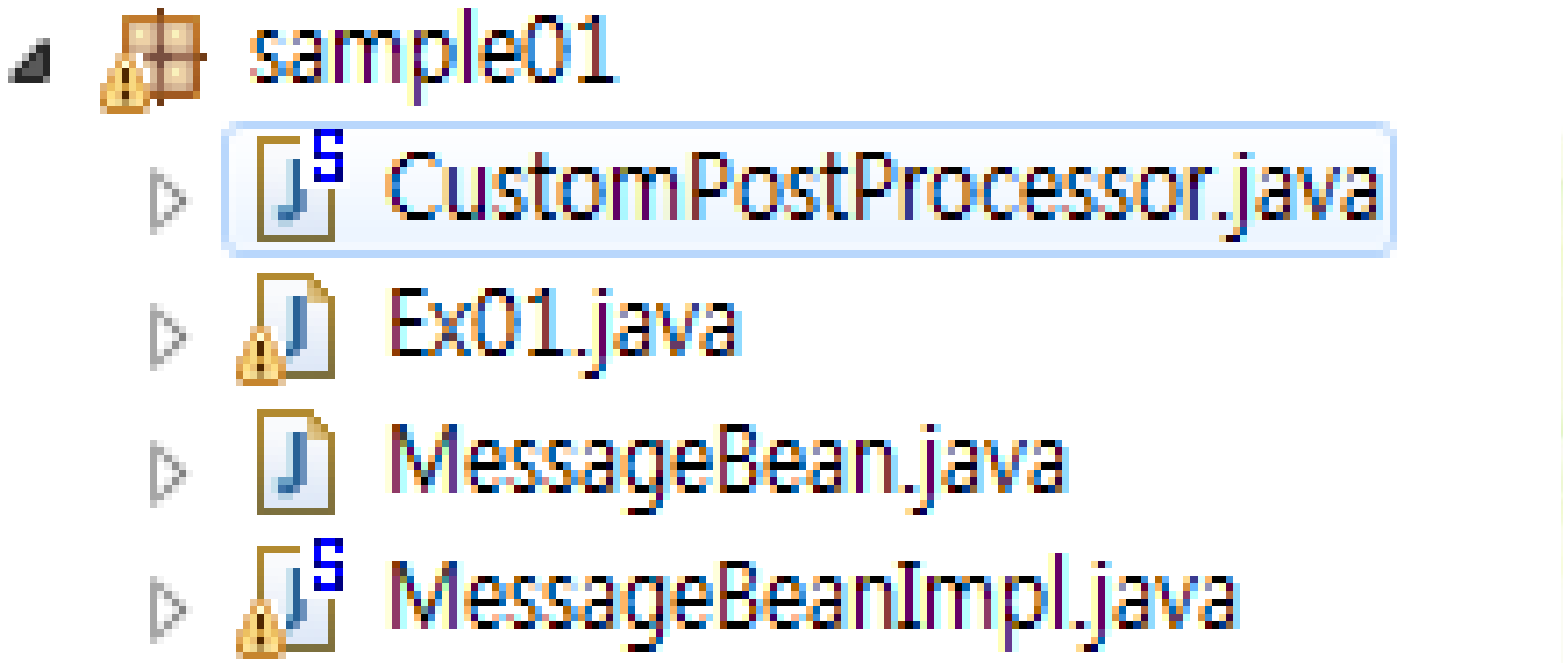
빈의 Life Cycle

강사 : 강병준

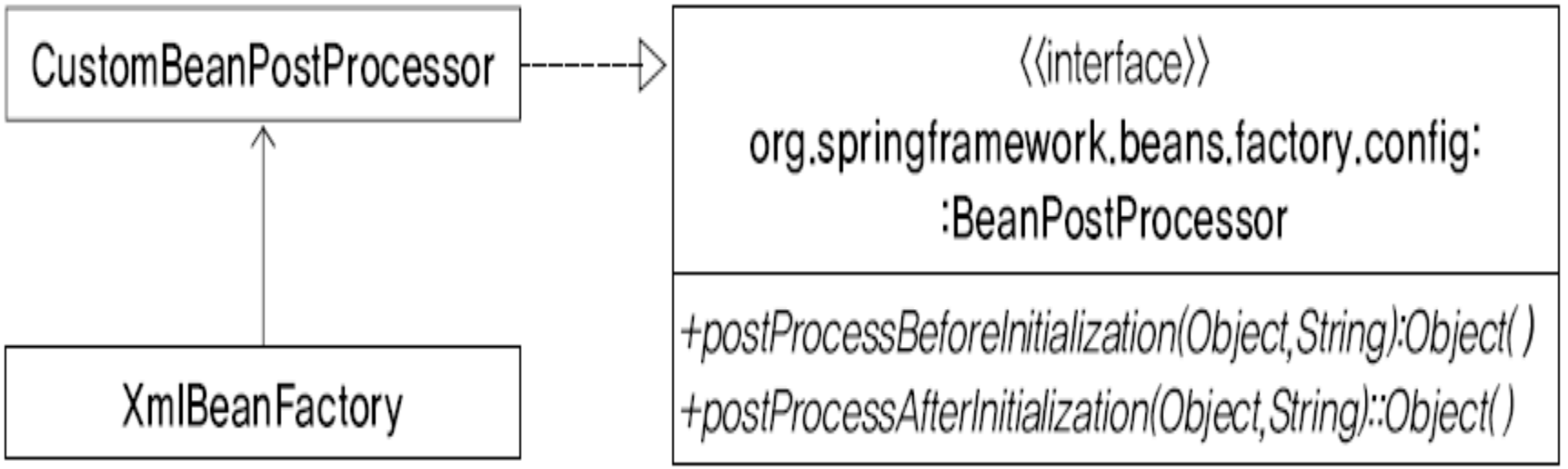
빈의 라이프사이클

빈의 라이프사이클과 스프링이 호출한 메소드의 호출 타이밍과 순서에 대해서 살펴보자.
MessageBeanImpl 클래스는 **org.springframework.beans.factory** 패키지에서 제공되고 있는 **BeanNameAware, BeanFactoryAware, InitializingBean, DisposableBean 4**개 인터페이스가 설정되어 있다.



빈의 라이프사이클

org.springframework.beans.factory.config.BeanPostProcessor 인터페이스를 갖는 **CustomBeanPostProcessor** 클래스를 작성하여 빈 팩토리와 관련 짓겠다.



▲ 클래스 그림(2)

빈의 라이프사이클

beans.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"
        xsi:schemaLocation="http://www.springframework.org/schema
/beans http://www.springframework.org/schema/beans/spring-
beans.xsd">
        <bean class="samp01.CustomerBeanPostProcessor"></bean>
        <bean id="mb" class="samp01.MessageBeanImpl"
                        init-method="init">
                <property name="name" value="홍길동"></property>
        </bean>
</beans>
```

빈의 라이프사이클

```
package sample1;

import org.springframework.beans.factory.config.*;

public class CustomBeanPostProcessor implements BeanPostProcessor {

   public Object postProcessBeforeInitialization(Object bean, String beanName) {
      System.out.println("⑤ 초기화 전 Bean에 대한 처리 실행");
      return bean;
   }

   public Object postProcessAfterInitialization(Object bean, String beanName) {
      System.out.println("⑧ 초기화 후 Bean에 대한 처리 실행");
      return bean;
   }
}
```

빈의 라이프사이클

```
package sample1;
import org.springframework.beans.factory.*;
public class MessageBeanImpl implements MessageBean,
        BeanNameAware, BeanFactoryAware,
        InitializingBean, DisposableBean {
    private String greeting;
    private String beanName;
    private BeanFactory beanFactory;

    public MessageBeanImpl() {
        System.out.println("① Bean의 생성자 실행");
    }
    public void setGreeting(String greeting) {
        this.greeting = greeting;
        System.out.println("② 세터 메서드 실행");
    }
    public void setBeanName(String beanName) {
        System.out.println("③ Bean명 지정");
        this.beanName = beanName;
        System.out.println(" -> " + beanName);
    }
```

빈의 라이프사이클

