Spring Framework

✓ 원리를 알면 IT가 맛있다

Spring Framework for Beginners

chapter 03.

AOP
(Aspact Oriented Programming)

❑ 학습목표

- AOP 개요
- AOP 주요 용어
- 스프링에서의 AOP 적용
- 스프링 API를 이용한 AOP 적용방법 1
- POJO클래스를 이용한 AOP 적용방법 2
- @AspectJ 어노테이션을 이용한 AOP적용방법 3

❑ 1) AOP 개요

- AOP는 문제를 바라보는 관점을 기준으로 프로그래밍하는 기법을 의미한다.
- 문제를 해결하기 위한 핵심 관심 사항과 전체에 적용되는 공통관심 사항을 기준으로 프로그래밍함으로써 공통모듈을 여러 코드에 쉽게 적용할 수 있도록 도와준다.
- AOP에서 중요한 개념은 '횡단 관점의 분리(Separation of Cross-Cutting Concren)' 이다.

따라서 OOP를 더욱 OOP 답게 만들어 준다.

- 공통관심사항(cross-cutting concern)

  : 공통기능으로 어플리케이션 전반에 걸쳐 필요한 기능

  예> 로깅, 트랜잭션, 보안등

- 핵심관심사항(core concren)

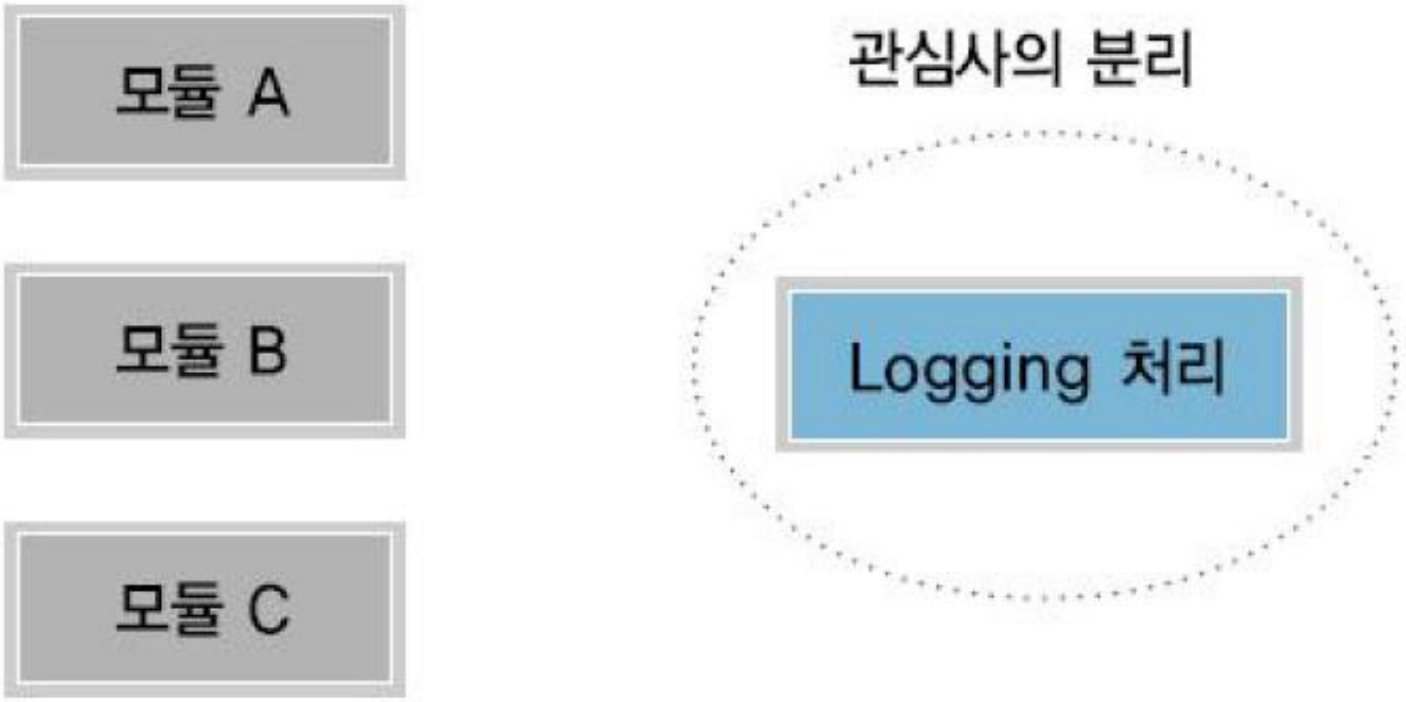
  :핵심로직, 핵심 비즈니스 로직

  예> 계좌이체, 이자계산, 대출처리등

- 공통관심사항(cross-cutting concern) 대표적인 예

  - logging and Tracing (around)
  - Transaction Management (around)
  - Security (before)
  - Caching (around)
  - Error Handling (after throwing)
  - Performance Monitoring (around)

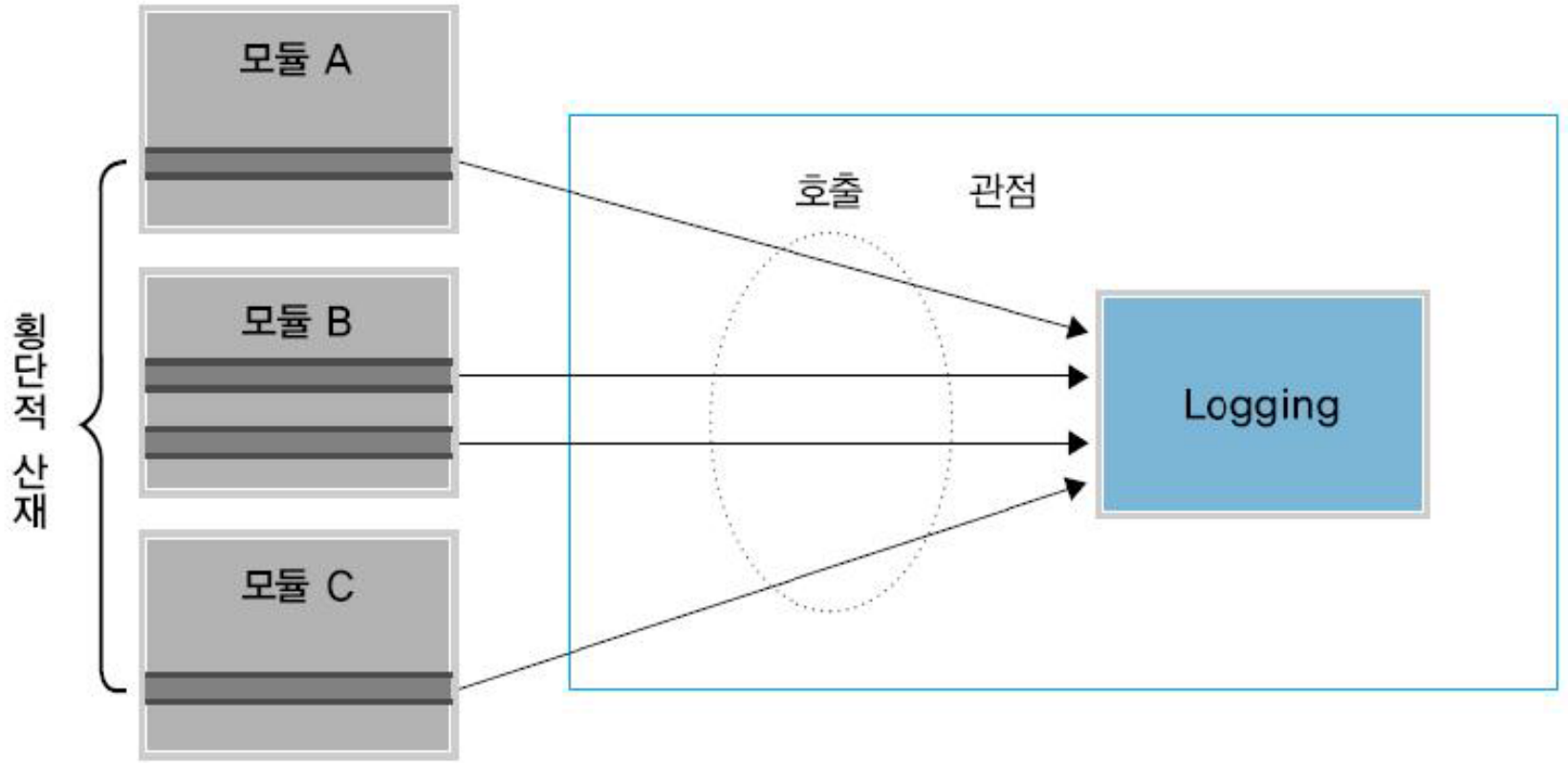
- OOP에서는 횡단 관점 분리를 위해 공통 기능들을 하나의 클래스라는 단위로 모으고 그것들을 모듈로부터 보호함으로써 재사용성과 유지보수성을 향상시킨다.



