Spring Framework

AOP

Non-AOP

```java
public interface Person {
    public void doSomething();
}
```

```java
public class Boy implements Person{
    @Override
    public void doSomething() {
        System.out.println("문을 열고 집에 들어간다.");
        try {
            System.out.println("컴퓨터로 게임을 하다.");
            System.out.println("잠을 잔다.");
        } catch (Exception e) {
            System.out.println("화재발생 : 119에 신고한다.");
        }finally{
            System.out.println("문열고 집을 나온다.");
        }

    }
}
```

```java
public class Girl implements Person{
    @Override
    public void doSomething() {
        System.out.println("문을 열고 집에 들어간다.");
        try {
            System.out.println("드라마을 본다.");
            System.out.println("잠을 잔다.");
        } catch (Exception e) {
            System.out.println("화재발생 : 119에 신고한다.");
        }finally{
            System.out.println("문열고 집을 나온다.");
        }
    }
}
```

Non-AOP

- 프로그래밍을 하게 되면 메인 프로그램 외에 여러 공통 사항을 계속 반복해야 하는 상황이 발생한다.
- 위의 코드에서 핵심 사항을 제외한 공통사항이 각 메서드마다 반복되는 것을 볼 수 있다.
- 위와 같은 코드는 생산성을 떨어뜨리며 상당히 비효율적인 코드라고 볼 수 있다.

Proxy Pattern

- Proxy는 우리말로 대리자, 대변인 이라는 뜻.
- 구체적으로 인터페이스를 사용하고 실행시킬 클래스에 대한 객체가 들어갈 자리에 대리자 객체를 대신 투입해 클라이언트 쪽에서 실제 실행시킬 클래스에 대한 객체를 통해 메서드를 호출하고 반환 값을 받는지, 대리자 객체를 통해 메서드를 호출하고 반환 값을 받는지 전혀 모르게 처리하는 것을 말한다.
- 중요한 것은 흐름제어만 할 뿐 결과값을 조작하거나 변경시키면 안된다. 프록시는 자신의 의견을 반영하는 것을 목적으로 하지 않고 제어의 흐름을 변경하거나 다른 로직을 수행하기 위해 사용한다.

Proxy Pattern

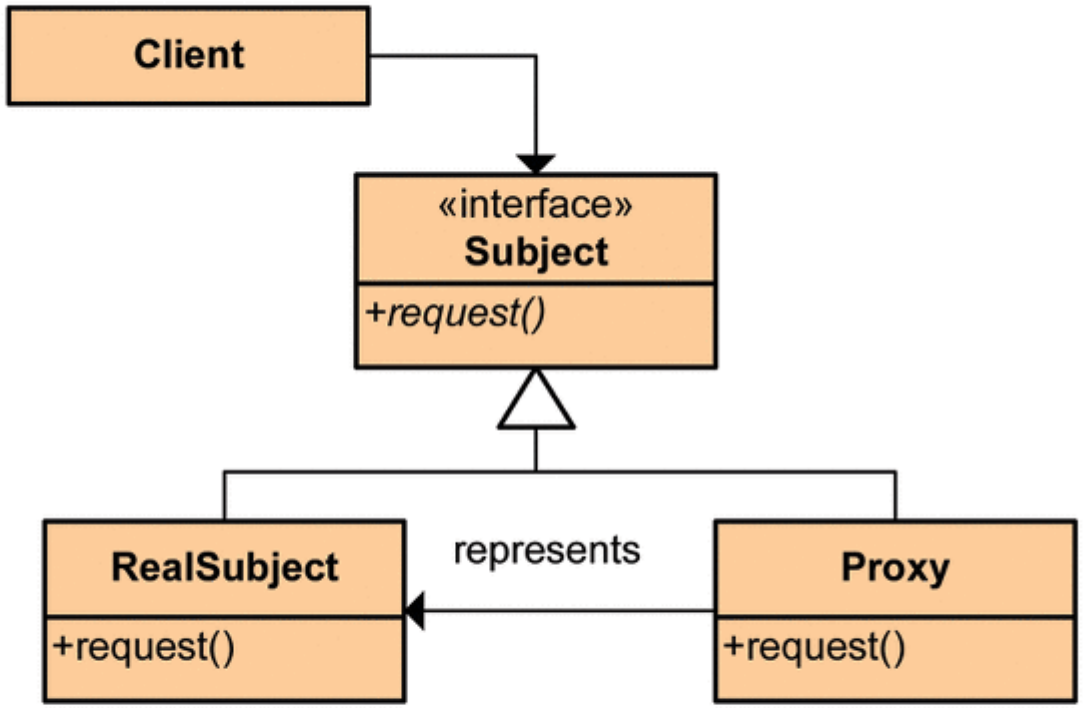
- Proxy 패턴의 특징
  - 대리자는 실제 서비스와 같은 이름의 메서드를 구현한다. 이때 인터페이스를 사용한다.
  - 대리자는 실제 서비스에 대한 참조 변수를 갖는다(합성)
  - 대리자는 실제 서비스의 같은 이름을 가진 메서드를 호출하고 그 값을 클라이언트에게 돌려준다.
  - 대리자는 실제 서비스의 메서드 호출 전후에도 별도의 로직을 수행할 수도 있다.