```
public void setBeanFactory(BeanFactory beanFactory) {
        System.out.println("④ BeanFactory 지정");
        this.beanFactory = beanFactory;
        System.out.println(" -> " +beanFactory.getClass());
    }
    public void init() {
        System.out.println("⑦ 초기화 메서드 실행");
    }

    public void destroy() {
        System.out.println("종료");
    }

    public void afterPropertiesSet() {
        System.out.println("⑥ 프로퍼티 지정 완료");
    }

    public void sayHello() {
        System.out.println(greeting + beanName + "!");
    }
}
```

빈의 라이프사이클

다음으로 빈 팩토리에 XmlBeanFactory 클래스가 빈(MessageBeanImpl)을 생성하고 그 빈이 이용 가능한 상태가 될 때까지의 흐름을 보도록 하겠다.

① 빈의 인스턴스화(생성자 호출)
② 필드값 설정
③ setBeanName() 메소드 호출(BeanNameAware 인터페이스를 설치하고 있을 경우)
④ setBeanFactory() 메소드 호출(BeanFactoryAware 인터페이스를 설치하고 있을 경우)
⑤ BeanPostProcessor의 postProcessBeforeInitialization() 메소드 호출(BeanFacotry에 BeanPostProcessor 클래스가 관련되어 있을 경우

빈의 라이프사이클

⑥ afterPropertiesSet() 메소드 호출(InitializingBean 인터페이스를 설치하고 있을 경우)

⑦ Custom 초기화 메소드 호출(Custom 초기화 메소드가 정의되어 있을 경우)

⑧ BeanPostProcessor의 postProcessAfterInitialization() 메소드 호출(BeanFactory에 BeanPostProcessor 클래스가 관련되어 있을 경우)

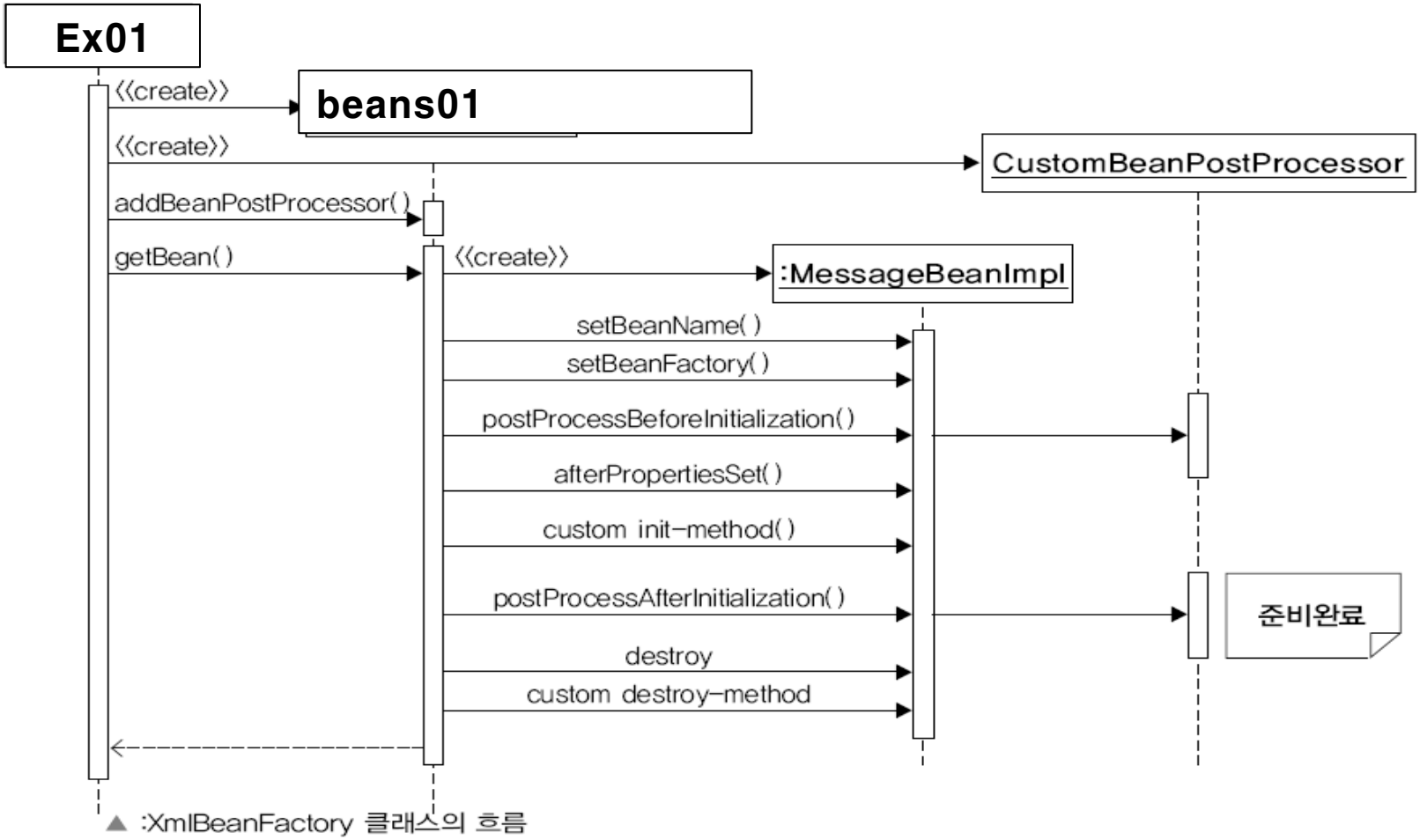
그리고 컨테이너가 종료할 때에는 다음 순서로 메소드가 호출된다.

① destroy() 메소드 호출 (DisposableBean 인터페이스를 설치하고 있는 경우)

② Custom Destroy 메소드 호출

빈의 라이프사이클

▼ 시퀀스 그림



▲ :XmlBeanFactory 클래스의 흐름

ApplicationContext 인터페이스

org.springframework.context.ApplicationContext 는 BeanFactory 인터페이스의 서브 인터페이스로 여러 개의 편리한 기능이 추가되었다.
추가된 기능은 다음과 같다.

- 메시지의 국제화
- 리소스로의 액세스 수단 간편화
- 이벤트 처리
- 복수 context 로드

```
package samp01;

import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

public class Ex01 {
	public static void main(String[] args) {
		AbstractApplicationContext ac = new