❑ 1) AOP 개요 - 횡단 관점의 분리

- 각 모듈로부터 공통기능으로 분리하는 것으로 성공했지만, 그 기능을 사용하기 위해 공통 기능을 호출하는 코드까지는 각 모듈로부터 분리할 수 없다.

  그렇기 때문에 분리한 공통 기능을 이용하기 위한 코드가 각 모듈에 횡단으로 산재하게 된다.



❑ 1) AOP 개요 - 횡단 관점의 분리

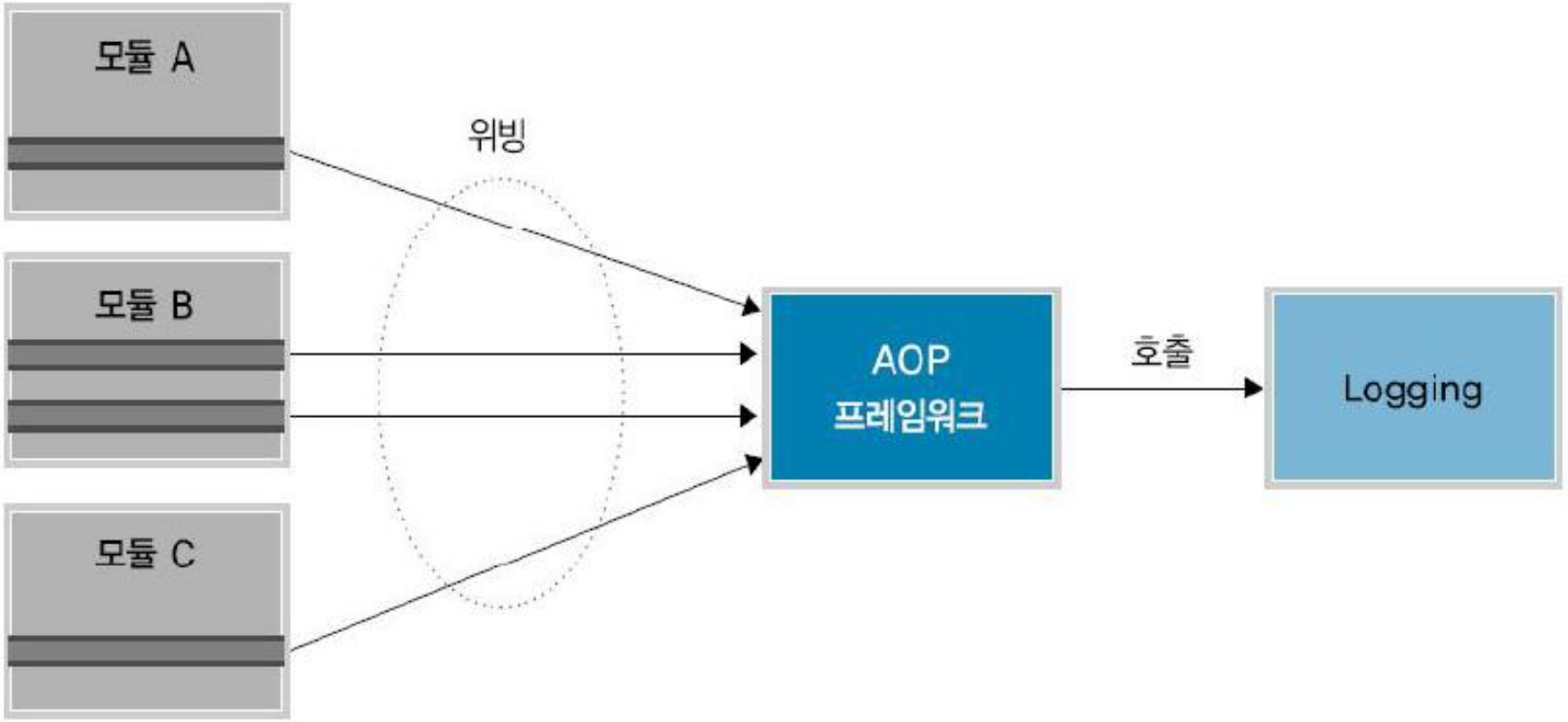
- AOP에서는 핵심 로직을 구현한 코드에서 공통기능을 직접적으로 호출하지 않는다.