Proxy Pattern

- Proxy 패턴의 특징

Proxy

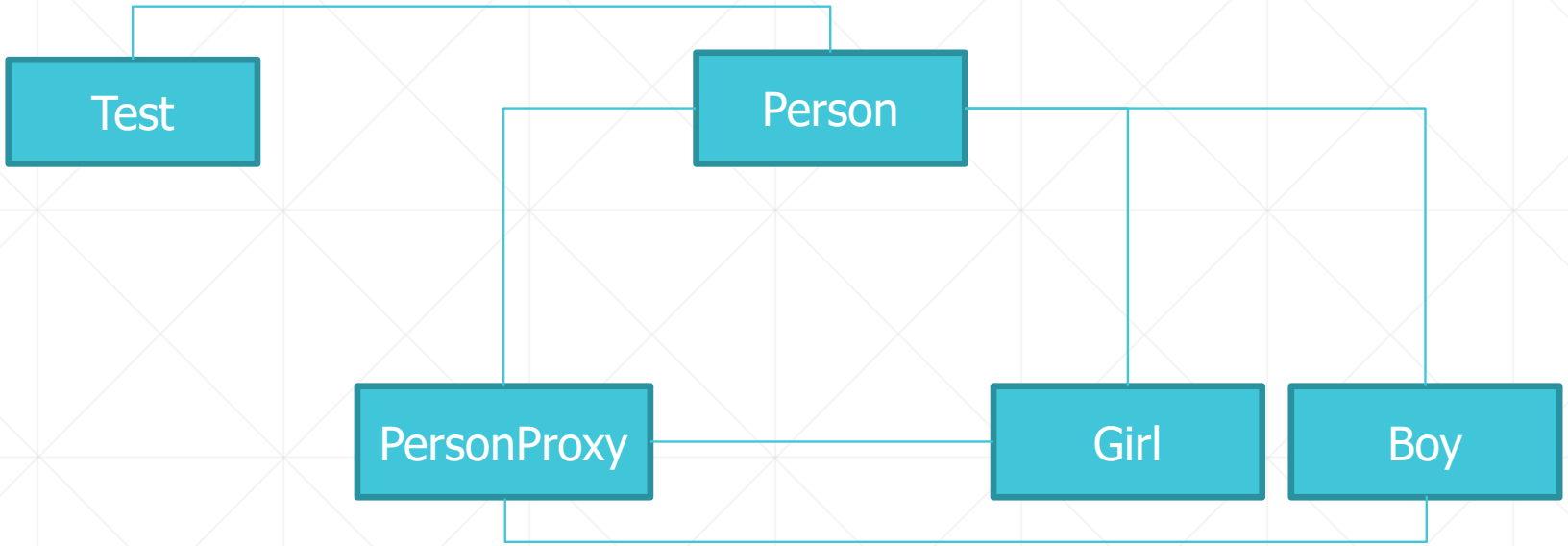
Type: Structural



What it is:
Provide a surrogate or placeholder for another object to control access to it.

Proxy Pattern

- Proxy 패턴 예제1



Proxy Pattern

- Proxy 패턴 예제1

```java
public interface Person {
    public void doSomething();
}

public class Boy implements Person{
    @Override
    public void doSomething() {
        System.out.println("컴퓨터로 게임을 하다.");
    }
}

public class Girl implements Person{
    @Override
    public void doSomething() {
        System.out.println("드라마를 본다.");
    }
}
```