	GenericXmlApplicationContext("/samp01/beans01.xml");
		MessageBean mb = ac.getBean(MessageBean.class);
		mb.sayHello();
		ac.close();
	}
}
```

// ApplicationContext는 BeanPostProcessor를 자동으로 읽어 들입니다
이를 위해서 beans.xml에 ***CustomBeanPostProcessor의 bean정의를 추가***

스프링의 AOP

강사 : 강병준

관점 지향 AOP

관점 지향 프로그래밍(Aspect Oriented Programming, 이하 AOP)은 결국 객체 지향 프로그래밍(Object Oriented Programming의 뒤를 있는 또 하나의 프로그래밍 언어 구조라고 생각될 수 있다.

AOP는 OOP를 대신 하기 위한 것이 아니라 OOP를 더욱 OOP답게 만들어 준다.

어플리케이션을 개발하다 보면 로깅, 트랜잭션처리, 보안 등 공통으로 사용하는 기능이 필요한 경우가 있다.

AOP는 공통관심사항을 여러 모듈에 효과적으로 적용하기 위해서 사용하는 기법으로 의존관계의 복잡성과 코드 중복을 해소해 준다. AOP는 각 클래스에서 공통관심사항을 구현한 모듈에 대한 의존관계를 갖기보다는 Aspect를 이용하여 핵심 로직을 구현한 각 클래스에 공통기능을 적용하게 된다

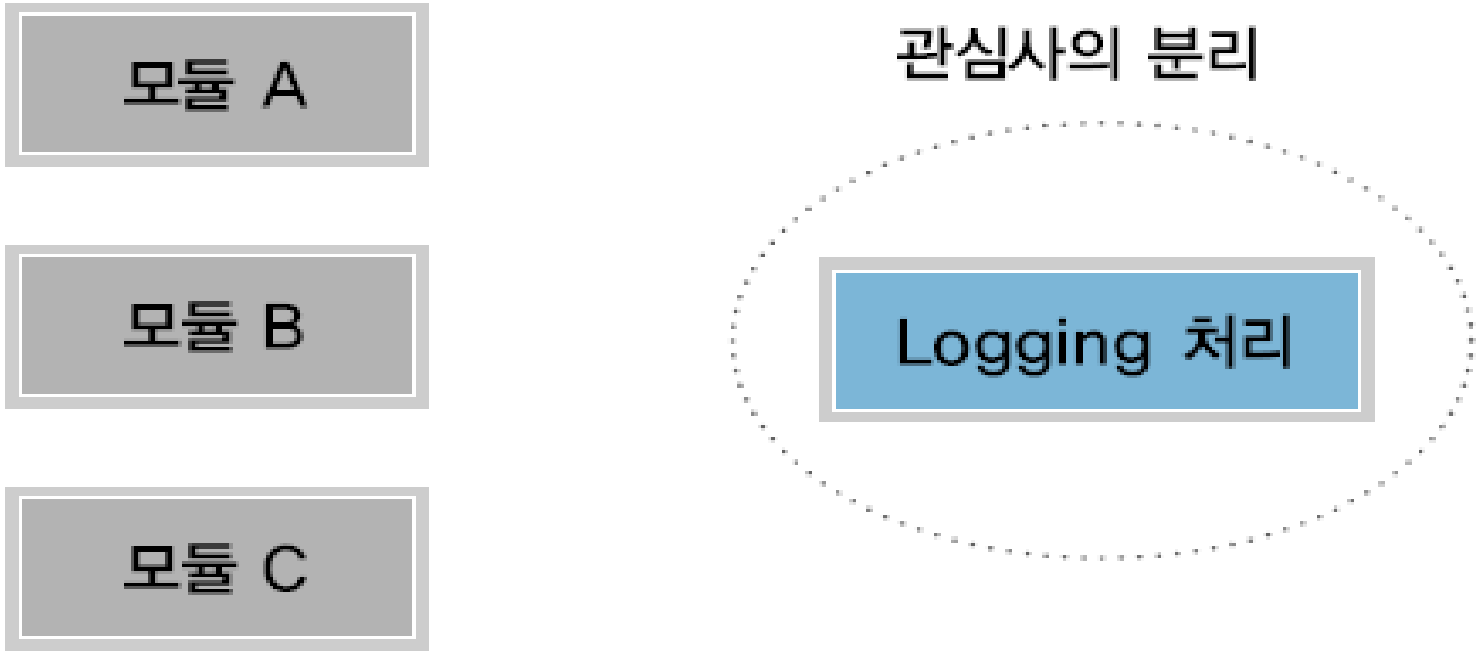
Aspect 지향과 횡단 관점 분리

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 「로깅」, 「보안」, 「트랜잭션」 등의 처리는 어플리케이션 전반에 걸쳐 필요한 기능으로 핵심 비즈니스 기능과는 구분하기 위해서 공통 관심 사항(Cross-Cutting Concern)이라고 표현한다.

Aspect 지향과 횡단 관점 분리

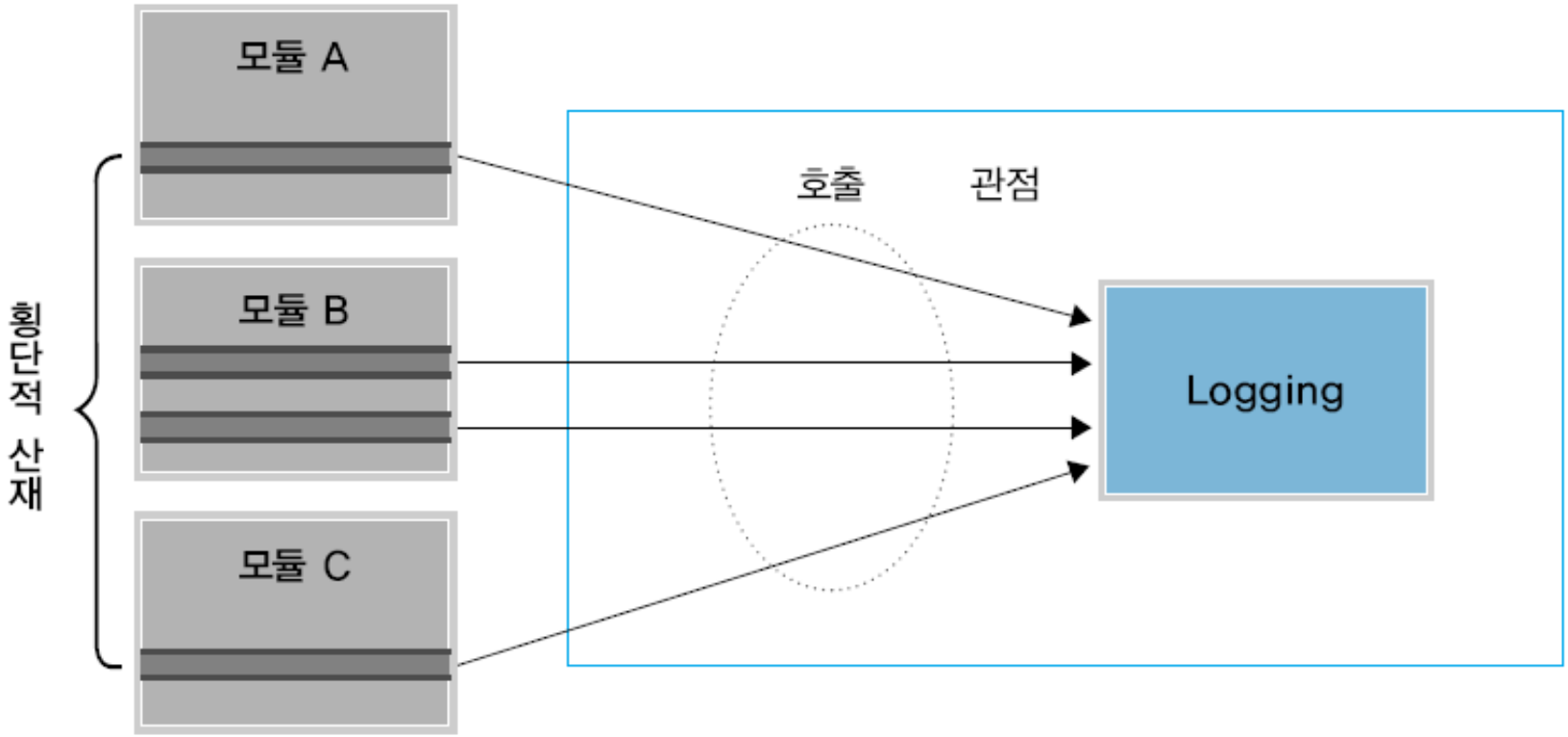
오브젝트 지향에서는 이들 업무들을 하나의 클래스라는 단위로 모으고 그것들을 모듈로부터 분리함으로써 재이용성과 보수성을 높이고 있다.



▲ 기존의 객체 지향에서 관점의 분리

Aspect 지향과 횡단 관점 분리

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.

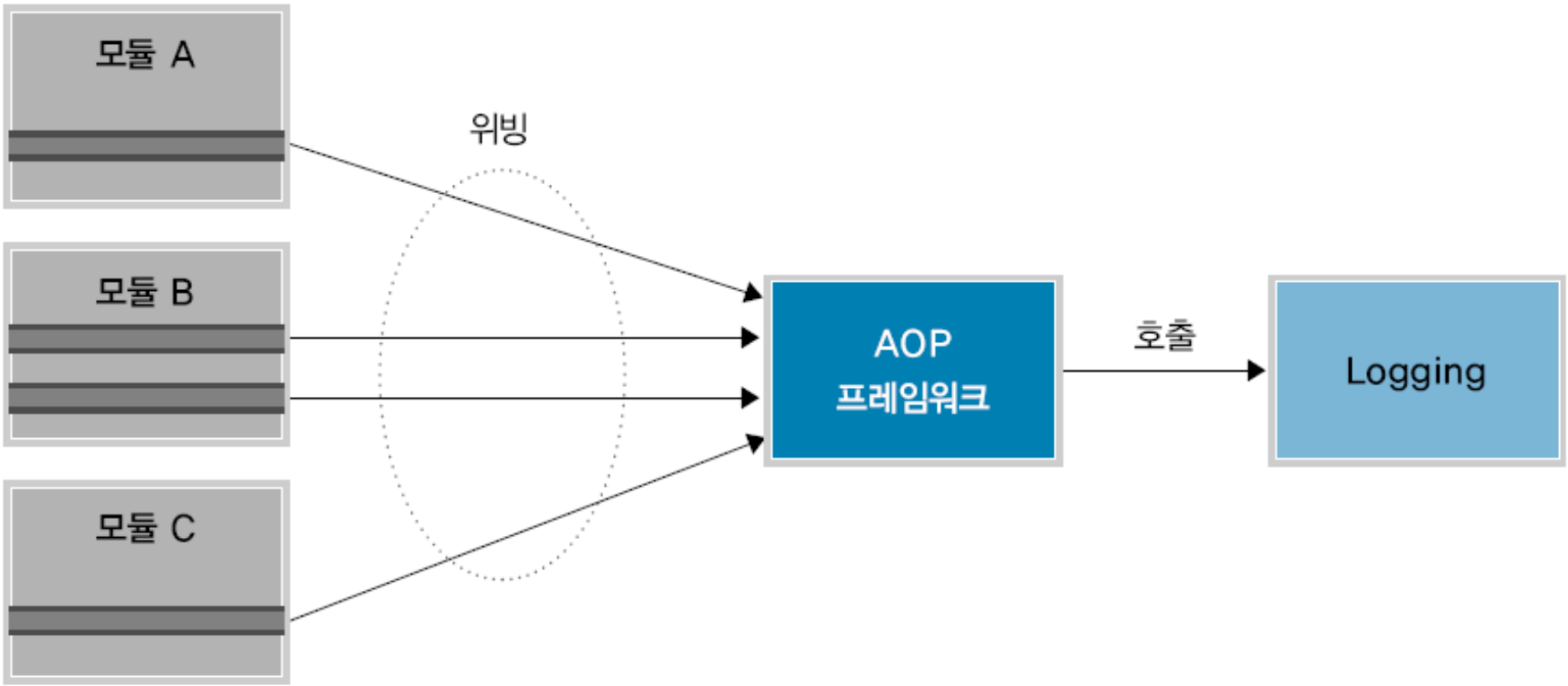


▲ 횡단적으로 산재하는 '기능의 호출'

Aspect 지향과 횡단 관점 분리

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.

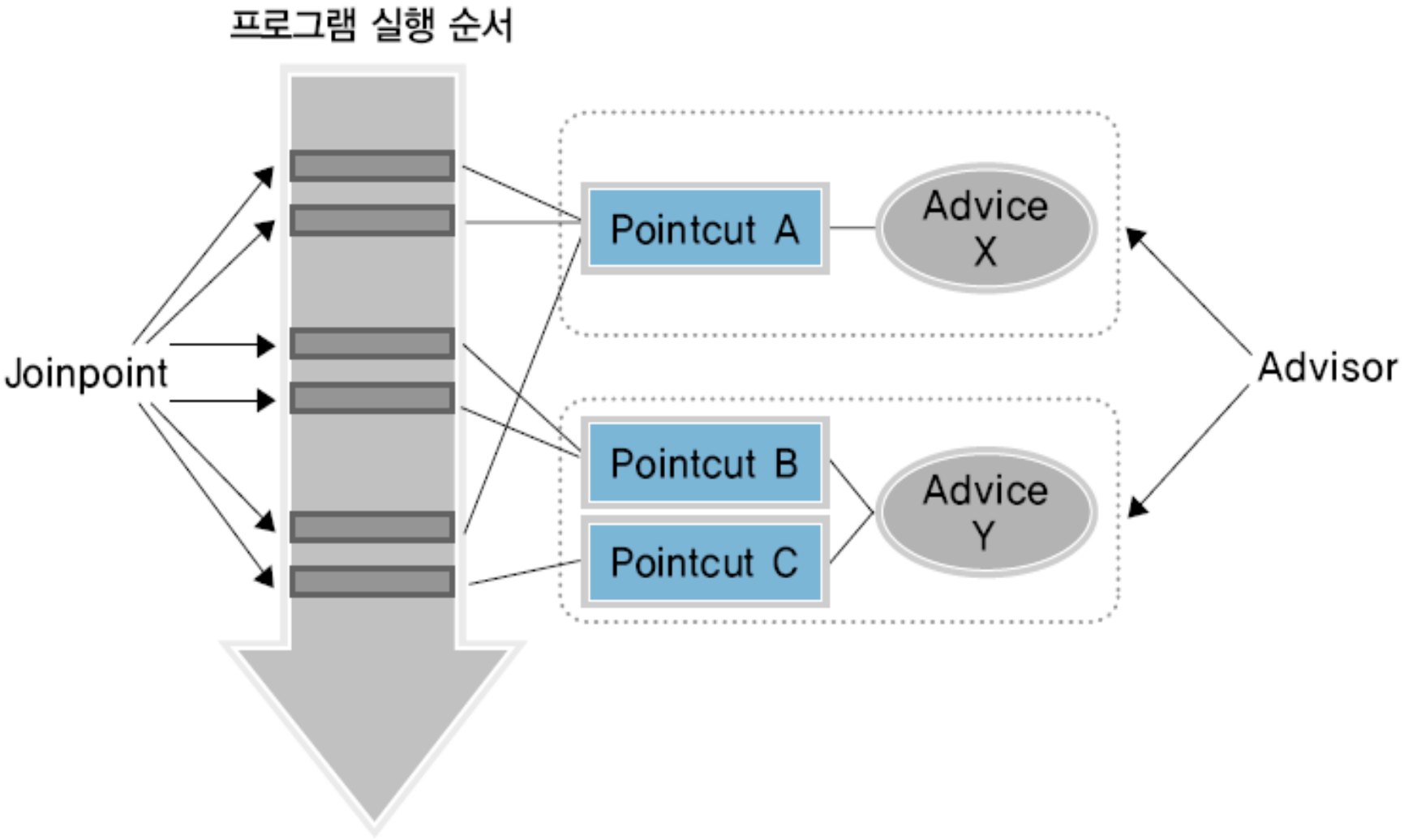


▲ AOP의 횡단 관점의 분리와 위빙

스프링 AOP에서의 용어

1. Aspect – 여러 객체에 공통으로 적용되는 공통 관점 사항을 Aspect라 부른다. 횡단관심사와 횡단관심사를 적용하는 소스코드상의 포인트를 모은 것. 하나 또는 그 이상의 어드바이스(동작)와 포인트 컷(동작을 적용하는 조건)을 조합한 것
2. Joinpoint – 「클래스의 인스턴스 생성 시점」, 「메소드 호출 시점」 및 「예외 발생 시점」과 같이 애플리케이션을 실행할 때 특정 작업이 시작되는 시점을 Joinpoint라 한다. 즉 어드바이스가 실행히는 동작을 끼어 넣을 수 있는 때
3. Advice – Joinpoint에 삽입되어져 동작할 수 있는 코드를 Advice라 한다.
4. Pointcut – 여러 개의 Joinpoint를 하나로 결합한 것을 Pointcut이라고 부른다.
5. Advisor – Advice와 Pointcut를 하나로 묶어 취급한 것을 Advisor라 부른다.
6. Weaving – Advice를 핵심 로직 코드에 삽입하는 것을 Weaving이라 부른다.
7. Target – 핵심 로직을 구현하는 클래스를 말한다.
8. 위빙(weaving) : 분리된 관심사를 모듈에 삽입하는 것, 각 모듈에서 분리된 기능을 사용하기 위한 코드를 기술할 필요 없음

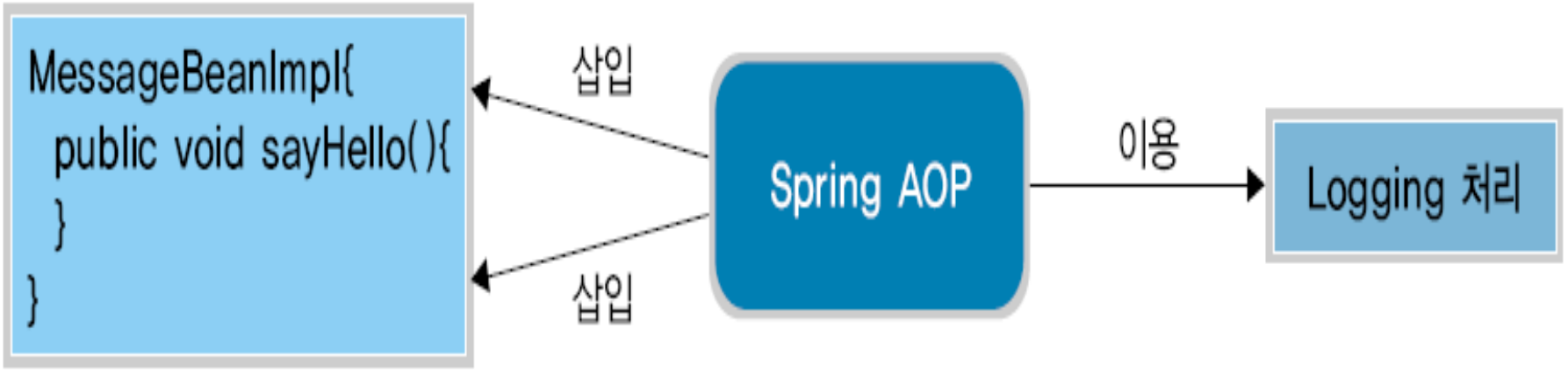
스프링 AOP에서의 용어



▲ 스프링 AOP에서의 용어와 개념

AOP를 이용한 logging 구현 예제

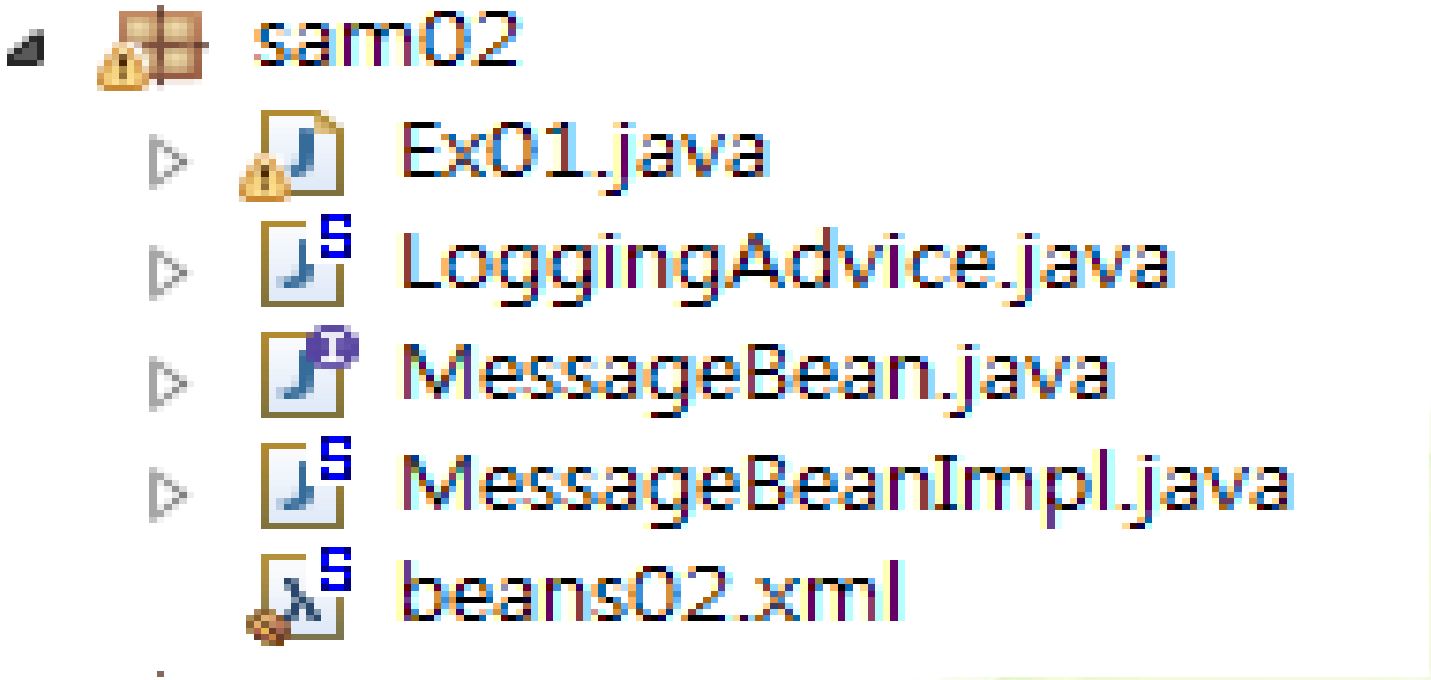
이 예제에서는 AOP 구조를 활용하여 메소드 트레이스 정보의 Logging 처리를 MessageBeanImpl의 sayHello() 메소드 호출 전후에 삽입한다.
로깅 처리 자체 및 그 호출은 MessageBeanImpl에는 기술하지 않는다.
스프링이 제공하는 기능인 「스프링 AOP」가 그 역할을 담당한다.



▲ 예제 개요 그림

AOP를 이용한 logging 구현 예제

▼ 파일 구성



sayHello() 메소드가 핵심로직이고 이 메소드를 멤버로 갖는 MessageBeanImpl는 타켓 클래스가 된다.