- AOP에서는 분리한 공통 기능의 호출까지도 관점으로 다룬다. 그리고 이러한 각 모듈로 산재한 관점을 '횡단 관점'이라 부른다.

- AOP에서는 이러한 횡단 관점까지 분리함으로써 각 모듈로부터 관점에 관한 코드를 완전히 제거하는 것을 목표로 한다.

❑ 1) AOP 개요 - 횡단 관점의 분리

- AOP에서는 핵심 로직을 구현한 코드를 컴파일 하거나, 컴파일된 클래스를 로딩하거나 또는 로딩한 클래스를 실행할 때(runtime 시) 핵심 로직 구현 코드안에 공통 기능이 삽입된다.



- AspectJ
  - AOP 원천 기술 (1995년)
  - Startup 시간이 많이 걸릴만큼 무겁다.
  - 다른 플랫폼에서도 사용 가능하다.

- Spring AOP
  - 자바기반의 AOP 프레임워크이다. (dynamic proxies 사용)
  - Enterprise 환경에서 발생되는 문제 해결에 초점을 맞춘다.
  - 유일하게 Spring 프레임워크에서만 사용 가능하다.
  - 교육과정에서 살펴볼 내용이다.

❑ 3] AOP 주요 용어

- Join point
  - '클래스의 인스턴스 생성 시점','메소드 호출 시점' 및 '예외 발생 시점'과 같이 애플리케이션을 실행할 때 특정 작업이 시작되는 시점을 의미한다.
  - Advice을 적용 가능한 지점이다.
  - 스프링 AOP에서는 메소드 호출 시점만 지원된다.
- Advice
  - 특정한 Join Point에서의 행위.
    (삽입되어져 동작할 수 있는 코드로서 공통기능 포함)
  - Before Advice, After Advice, Around Advice, After Returning, After Throwing Advice
- Pointcut
  - Join point의 부분집합이다.
  - 실제로 Advice가 적용되는 Join point이다.
  - 스프링에서는 정규표현식이나 AspectJ 문법을 사용하여 Pointcut을 정의할 수 있다.