Pr

- Proxy 패턴 예제1

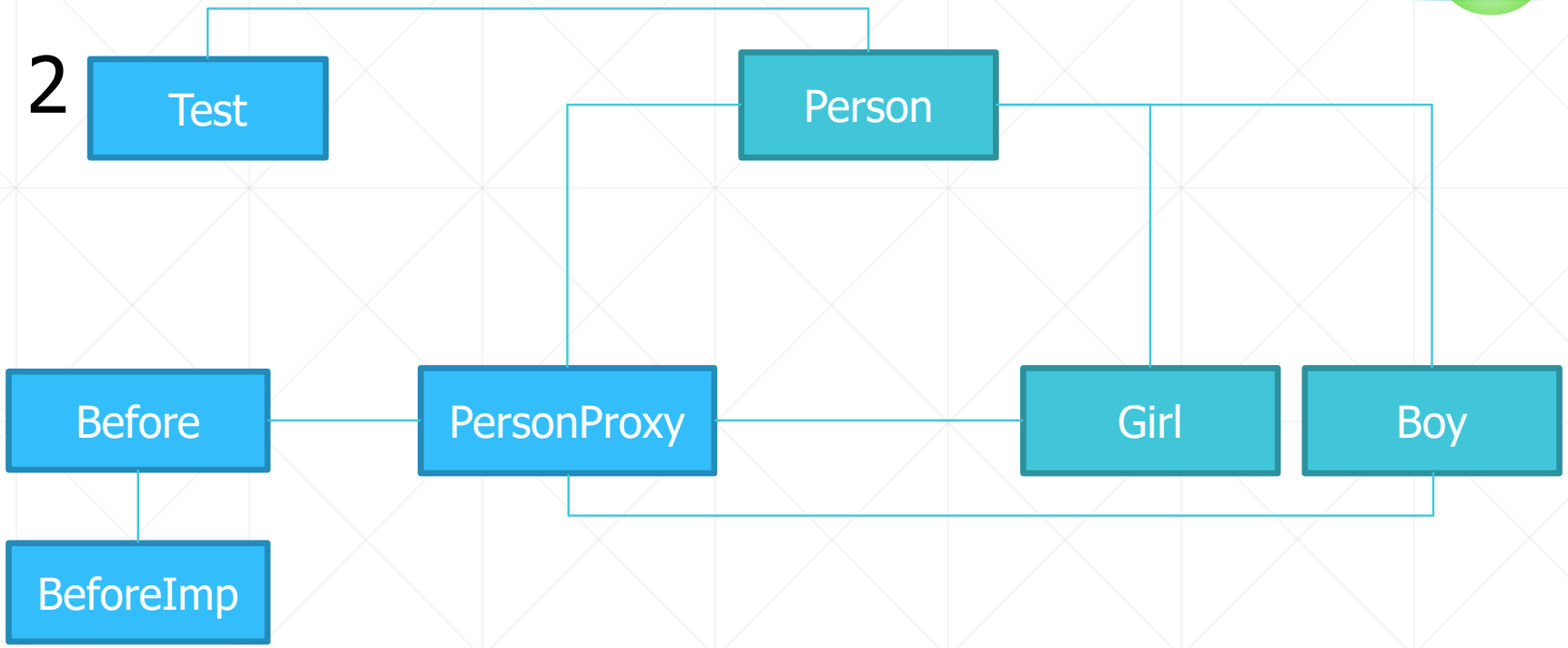
```
public class PersonProxy implements Person{

    private Person person;
    public Person getPerson() { return person;  }
    public void setPerson(Person person) {  this.person = person;    }

    @Override
    public void doSomething() {
       System.out.println("문을 열고 집에 들어간다.");
       try {
           person.doSomething();
           System.out.println("잠을 잔다.");
       } catch (Exception e) {
            System.out.println("화재발생:119에 신고한다.");
       }finally{
        System.out.println("문을 열고 집을 나온다.");
       }
    }

}
public static void main(String[] args) {
     Person boy = new Boy();
     Person girl = new Girl();
     PersonProxy proxy = new PersonProxy();

     // proxy.setPerson(boy);
     proxy.setPerson(girl);
     proxy.doSomething();
}
```

Proxy Pattern

- Proxy 패턴 1의 문제
  - 만약 각각의 공통 기능이 상당히 커질 경우 하나의 메서드 안에서 처리하는 것은 힘들다
  - 결국 공통기능을 밖으로 빼서 하나의 메서드 혹은 클래스로 만들어 처리하는 것이 가장 이상적이다.

Proxy Pattern

- Proxy 패턴 2



Proxy Pattern

- Proxy 패턴 2

```java
public interface Before {
    public void before();
}

public class BeforeImp implements Before{

    @Override
    public void before() {
        System.out.println("문을 열고 집에 들어 들어간다");
    }

}
```

Proxy Pattern

- Proxy 패턴 2

```
public class PersonProxy implements Person{

    private Person person;
    private Before before;

    public Person getPerson() { return person;  }
    public void setPerson(Person person) {  this.person = person;    }
    public Before getBefore() { return before;  }
    public void setBefore(Before before) {  this.before = before;    }

    @Override
    public void doSomething() {
       //System.out.println("문열고 집에 들어간다.");
        if(before != null){
            before.before();
        }
        try {
            person.doSomething();
        //  System.out.println("잠을 잔다.");
        } catch (Exception e) {
        //  System.out.println("화재발생 : 119에 신고한다.");
        }finally{
        //  System.out.println("문열고 나온다.");
        }
    }
}
```

Proxy Pattern

- Proxy 패턴 2

```
public class Test {
    public static void main(String[] args) {
        Person boy = new Boy();
        Person girl = new Girl();
        PersonProxy proxy = new PersonProxy();

    // proxy에 핵심관심사항을 실행전에 before를 실행하도록 틀을 만들어 놓음
        proxy.setBefore(new BeforeImp());

    //  proxy.setPerson(boy);
        proxy.setPerson(girl);
        proxy.doSomething();
    }
}
```

Proxy Pattern

- 위와 같은 방식으로 하나의 공통 관심 사항 당 로직을 만들게 되면 하나의 공통 관심 사항 당 클래스를 하나씩 만들고 그걸 넣어야 하기 때문에 상당히 번거로워 진다.