LoggingAdvice 클래스가 로깅처리를 담당하고 있게 된다.

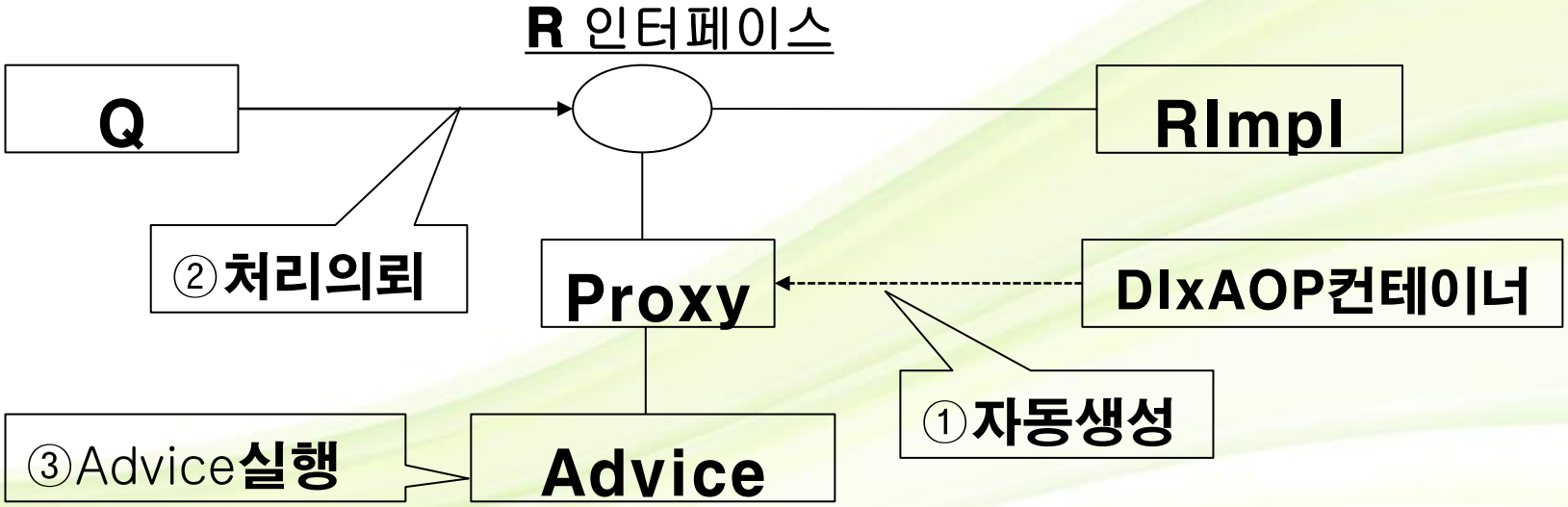
스프링 API를 이용한 AOP

스프링에서 AOP를 구현하는 과정은 다음과 같다.

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

프록시를 이용한 AOP

1. 인터페이스를 구현한 프록시를 이용하여 **Q**클래스가 호출한 메소드를 가로채 어드바이스를 동작
2. **DI**를 이용하기 위하여 인터페이스 **R**을 구현한 **RImpl**을 준비
3. 그 클래스를 이용하는 것은 **Q**클래스이며 **Q**클래스에는 **R**인터페이스 타입의 인스턴스 변수 준비
4. **DI/AOP**컨테이너는 **R**인터페이스를 구현한 프록시 클래스의 인터페이스를 자동으로 생성해서 **Q**클래스의 **R**인터페이스형 인스턴스 변수에 인젝션
5. **Q**클래스는 **R**인터페이스를 구현한 클래스의 인스턴스가 인젝션 되므로 그 인스턴스가 진짜 **Rimpl**클래스의 모르게 된다.
6. 자동 생성된 프록시 클래스의 인스턴스는 진짜**Rimpl**클래스로 구현된 메소드를 호출하게 구현되어 있고 종류에 따라 **RImpl**의 메소드를 호출하기 전후에 어드바이스를 호출하게 되어있다



AOP

1. 스프링에서는 자체적으로 Proxy기반의 AOP를 지원
2. 스프링의 AOP는 메서드 호출 JoinPoint 만을 지원(필드 값 변경 시 AOP를 사용하고자 할 때는 Aspect J와 같은 별도의 AOP 도구를 이용)
3. AOP의 적용대상이 되는 객체에 직접 접근하는 것이 아니고 Proxy를 통해서 간접적으로 접근하며 Proxy를 이용하지 않고도 AOP 구현 가능
4. 스프링에서 AOP 구현 방법 – final class에는 적용이 불가
    1) XML 기반의 POJO 클래스 또는 MethodInterceptor 인터페이스를 이용한 방법
    2) @Aspect Annotation 기반의 구현
    3) 스프링 API를 이용한 AOP 구현

❖구현 가능한 Advice

1) Before Advice: 대상 객체의 메서드 호출 전에 공통 기능 실행
2) After Returning Advice: 대상 객체의 메서드가 예외 없이 실행한 이후에 공통 기능 실행
3) After Throwing Advice: 대상 객체의 메서드가 예외를 발생한 경우에 공통 기능 실행
4) After Advice: 대상 객체의 메서드를 실행하는 도중에 예외가 발생했는지의 여부와 상관없이 메서드 실행 후 공통 기능 실행
5) Around Advice: 대상 객체의 메서드 실행 전, 후 또는 예외 발생 시점에 공통 기능 실행

AOP beans.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"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
        <bean id="mb" class="samp02.MessageBeanImpl">
                <property name="name" value="홍길동"></property>
        </bean>
        <bean id="la" class="samp02.LogAdvice"></bean>
        <bean id="proxy"
        class="org.springframework.aop.framework.ProxyFactoryBean">
                <property name="target" ref="mb" />
                <property name="interceptorNames" value="advisor"></property>
        </bean>
        <bean id="advisor"
        class="org.springframework.aop.support.DefaultPointcutAdvisor">
          <property name="advice" ref="la"></property>
          <property name="pointcut">
            <bean
class="org.springframework.aop.support.JdkRegexpMethodPointcut">
                        <property name="pattern">
                                <value>.*sayHello.*</value>
                        </property>
            </bean>
          </property>
        </bean>
</beans>
```

