*(..))" />
```

- XML 문서이기 때문에 값에 들어가는 '&&' '||'를 '&amp;&amp;'로 표현.
- 설정파일에서 '&&'나 '||' 대신 'and'와 'or'를 사용할 수 있도록 하고 있음.

```
<aop:pointcut id="propertyMethod"  
    expression="execution(public * get*()) and execution(public void set*(..))" />
```

# 참고

---

- ✖ aop관련 api문서
- ✖ <http://www.eclipse.org/aspectj/doc/next/runtime-api/overview-summary.html>
- ✖ <http://docs.spring.io/spring/docs/current/java-doc-api/org/aopalliance/intercept/MethodInterceptor.html>