AOP(Aspect Of Programing)

- AOP : 관점 지향 프로그래밍
- 애플리케이션 전체에 걸쳐 사용되는 기능을 재사용하도록 지원하는 것
- 풀어보면 위에서 이야기 했던 공통 관심 사항들을 AOP를 통해 지원하는 것을 의미한다.
- 핵심 기능에 공통 기능을 삽입하기 위한 방법에는 크게 아래와 같은 3가지 방법이 있다.
  - 컴파일 시점에 코드에 공통 기능을 추가하는 방법
    - AOP 개발 도구가 소스 코드를 컴파일하기 전에 공통 구현 코드를 소스에 삽입하는 방식
  - 클래스 로딩 시점에 바이트 코드에 공통 기능을 추가하는 방법
    - 클래스를 로딩할 때 바이트 코드에 공통 기능을 클래스에 삽입하는 방식
  - 런타임에 프록시 객체를 생성해서 공통 기능을 추가하는 방법
    - 일반적으로 사용하고 있는 AOP 방식

AOP(Aspect Of Programing)

AOP 주요 용어

용어	의미
JoinPoint	Advice를 적용 가능한 지점을 의미한다. 메서드 호출, 필드 값 변경 등이 Joinpoint에 해당한다. 스프링은 프록시를 이용해서 AOP를 구현하기 때문에 메서드 호출에 대한 Joinpoint만 지원한다.
PointCut	Joinpoint의 부분 집합으로서 실제로 Adivce가 적용되는 Joinpoint를 나타낸다. 스프링에서는 정규 표현식이나 AspectJ의 문법을 이용하여 Pointcut을 정의할 수 있다.
Adivce	언제 공통 관심 기능을 핵심 로직에 적용할 지를 정의하고 있다. 예를 들어, '메서드를 호출하기 전'(언제)에 '트랜잭션 시작' (공통기능) 기능을 적용한다는 것을 정의하고 있다.
Weaving	Advice를 핵심 로직 코드에 적용하는 것을 말한다.
Aspect	여러 객체에 공통으로 적용되는 기능을 Aspect라고 한다. 트랜잭션이나 보안 등이 Aspect의 좋은 예이다

AOP(Aspect Of Programing)

- Advice의 종류
  - 스프링 프록시를 이용해서 메서드 호출 시점에 Aspect를 적용하기 때문에 구현 가능한 Adivce의 종류는 아래와 같다.

종류	설명
Before Advice	대상 객체의 메서드 호출 전에 공통 기능을 실행한다.
After Returning Advice	대상 객체의 메서드가 Exception 없이 실행된 이후에 공통 기능을 실행한다.
After Throwing Advice	대상 객체의 메서드를 실행하는 도중 Exception 이 발생한 경우에 공통 기능을 실행한다.
After Advice	대상 객체의 메서드를 실행하는 도중에 Exception 이 발생했는지의 여부에 상관없이 메서드를 실행 후 공통 기능을 실행한다.(finally 블록과 비슷하다)
Around Advice	대상 객체의 메서드 실행 전, 후 또는 Exception 발생 시점에 공통 기능을 실행하는데 사용된다.