AOP

```
package sample1;
public interface MessageBean {
  void sayHello();
}

package sample1;
public class MessageBeanImpl implements MessageBean {

    private String name;
    public void setName(String name) {
        this.name = name;
    }
    public void sayHello() {
        try {
            Thread.sleep(5000);
        } catch(InterruptedException e) {}
        System.out.println("Hello, " + name + "!");
    }
}
```

AOP

```
package samp02;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.util.StopWatch;
public class LogAdvice implements MethodInterceptor {
	public Object invoke(MethodInvocation invocation) throws Throwable {
		String methodName = invocation.getMethod().getName();
		StopWatch sw = new StopWatch();
		sw.start(methodName);
		System.out.println("작업시작 : "+methodName);

		// 원작업 실행
		Object obj = invocation.proceed();

		sw.stop();
		System.out.println("작업종료:"+sw.getTotalTimeSeconds()+"초");
		return obj;
	}
}
```

AOP

```
package samp02;
import
org.springframework.context.support.AbstractApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class Ex01 {
    public static void main(String[] args) {
        AbstractApplicationContext ac = new

        ClassPathXmlApplicationContext("/samp02/beans02.xml");
        MessageBean mb = ac.getBean("proxy",MessageBean.class);
        mb.sayHello();
        ac.close();
    }
}
```