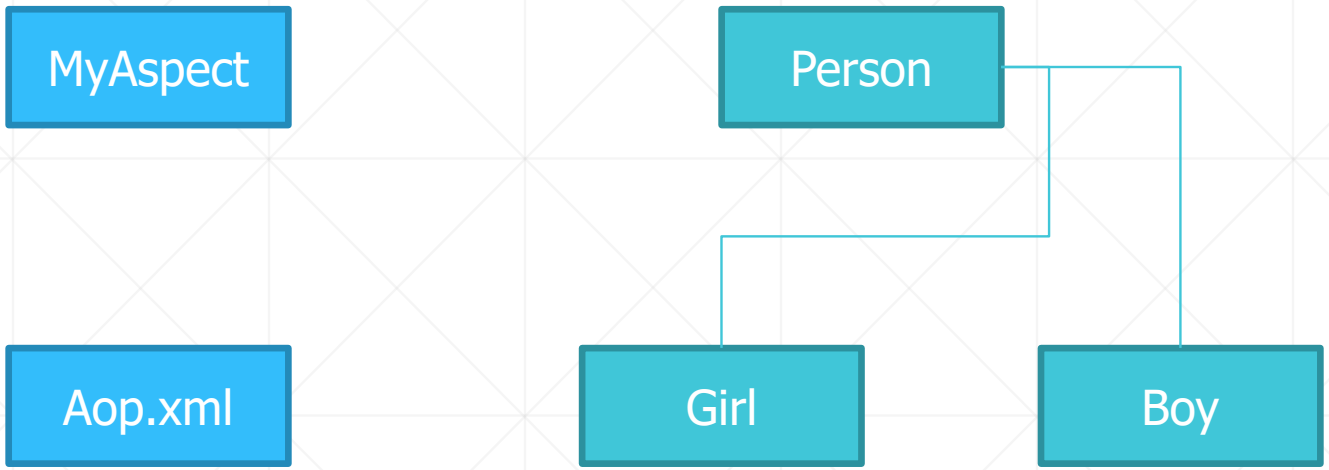
AOP(Aspect Of Programing)

- Advice 예시

```
public class Boy implements Person{
    @Override
    public void doSomething() {
        System.out.println("문을 열고 집에 들어간다."); // 공통관심사항 before
        try {
            System.out.println("컴퓨터로 게임을 하다."); // 핵심관심사항
            System.out.println("잠을 잔다."); // 공통관심사항 after-returning
        } catch (Exception e) {
            System.out.println("화재발생 : 119에 신고한다."); // 공통관심사항 after-throwing
        }finally{
            System.out.println("문열고 집을 나온다."); // 공통관심사항 after
        }

    }
}
```

AOP(Aspect Of Programing)

- AOP (XML 스키마 기반)



AOP(Aspect Of Programing)

- Pom.xml에 다음 라이브러리를 추가한다.

```
<dependency>
<groupId>org.aspectj</groupId>
<artifactId>aspectjweaver</artifactId>
<version>${org.aspectj-version}</version>
</dependency>
```

AOP(Aspect Of Programing)

- AOP (XML 스키마 기반)
  - MyAspect.java

```
public class MyAspect {
    private double time;
    public void before(){
        time = System.currentTimeMillis();
        System.out.println("문을 열고 집에 들에 들어간다.");
    }
    public void af(){
        time = System.currentTimeMillis()-time;
        System.out.println("문을 열고 집에 나온다. ");
        System.out.println("걸린 시간 : " + time + "ms");
    }
    public void after_returning(){
        System.out.println("잠을잔다.");
    }
    public void after_throwing(){
        System.out.println("화재발생:119에 신고한다.");
    }
}
```

AOP(Aspect Of Programing)

- AOP (XML 스키마 기반)
  - aop.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/aop http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="boy" class="ver1.Boy" />
    <bean id="girl" class="ver1.Girl" />
    <bean id="myAspect" class="ver1.MyAspect" />
    <aop:config>
        <!-- 공통관심사항 있는 클래스 -->
        <aop:aspect  ref="myAspect">
            <!-- 클래스와 상관없이 doSomething()메소드를 pointcut(접합점)이다.  -->
            <aop:pointcut expression="execution(* doSomething())" id="myPointcnt"/>
            <aop:before method="before" pointcut-ref="myPointcnt" />
            <aop:after method="af" pointcut-ref="myPointcnt"/>
            <aop:after-returning method="after_returning" pointcut-ref="myPointcnt"/>
            <aop:after-throwing method="after_throwing" pointcut-ref="myPointcnt"/>
        </aop:aspect>
    </aop:config>
</beans>
```

AOP(Aspect Of Programing)

- AOP (XML 스키마 기반)
  - Test.java

```
public class Test {
    public static void main(String[] args) {
        ApplicationContext context =
                new GenericXmlApplicationContext("ver1/aop.xml");
        // 꼭 인터페이스타입으로
        Person person = (Person) context.getBean("girl");
        person.doSomething();
    }
}
```

AOP(Aspect Of Programing)

- AOP (@Aspect 어노테이션 버전)
  - 어노테이션을 통한 AOP 클래스 등록이 가능하다.
  - 이럴 경우 Class 부분에 @Aspect라는 어노테이션이 있어야 한다
  - MyAspect.java

```java
@Aspect
@Component
public class MyAspect {
    private double time;

    @Pointcut("execution(* doSomething())")
    public void myPointcut(){}

    @Before("myPointcut()")
    public void before(){
        time = System.currentTimeMillis();
        System.out.println("문을 열고 집에 들에 들어간다.");
    }
    @After("myPointcut()")
    public void af(){
        time = System.currentTimeMillis()-time;
        System.out.println("문을 열고 집에 나온다. ");
        System.out.println("걸린 시간 : " + time + "ms");
    }
    @AfterReturning("myPointcut()")
    public void after_returning(){
        System.out.println("잠을잔다.");
    }
    @AfterThrowing("myPointcut()")
    public void after_throwing(){
        System.out.println("화재발생:119에 신고한다.");
    }
}
```

AOP(Aspect Of Programing)

- AOP (@Aspect 어노테이션 버전)
  - 어노테이션을 읽기 위해 xml에는 context:annotation-config 를 추가한다
  - 또한 aop 객체를 읽기 위해 aspectj-autoproxy 설정의 expose-proxy 설정을 true로 놓는다.
  - aop.xml

```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"
       xsi:schemaLocation="http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
           http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.2.xsd">

    <context:annotation-config />
    <context:component-scan base-package="ver1_a" />
    <aop:aspectj-autoproxy expose-proxy="true" />

    <bean id="boy" class="ver1_a.Boy" />
    <bean id="girl" class="ver1_a.Girl" />
```

AOP(Aspect Of Programing)

- AOP 로 값 전달(xml 스키마 기반)
  - 실제 AOP 쪽으로 핵심 로직에서 값을 전달받고자 할 때 쓰인다.
  - Boy.java/Girl.java

```java
public class Boy {
    public String doSomething() throws Exception{
        if(new Random().nextBoolean()){
            throw new Exception("화재발생");
        }
        System.out.println("컴퓨터로 게임을 하다.");
        return "I am a boy";
    }
}
```

```java
public class Girl {
    public String doSomething() throws Exception{
        if(new Random().nextBoolean()){
            throw new Exception("화재발생");
        }
        System.out.println("드라마를 본다.");
        return "I am a girl";
    }
}
```

AOP(Aspect Of Programing)

- AOP 로 값 전달(xml 스키마 기반)
  - MyAspect.java

```
public class MyAspect {
    private double time;
    public void before(JoinPoint  jp){
        System.out.println(jp.getTarget());
        System.out.println(jp.getThis());
        System.out.println(jp.getSignature());
        time = System.currentTimeMillis();
        System.out.println("문을 열고 집에 들에 들어간다.");
    }
    public void af(JoinPoint jp){
        time = System.currentTimeMillis()-time;
        System.out.println("문을 열고 집에 나온다. ");
        System.out.println("걸린 시간 : " + time + "ms");
    }
    public void after_returning(JoinPoint jp, String str){
        System.out.println(str+"잠을잔다.");
    }
    public void after_throwing(JoinPoint jp, Throwable thr){
        System.out.println(thr+":119에 신고한다.");
    }
}
```

AOP(Aspect Of Programing)

- AOP 로 값 전달(xml 스키마 기반)
  - 반드시 xml 스키마에서 리턴되거나 에러가 나는 데이터들에 대한 명시가 있어야 한다

```
<bean id="boy" class="ver2.Boy" />
<bean id="girl" class="ver2.Girl" />
<bean id="myAspect" class="ver2.MyAspect" />
<aop:config>
     <!-- 공통관심사항 있는 클래스 -->
     <aop:aspect  ref="myAspect">
        <!-- 클래스와 상관없이 doSomething()메소드를 pointcut(접합점)이다.  -->
        <aop:pointcut expression="execution(* doSomething())" id="myPointcnt"/>
        <aop:before method="before" pointcut-ref="myPointcnt" />
        <aop:after method="af" pointcut-ref="myPointcnt"/>
        <aop:after-returning method="after_returning" pointcut-ref="myPointcnt" returning="str"/>
        <aop:after-throwing method="after_throwing" pointcut-ref="myPointcnt" throwing="thr"/>
    </aop:aspect>
</aop:config>
```

AOP(Aspect Of Pro

- AOP 로 값 전달(@Aspect 어노테이션 버전)
  - MyAspect.java

```java
@Aspect
@Component
public class MyAspect {
    private double time;

    @Pointcut("execution(* doSomething())")
    public void myPointcut(){}

    @Before("myPointcut()")
    public void before(JoinPoint  jp){
        System.out.println(jp.getTarget());
        System.out.println(jp.getThis());
        System.out.println(jp.getSignature());
        time = System.currentTimeMillis();
        System.out.println("문을 열고 집에 들에 들어간다.");
    }
    @After("myPointcut()")
    public void af(JoinPoint jp){
        time = System.currentTimeMillis()-time;
        System.out.println("문을 열고 집에 나온다. ");
        System.out.println("걸린 시간 : " + time + "ms");
    }
    @AfterReturning(pointcut="myPointcut()", returning="str")
    public void after_returning(JoinPoint jp, String str){
        System.out.println(str+"잠을잔다.");
    }
    @AfterThrowing(pointcut="myPointcut()", throwing="thr")
    public void after_throwing(JoinPoint jp, Throwable thr){
        System.out.println(thr+":119에 신고한다.");
    }
}
```

AOP(Aspect Of Programing)

- AOP 로 값 전달(@Aspect 어노테이션 버전)
  - aop.xml