AOP

Advice타입

AroundAdvice, BeforeAdvice, AfterRunningAdvice, AfterThrowingAdvice, Introduction

Advice 인테페이스 상속 계층

Advice(모두 interface)

|

Interceptor

|

MethodIntercepter　　FieldInterceptor　　ConstructorInterceptor

Joinpoint인터페이스 상속계층

Jointpoint(getStaticPart(), getThis(), proceed())

|

Invocation(+GETArguments())

|

MethodInvocation(+getMethod())

어노테이션을 이용한 AOP

강사 : 강병준

AOP 설정 태그

<aop:config>

<aop:pointcut>, <aop:aspect>, <aop:advisor> 태그를 포함

<aop:pointcut>

포인트컷은 공통코드가 실행되어야 할 핵심코드를 지정할 때 사용

속성

id : 각 포인트컷의 고유 아이디 값

expression : 포인트컷 표현식을 지정

<aop:aspect>

공통코드 객체를 지정

어드바이스와 포인트컷을 연결

속성

id : aspect 고유 아이디 값

order : 특정 포인트컷에 여러 개 어드바이스가 실행될 때 **aspect**의 순서를 제어

ref : 포인트컷의 아이디를 지정

<aop:pointcut> 태그와

<aop:after>, <aop:after-returning>, <aop:after-throwing>, <aop:around>, <aop:before> 등 어드바이스 태그를 포함

어드바이스

<aop:before>

before 어드바이스는 핵심코드가 실행되기 전에 공통코드가 실행

예: **<aop:before pointcut-ref="hello" method="log" />**

<aop:after>

after 어드바이스는 핵심코드가 실행된 후(리턴값이 없을 경우 또는 **finally** 블록이 실행된 후) 공통코드가 실행

예: **<aop:after pointcut-ref="hello" method="log" />**

<aop:after-returning>

after-returning 어드바이스는 핵심코드 메서드가 리턴한 다음 공통코드가 실행

예: **<aop:after-returning printcut-ref="hello" method="resultLog" return="resultObj" />**

메서드 예: **public void resultLog(Object resultObj) { … }**

<aop:after-throwing>

after-throwing 어드바이스는 핵심코드에서 예외가 발생할 경우 공통코드가 실행

예: **<aop:after-throwing printcut-ref="xxx" method="exceptionLog" throwing="ex" />**

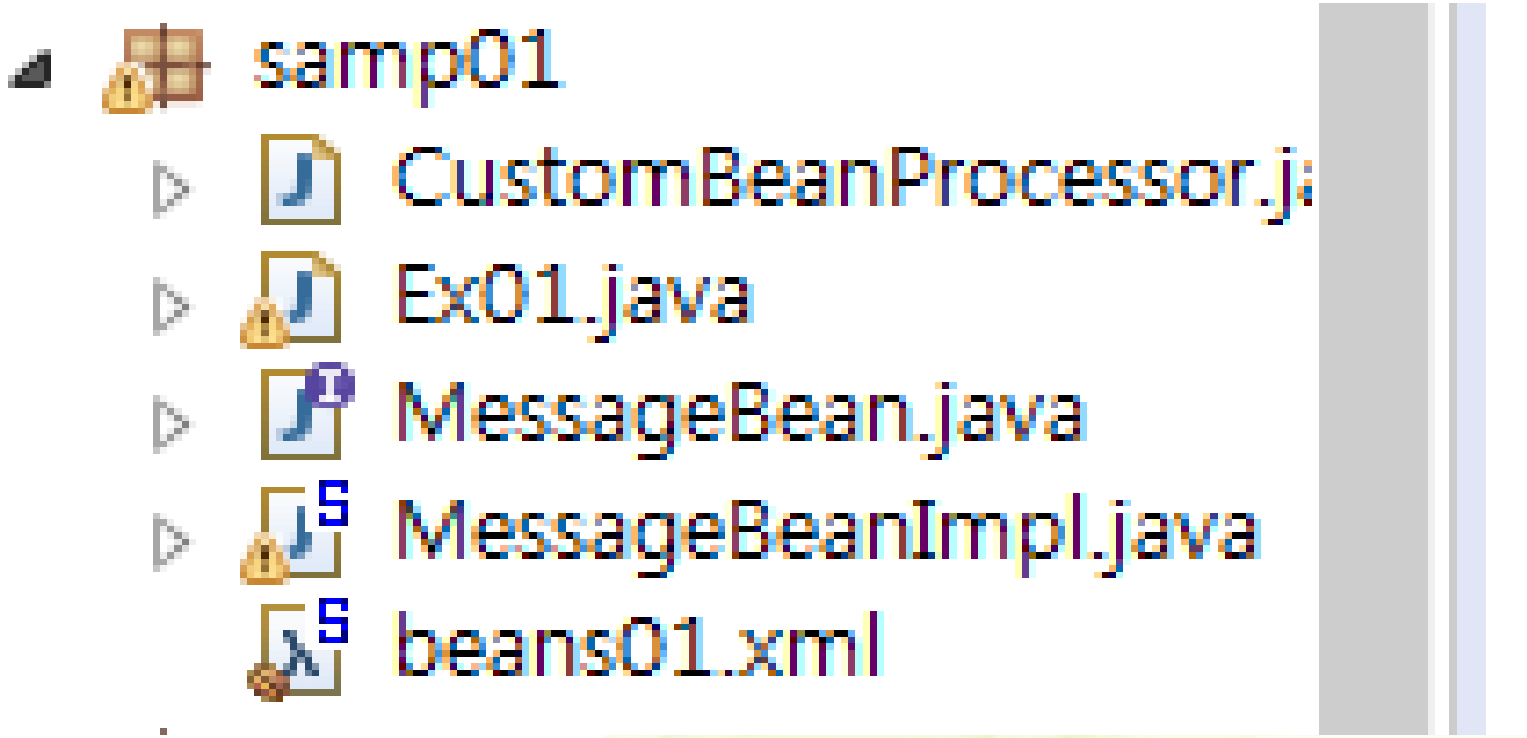
메서드 예: **public void exceptionLog(Exception ex) { … }**

<aop:around>

around 어드바이스는 핵심코드가 실행되는 동안 공통코드가 실행

구조

POJO라는 **aop**스키마 활용
로그 출력처리가 있는 **LoggingSample**클래스는 인터페이스를 구현하지 않음
설정파일 **beans.xml**을 사용해서 **LoggingSample**를 **Advice**로 정의하고,
MessageBeanImpl클래스의 **sayHello()**메세드 전후처리에 로그출력 삽입



AOP

aspectJ 표현식

- **aspectJ**에서 지원하는 패턴 표현식
- 명시자
  1) execution : 실행시킬 메소드 패턴을 직접 입력하는 경우
  2) within : 메소드가 아닌 특정 타입에 속하는 메소드들을 설정할 경우
  3) bean : 2.5버전에 추가됨. 설정파일에 지정된 빈의 이름**(name** 속성**)**을 이용해

pointcut 설정

- 표현
  명시자**(**수식어패턴**?** 리턴타입패턴 패키지패턴**?**
  클래스이름패턴**.**메소드이름패턴**(**파라미터패턴**))** **- ?**는 생략가능
  예**) excution(public * abc.def..*Service.set*(..))**
- 수식어 패턴에는 **public, protected** 또는 생략한다.
  1) * : 1개이상 모든 값을 표현
  · **argument**에서 쓰인 경우 **: 1**개 이상의 **argument**
  · **package**에 쓰인 경우 **: 1**개 이상의 하위 **package**
  2) .. : 0개 이상
  · **argument**에서 쓰인 경우 **: 0**개 이상의 **argument**
  · **package**에 쓰인 경우 **: 0**개 이상의 하위 **package**
  ※ 위 예 설명
  적용 하려는 메소드들의 패턴은 **public** 제한자를 가지며 리턴 타입에는 모든 타입이 다 올 수 있다. 이름은 **abc.def** 패키지와 그 하위 패키지에 있는 모든 클래스 중 **Service**로 끝나는 클래스들에서 **set**으로 시작하는 메소드이며 **argument**는 **0**개 이상오며 타입은 상관 없다

XML 스키마 기반의 AOP

❖XML 기반의 POJO 클래스를 이용하는 경우의 클래스

- ◆ POJO 클래스를 Advice 클래스로 사용하고자 하는 경우에는 아래와 같은 원형의 메서드를 선언합니다.

public Object 메서드이름 (ProceedingJoinPoint joinPoint)

- ◆ 위 메서드를 생성한 후 매개변수인 joinPoint 객체가 proceed 메서드를 호출하면 원본 객체의 메서드가 수행
- ◆ proceed 메서드를 호출하기 전에 적용된 코드는 메서드 호출 전에 수행되고 후에 적용된 코드는 메서드 호출 후에 수행됩니다.