```
<context:annotation-config />
<context:component-scan base-package="ver2_a" />
<aop:aspectj-autoproxy proxy-target-class="true"/>
<bean id="boy" class="ver2_a.Boy" />
<bean id="girl" class="ver2_a.Girl" />
```

  - Test.java

```
public static void main(String[] args) {
    ApplicationContext context =
            new GenericXmlApplicationContext("ver2_a/aop.xml");

    Girl girl = (Girl) context.getBean("girl");
    try {
        girl.doSomething();
    } catch (Exception e) {
    }
}
```

AOP(Aspect Of Programing)

- Around Advice
  - Before, After, AfterThrowing, AfterReturning 등 다양한 Advice가 필요하지 않고 기능 하나에 모든 것을 정의하고자 할 경우 쓰인다.
  - Around 메서드는 반드시 ProceedingJoinPoint 인스턴스를 매개변수로 받아 선언하며 해당 인스턴스를 통해 핵심 사항을 실행시킬 수 있다.
  - 핵심 사항 실행 시 proceed()라는 메서드를 이용해 실행시킬 수 있다.

AOP(Aspect Of Programing)

- Around Advice(XML 스키마 기반)
  - MyAspect.java

```java
public class MyAspect {
    private double time;
    public void before(JoinPoint  jp){
        System.out.println(jp.getTarget());
        System.out.println(jp.getThis());
        System.out.println(jp.getSignature());
        time = System.currentTimeMillis();
        System.out.println("문을 열고 집에 들에 들어간다.");
    }
    public void af(JoinPoint jp){
        time = System.currentTimeMillis()-time;
        System.out.println("문을 열고 집에 나온다. ");
        System.out.println("걸린 시간 : " + time + "ms");
    }
    public void after_returning(JoinPoint jp, Object obj){
        System.out.println(obj+"잠을잔다.");
    }
    public void after_throwing(JoinPoint jp, Throwable thr){
        System.out.println(thr+":119에 신고한다.");
    }

    public void around(ProceedingJoinPoint proceedingJoinPoint){
        before(proceedingJoinPoint);
        try {
            after_returning(proceedingJoinPoint, proceedingJoinPoint.proceed());
        } catch (Throwable e) {
            after_throwing(proceedingJoinPoint, e);
        }finally{
            af(proceedingJoinPoint);
        }
    }
```

AOP(Aspect Of Programing)

- Around Advice(XML 스키마 기반)
  - aop.xml

```
<bean id="boy" class="ver3.Boy" />
<bean id="girl" class="ver3.Girl" />
<bean id="myAspect" class="ver3.MyAspect" />
<aop:config>
     <!-- 공통관심사항 있는 클래스 -->
     <aop:aspect  ref="myAspect">
        <!-- 클래스와 상관없이 doSomething()메소드를 pointcut(접합점)이다.  -->
        <aop:pointcut expression="execution(* doSomething())" id="myPointcnt"/>
        <aop:around method="around" pointcut-ref="myPointcnt"/>
        <!-- <aop:before method="before" pointcut-ref="myPointcnt" />
             <aop:after method="af" pointcut-ref="myPointcnt"/>
             <aop:after-returning method="after_returning" pointcut-ref="myPointcnt" returning="str"/>
             <aop:after-throwing method="after_throwing" pointcut-ref="myPointcnt" throwing="thr"/> -->
    </aop:aspect>
</aop:config>
```

AOP(Aspect Of Programing)

Around Advice(@Aspect 어노테이션 버전)

MyAspect.java

```java
@Aspect
@Component
public class MyAspect {
    private double time;

    @Pointcut("execution(* doSomething())")
    public void myPointcut(){}

    public void before(JoinPoint  jp){
        System.out.println(jp.getTarget());
        System.out.println(jp.getThis());
        System.out.println(jp.getSignature());
        time = System.currentTimeMillis();
        System.out.println("문을 열고 집에 들에 들어간다.");
    }
    public void af(JoinPoint jp){
        time = System.currentTimeMillis()-time;
        System.out.println("문을 열고 집에 나온다. ");
        System.out.println("걸린 시간 : " + time + "ms");
    }
    public void after_returning(JoinPoint jp, Object obj){
        System.out.println(obj+"잠을잔다.");
    }
    public void after_throwing(JoinPoint jp, Throwable thr){
        System.out.println(thr+":119에 신고한다.");
    }