XML 스키마 기반의 AOP

Beans.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"
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" />
<aop:config>
   <aop:aspect id="logAspece" ref="loggingSample">
   <aop:pointcut expression="execution(* sayHello())" id="logPointCut"/>
   <aop:around pointcut-ref="logPointCut" method="logAround"/>
   </aop:aspect>
   </aop:config>
<bean id="targetBean" class="sample1.MessageBeanImpl">
      <property name="name">
         <value>Spring</value>
      </property>
   </bean>
</beans>
```

XML 스키마 기반의 AOP

```
package sample1;
import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;
public class LoggingSample {

  public Object logAround(ProceedingJoinPoint pjp) throws Throwable {
       String methodName = pjp.getKind();
      // pjp. getScriptPart().toString();
       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;
    }
}
```

XML 스키마 기반의 AOP

LoggingSample클래스 : 어떤 인터페이스도 구현하지 않는 **POJO**

logAround()메서드는 **Advice**의 처리로서 실행하는 메서드, **Around Advice**로서 동작 시킬 메서드는 첫 번째 파라미터가 **proceedingJoinPoint**타입 이어야 함.
Advice로 실행할 메서드 중 **ProceedingJoinPoint** 클래스의 **proceed()** 메서드를 호출.
proceed() 메서드는 **AOP**에서 대상 객체의 메서드 호출을 시작하는 메서드 임.
대상객체의 메서드가 인수를 갖는다면, 필요한 값을 **Object**배열로 **proceed()** 메서드에 넘김.
proeed()메서는 반드시 한번 만 호출함

XML 스키마 기반의 AOP

```
package sample1;

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

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

      bean.sayHello();
   }
}
```

AspectJ를 이용해서 **AOP**를 실행하고 있기 때문에 프록시가 아닌 **Bean** 그대로를 취득.
취득한 **Bean**에는 **Advice**가 위빙되어 있기 때문에 **sayHello()**를 호출하면 메서드 추적 정보가 출력 됨

XML 스키마 기반의 AOP

```
package sample1;
public interface MessageBean {
    void sayHello();
}

package sample1;
public class MessageBeanImpl implements MessageBean {
    private String name;
    public void setName(String name) {
        this.name = name;
    }
    public void sayHello() {
        try {
            Thread.sleep(5000);
        } catch(InterruptedException e) {}

        System.out.println("Hello, " + name + "!");
    }
}
```

어노테이션에 의한 **autowiring**

@Autowired

```
@Autowired
public void setOutputter(Outputter outputter) {
    this.outputter =outter;
}
```

@Autowired 어노테이션을 사용하면 **Autowiring**은 **byType** 즉 **Bean**의 **Type**을 사용해서 연결하는 방법으로 실행됩니다.

설정파일 **beans.xml**에 **<context:annotation-config>**효소를 추가합니다.

pom.xml에 추가

```
<dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.9.6</version>
        <scope>runtime</scope>
</dependency>
        <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjrt</artifactId>
        <version>1.8.2</version>
</dependency>
```

어노테이션에 의한 **autowiring**

aspectJ는 제록스 팔로알토 연구소에서 개발한 **AOP**구현
http://www.eclipse.org/aspectj/
- **aspectJ**에 정의된 포인트 컷
  execution Bean의 조건에 맞는 메서드나 생성자의 실행을 **Pointcut**으로 함

execution(public * set(..))
  public메서드이면서 **set**으로 시작하는 모든 메서드를 대상

execution(* sample1.*.*(..))
  sample1 패키지에 있는 모든 클래스의 모든 메서드를 대상

execution(* sample1..*.*(..))
  sample1 패키지와 그 하위 패키지에 있는, 모든 클래스의 모든 메서드를 대상

AspectJ에서 사용할 수 있는 논리 연산자
- **&& and**
- **|| or**
- **! not**

aop 어노테이션

@Aspect
- 공통 코드를 정의한 클래스에 사용
- 공통코드 객체가 빈으로 등록되어야 하기 때문에 @Aspect 아노테이션과 @Component 아노테이션과 같이 사용
- 예

```
@Component
@Aspect
public class LogAspect {
// 생략...
}
```

@Pointcut
- 포인트컷을 지정할 때 사용
- Aspect 클래스 내에 아무 기능도 구현하지 않은 메서드를 추가하고 그 위에 @Pointcut 아노테이션을 이용하여 포인트컷 표현식을 작성
- 예

```
@Pointcut(value="execution(*                    com.coderby..*.sayHello(..))")
private void helloPointcut() {}
@Pointcut(value="execution(*                com.coderby..*.sayGoodbye(..))")
private void goodbyePointcut() {}
```

어드바이스

@Before
핵심코드가 실행되기 전에 실행할 메서드에 정의
@Before("helloPointcut()")
@After
핵심코드가 실행된 후 실행
메서드가 정상 종료될 때 뿐 아니라 예외가 발생해도 실행
@After("helloPointcut()")
@AfterReturning
핵심코드가 실행된 후 실행
메서드가 정상 실행 한 후에만 실행
@AfterReturning(pointcut="helloPointcut()", returning="message")
@AfterThrowing
예외가 발생할 경우 실행
@AfterThrowing(pointcut="goodbyePointcut()", throwing="exception")
@Around
@Around("execution(* com.coderby..*.*(..))")

어노테이션으로 AOP정의하기

```
<?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"
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/>
 <bean id="loggingSample" class="sample2.LoggingSample" />
 <bean id="targetBean" class="sample2.MessageBeanImpl">
   <property name="name">
    <value>Spring</value>
   </property>
   </bean>

</beans>
```

```
package sample2;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.util.StopWatch;
@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 calling.");
        Object rtnObj = pjp.proceed();

        sw.stop();

        System.out.println("[LOG] METHOD: " + methodName + " was called.");
        System.out.println("[LOG] 처리시간 " + sw.getTotalTimeSeconds() + "초");

        return rtnObj;
    }
}
```

XML 스키마 기반의 AOP

- JoinPoint 인터페이스: 호출되는 대상 객체, 메소드, 그리고 전달되는 파라미터 목록에 접근할 수 있는 메소드를 제공하는 인터페이스
  - Signature getSignature(): 호출되는 메소드에 대한 정보 제공
  - Object getTarget(): 대상이 되는 객체
  - Object[] getArgs(): 파라미터 목록
- Signature 인터페이스: 호출되는 메소드에 대한 정보를 제공하는 인터페이스
  - String getName(): 메소드의 이름 리턴
  - String toLoginString(): 메소드를 완전하게 표현한 문장 리턴
  - String toShortString(): 메소드를 축약해서 표현한 문장 리턴

XML 스키마 기반의 AOP

❖XML 기반의 POJO 클래스를 이용하는 경우의 bean 설정

```
<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">
        <!-- Advice 클래스를 빈으로 등록 -->
        <bean id= "아이디" class= " 앞에서 생성한 POJO 클래스" />
        <!-- Aspect 설정: Advice를 어떤 Pointcut에 적용할 지 설정 -->
        <aop:config>
            <aop:aspect id= "자신의 아이디" ref= " 위의 아이디">
              <aop:pointcut id= " 포인트 컷 아이디" expression= " 적용될 대상" />
              <aop:around pointcut-ref= " 자신의 아이디"
                  method= "AOP의 대상이 되는 메서드" />
            </aop:aspect>
        </aop:config>
```

XML 스키마 기반의 AOP

1. Simple Spring Maven 프로젝트 생성
2. pom.xml 파일에 내용 추가(spring aspectj를 사용하기 위해서)

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>${spring-framework.version}</version>
</dependency>
```