    @Around("myPointcut()")
    public void around(ProceedingJoinPoint proceedingJoinPoint){
        before(proceedingJoinPoint);
        try {
            after_returning(proceedingJoinPoint, proceedingJoinPoint.proceed());
        } catch (Throwable e) {
            after_throwing(proceedingJoinPoint, e);
        }finally{
            af(proceedingJoinPoint);
        }
    }
}
```

AOP(Aspect Of Programing)

- Around Advice(@Aspect 어노테이션 버전)
  - aop.xml

```
<context:annotation-config />
<context:component-scan base-package="ver3_a"/>
<aop:aspectj-autoproxy proxy-target-class="true" />

<bean id="boy" class="ver3_a.Boy" />
<bean id="girl" class="ver3_a.Girl" />
```

  - Test.java

```
public static void main(String[] args) {
    ApplicationContext context =
            new GenericXmlApplicationContext("ver3_a/aop.xml");

    Girl girl = (Girl) context.getBean("girl");
    try {
        girl.doSomething();
    } catch (Exception e) {
    }
}
```

AOP(Aspect Of Programing)

- Aop Pointcut 표현식
  - Aspect를 적용할 위치를 지정할 때 사용한 Pointcut 설정을 보면 execution 명시자를 통해 옵션을 적용한 것을 볼 수 있다.
  - execution 같은 옵션들을 지시자라 부르며 execution 외에도 within, bean 같은 명시자들이 존재한다.

명시자	설명
execution	모든 것을 기준으로 하며 가장 정교한 PointCut을 만들 수 있다. 리턴타입 패키지경로 클래스명 메소드명(매개변수)가 쓰인다.
within	클래스나 패키지를 기준으로 하며 타입패턴 내에 해당하는 모든 것들을 PointCut
bean	Bean이름으로 PointCut

  - 각 패턴은 '*'를 이용하여 모든 값을 표현할 수 있다. 또한 '..'를 이용하여 0개 이상이라는 의미를 표현할 수 있다.

AOP(Aspect Of Programing)

- execution 명시자 표현식
  - 지시자 중에서 가장 많이 사용되는 것은 execution 지시자이며 해당 지시자를 통해 가장 정교한 Pointcut을 만들 수 있다.
  - 형식은 아래처럼 쓸 수가 있다.

execution(수식어패턴? 리턴타입패턴 클래스이름패턴?메서드이름패턴(파라미터패턴)

execution([public|protected|private] [return type] [className][메소드명](파라미터))

AOP(Aspect Of Programing)

- execution 명시자 표현식 예시

execution(public void set*(..))
 -> 리턴 타입이 void이고 메서드 이름이 set으로 시작하고, 파라미터가 0개 이상인 메서드 호출.
execution(* chobo.spring.chap03.core.*.*())
 -> chobo.spring.chap03.core 패키지의 파라미터가 없는 모든 메서드 호출.
execution(*.chobo.spring.chap03.core..*.*(..))
 -> chobo.spring.chap03.core 패키지 및 하위 패키지에 있는 파라미터가 0개 이상인 메서드 호출.
execution(Integer chobo.spring.chap03.core.WriteArticleService.write(..))
 -> 리턴 타입이 Integer인 WriteArticleService 인터페이스의 write() 메서드 호출.
execution(* get*(*))
 -> 이름이 get으로 시작하고 1개의 파라미터를 갖는 메서드 호출.
execution(* get*(*, *))
 -> 이름이 get으로 시작하고 2개의 파라미터를 갖는 메서드 호출.
execution(* read*(Integer, ..))
 -> 메서드 이름이 read로 시작하고, 첫 번째 파라미터 타입이 Integer이며, 1개 이상의 파라미터를 갖는 메서드 호출.

AOP(Aspect Of Programing)

- within 명시자 표현식
  - 메서드가 아닌 특정 타입에 속하는 메서드를 Pointcut으로 설정할 때 사용한다.
  - 보통은 클래스나 패키지 단위에 속한 모든 메서드들을 설정할 때 쓸 수 있다.
  - 포인트 컷에서 패키지와 클래스명만 있는 형태와 유사하다.

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

AOP(Aspect Of Programing)

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

```
bean(writeArticleService)
 -> 이름이 writeArticleService인 빈의 메서드 호출.
bean(*ArticleService)
 -> 이름이 ArticleServie로 끝나는 빈의 메서드 호출.
```

AOP(Aspect Of Programing)

- 표현식 연결
  - 위의 나온 표현식들은 Java 에서 “&&” 나 “||” 연산자를 이용하여 연결이 가능하다.
  - 단 xml에서는 and 혹은 or 로 표현한다.

```
@Pointcut("execution(* doSomething()) && execution(public String *())")
```

```
<aop:pointcut expression="execution(* doSomething()) and execution(public String *())" id="myPointcnt"/>
```