프록시를 이용한 AOP

```
package sample6.di.business.domain;
public class Product {
    private String name;
    private int price;
    public Product(String name, int price) {
        this.name = name;
        this.price = price;
    }
    public String getName() {
        return name;
    }
    public int getPrice() {
        return price;
    }
    @Override
    public String toString() {
        return "Product [name=" + name + ", price=" + price + "]";
    }
}
```

```
package sample6.di.business.service;
import sample6.di.business.domain.Product;
public interface ProductDao {
    Product getProduct(String name);
}


package sample6.di.dataaccess;
import org.springframework.stereotype.Component;
import sample6.di.business.domain.Product;
import sample6.di.business.service.ProductDao;
@Component
public class ProductDaoImpl implements ProductDao {
    // Dao이지만 간단히 하고자 RDB에는 액세스하지 않는다.
    public Product getProduct(String name) {
        // Dao답게 제품명과 가격을 가진 Product를 검색한 것처럼 반환한다.
        return new Product(name, 100);
    }
}
```

```
package sample6.di.business.service;
import sample6.di.business.domain.Product;
public interface ProductService {
    Product getProduct();
}
```

```
package sample6.di.business.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import sample6.di.business.domain.Product;
@Component("productService")
public class ProductServiceImpl implements ProductService {
    @Autowired
    private ProductDao productDao;
    public Product getProduct() {
        // 그럴 듯하게 검색 조건을 넣고 있다
        return productDao.getProduct("호치키스");
    }
}
```

```
package sample6.aop;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
import sample6.di.business.domain.Product;
public class MyFirstAspect {
	public void before(JoinPoint jp) {
	 // 메소드 시작 시에 Weaving하는 Advice
		System.out.println("Hello Before! *** 메소드가 호출되기 전에 나온다!");
		Signature sig = jp.getSignature();
		System.out.println("-----> 메소드 이름을 취득한다:" + sig.getName());
		Object[] o = jp.getArgs();
		System.out.println("-----> 가인수 값을 취득한다:" + o[0]);
	}
	public void after() {
	 // 메소드 종료 후에 Weaving하는 Advice
		System.out.println("Hello After! *** 메소드가 호출된 후에 나온다!");
	}
	public void afterReturning(JoinPoint jp, Product product) {
	 // 메소드 호출이 예외를 내보내지 않고 끝났을 때 호출되는 Advice
		System.out.println("Hello AfterReturning! *** 메소드 호출 후에 나온다");
		// System.out.println("-----> return value = " + ret);
		Signature sig = jp.getSignature();
		System.out.println("-----> 메소드 이름을 취득한다:" + sig.getName());
		Object[] o = jp.getArgs();
		System.out.println("-----> 가인수 값을 취득한다:" + o[0]);
	}
```

```
    public Product around(ProceedingJoinPoint pjp) throws Throwable {
        //메소드 호출 전후에 Weaving하는 Advice
        System.out.println("Hello Around! before *** 메소드 호출하기 전에 나온다!");
        // Signature sig = pjp.getSignature();
        // System.out.println("--> aop:around 메소드 이름을 취득한다:" +
        //  sig.getName());
        Product p = (Product) pjp.proceed();
        // msg = msg + ":결과에 멋대로 추가해버린 hoge!";
        System.out.println("Hello Around! after *** 메소드를 호출한 후에 나온다!");
        return p;
    }

    public void afterThrowing(Throwable ex) {
        // 메소드 호출이 예외를 내보냈을 때 호출되는 Advice
        System.out.println("Hello Throwing! *** 예외가 생기면 나온다");
        System.out.println("exception value = " + ex.toString());
    }
}
```

```
<?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:context="http://www.springframework.org/schema/context"
 xmlns:aop="http://www.springframework.org/schema/aop"
 xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.1.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-3.1.xsd">
 <context:component-scan base-package="sample6" />
 <aop:config>
  <aop:aspect id="myAspect" ref="myFirstAspect">
   <aop:pointcut id="pc" expression="execution(* getProduct(String))" />
   <aop:before pointcut-ref="pc" method="before" />
   <aop:after pointcut-ref="pc" method="after" />
   <aop:after-returning pointcut-ref="pc" method="afterReturning"
returning="product"/>
   <aop:around pointcut-ref="pc" method="around" />
   <aop:after-throwing pointcut-ref="pc" method="afterThrowing"
throwing="ex"/>
  </aop:aspect>
 </aop:config>
 <bean id="myFirstAspect" class="sample6.aop.MyFirstAspect" />
</beans>
```

```
package sample6;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import sample6.di.business.domain.Product;
import sample6.di.business.service.ProductService;

public class ProductSampleRun {

   public static void main(String[] args) {
      ProductSampleRun productSampleRun = new ProductSampleRun();
      productSampleRun.execute();
   }

   public void execute() {
      ApplicationContext ctx = new ClassPathXmlApplicationContext(
            "/sample6/config/applicationContext.xml");
      ProductService productService = (ProductService) ctx
            .getBean("productService");
      Product product = productService.getProduct();
      System.out.println(product);
   }
}
```

@Aspect 어노테이션 기반의 AOP

❖@Aspect 어노테이션은 자바 코드에 AOP를 설정하는 방식입니다.

❖@Aspect: AOP 기능을 수행할 클래스의 상단에 기재해서 이 클래스가 AOP 클래스라는 것을 설정

❖@Around(“표현식”) 메소드 상단에 기재해서 표현식에 해당하는 메소드가 호출될 때 메소드를 호출해서 수행

❖위에 설정된 클래스를 AOP 클래스로 등록하기 위해서는 설정 파일에 <aop:aspectj-autoproxy />를 추가

@Aspect 어노테이션 기반의 AOP

❖LoggingAdvice 클래스 수정

```
package aop;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class LoggingAdvice {
    @Around("execution(public * message..*(..))")
    public Object invoke(ProceedingJoinPoint joinPoint) throws Throwable {
        String methodName = joinPoint.getSignature().toLongString();
        System.out.println( methodName +  " is calling.");
        long start = System.currentTimeMillis();
        joinPoint.proceed();
```

Annotation Product 예제

```
package sample7.di.business.domain;
public class Product {
    private String name;
    private int price;
    public Product(String name, int price) {
        this.name = name;
        this.price = price;
    }
    public String getName() {
        return name;
    }
    public int getPrice() {
        return price;
    }
    @Override
    public String toString() {
        return "Product [name=" + name + ", price=" + price + "]";
    }
}
```

```
package sample7.di.business.service;
import sample7.di.business.domain.Product;
public interface ProductDao {
   Product getProduct(String name);
}
```

```
package sample7.di.dataaccess;
import org.springframework.stereotype.Component;
import sample7.di.business.domain.Product;
import sample7.di.business.service.ProductDao;
@Component
public class ProductDaoImpl implements ProductDao {
    // Dao이지만 간단히 하고자 RDB에는 액세스 하지 않는다.
    public Product getProduct(String name) {
        // Dao답게 제품명과 가격을 가진 Product를 검색한 것처럼 반환한다.
        return new Product(name, 100);
   }
}
```

```
package sample7.di.business.service;
import sample7.di.business.domain.Product;
public interface ProductService {
   Product getProduct();
}


package sample7.di.business.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import sample7.di.business.domain.Product;
@Component("productService")
public class ProductServiceImpl implements ProductService {
    @Autowired
    private ProductDao productDao;
    public Product getProduct() {
        // 그럴듯하게 검색 조건을 집어넣는다
        return productDao.getProduct("호치키스");
    }
}
```

```
package sample7.aop;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;import
org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;
import sample7.di.business.domain.Product;
@Aspect
@Component
public class MyFirstAspect {
    @Before("execution(* getProduct(String))")
    public void before(JoinPoint jp) {
        // 메소드 시작 시에 Weaving하는 Advice
        System.out.println("Hello Before! *** 메소드가 호출되기 전에 나온다!");
        Signature sig = jp.getSignature();
        System.out.println("----->메소드 이름을 취득한다:" + sig.getName());
        Object[] o = jp.getArgs();
         System.out.println("-----> 가인수 값을 취득한다:" + o[0]);
    }
```

```
@After("execution(* getProduct(String))")
public void after() {     // 메소드 종료 후에 Weaving하는 Advice
    System.out.println("Hello After! *** 메소드가 호출된 후에 나온다!");
}
@AfterReturning(value = "execution(* getProduct(String))",
            returning = "product")
public void afterReturning(JoinPoint jp, Product product) {
    // 메소드 호출이 예외 송출 없이 종료했을 때 호출되는 Advice
    System.out.println("Hello AfterReturning! *** 메소드 호출한 후에 나온다! ");
    // System.out.println("-----> return value = " + ret);
    Signature sig = jp.getSignature();
    System.out.println("-----> 메소드 이름을 취득한다:" + sig.getName());
    Object[] o = jp.getArgs();
    System.out.println("----->가인수 값을 취득한다:" + o[0]);
}
@Around("execution(* getProduct(String))")
public Product around(ProceedingJoinPoint pjp) throws Throwable {
     // 메소드 호출 전후에 Weaving하는 Advice
     System.out.println("Hello Around! before ** 메소드 호출하기 전 나온다!");
     // Signature sig = pjp.getSignature();
     // System.out.println("-----> aop:around 메소드 이름을 취득한다:" +
     // sig.getName());
     Product p = (Product) pjp.proceed();
     // msg = msg + ": 결과에 멋대로 추가해버린 hoge!";
    System.out.println("Hello Around! after *** 메소드를 호출한 후에 나온다!");
    return p;
}
```

```
@AfterThrowing(value = "execution(* getProduct(String))",
               throwing = "ex")
    public void afterThrowing(Throwable ex) {
        // 메소드 호출이 예외를 내보냈을 때 호출되는 Advice
        System.out.println("Hello Throwing! *** 예외가 생기면 나온다");
        System.out.println("exception value = " + ex.toString());
    }
}
```

```
<?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:context="http://www.springframework.org/schema/context"
 xmlns:aop="http://www.springframework.org/schema/aop"
 xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.1.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.1.xsd">
 <context:annotation-config />
 <context:component-scan base-package="sample7" />
 <aop:aspectj-autoproxy />
</beans>
```

```
package sample7;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import sample7.di.business.domain.Product;
import sample7.di.business.service.ProductService;

public class ProductSampleRun {

   public static void main(String[] args) {
      ProductSampleRun productSampleRun = new ProductSampleRun();
      productSampleRun.execute();
   }

   public void execute() {
      ApplicationContext ctx = new ClassPathXmlApplicationContext(
            "/sample7/config/applicationContext.xml");
      ProductService productService = (ProductService) ctx
            .getBean("productService");
      Product product = productService.getProduct();
      System.out.println(product);
   }
}
```