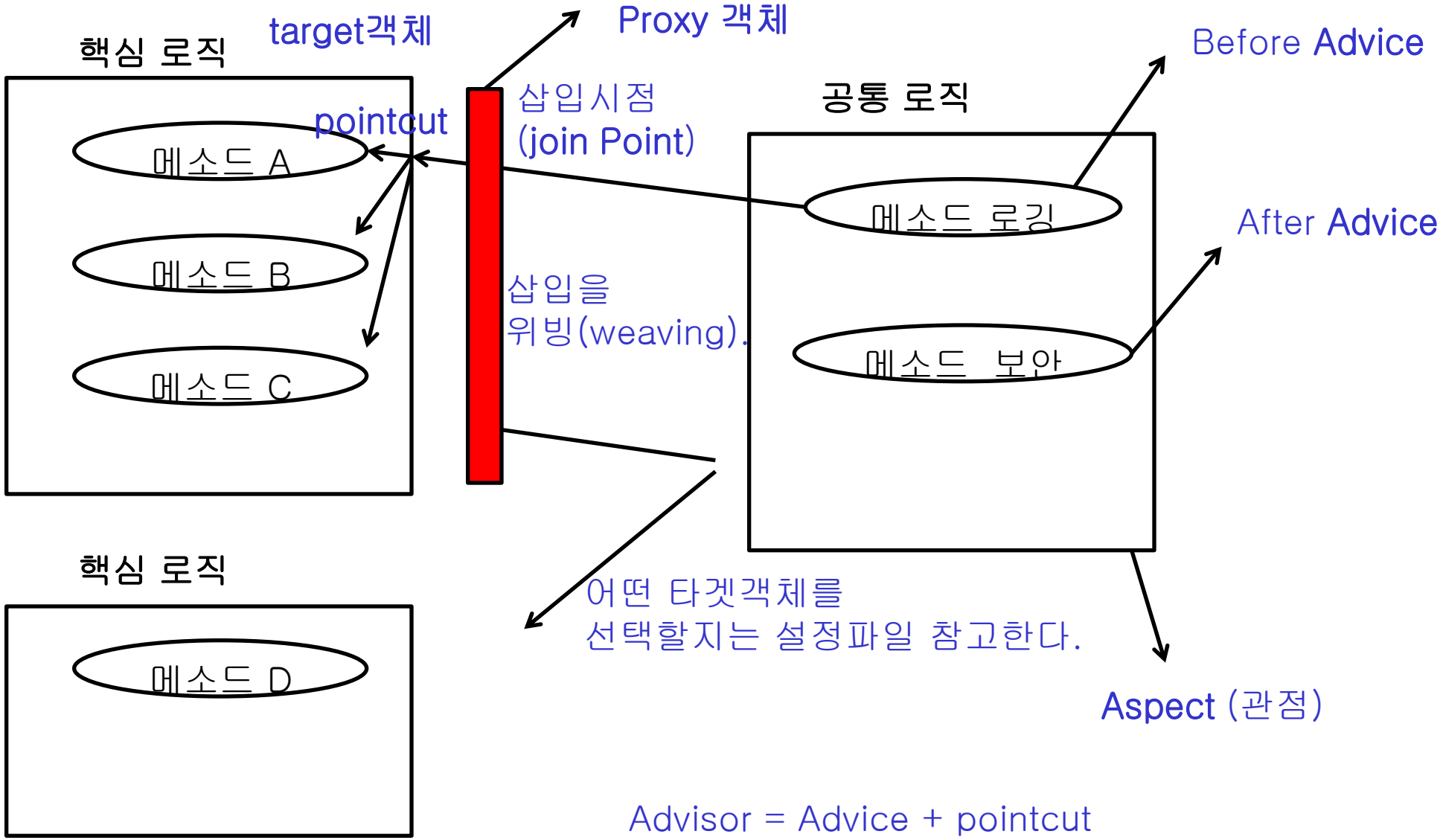
❑ 3] AOP 주요 용어

- Weaving
  - Advice(공통코드)를 핵심 로직 코드에 삽입하는것을 의미한다.

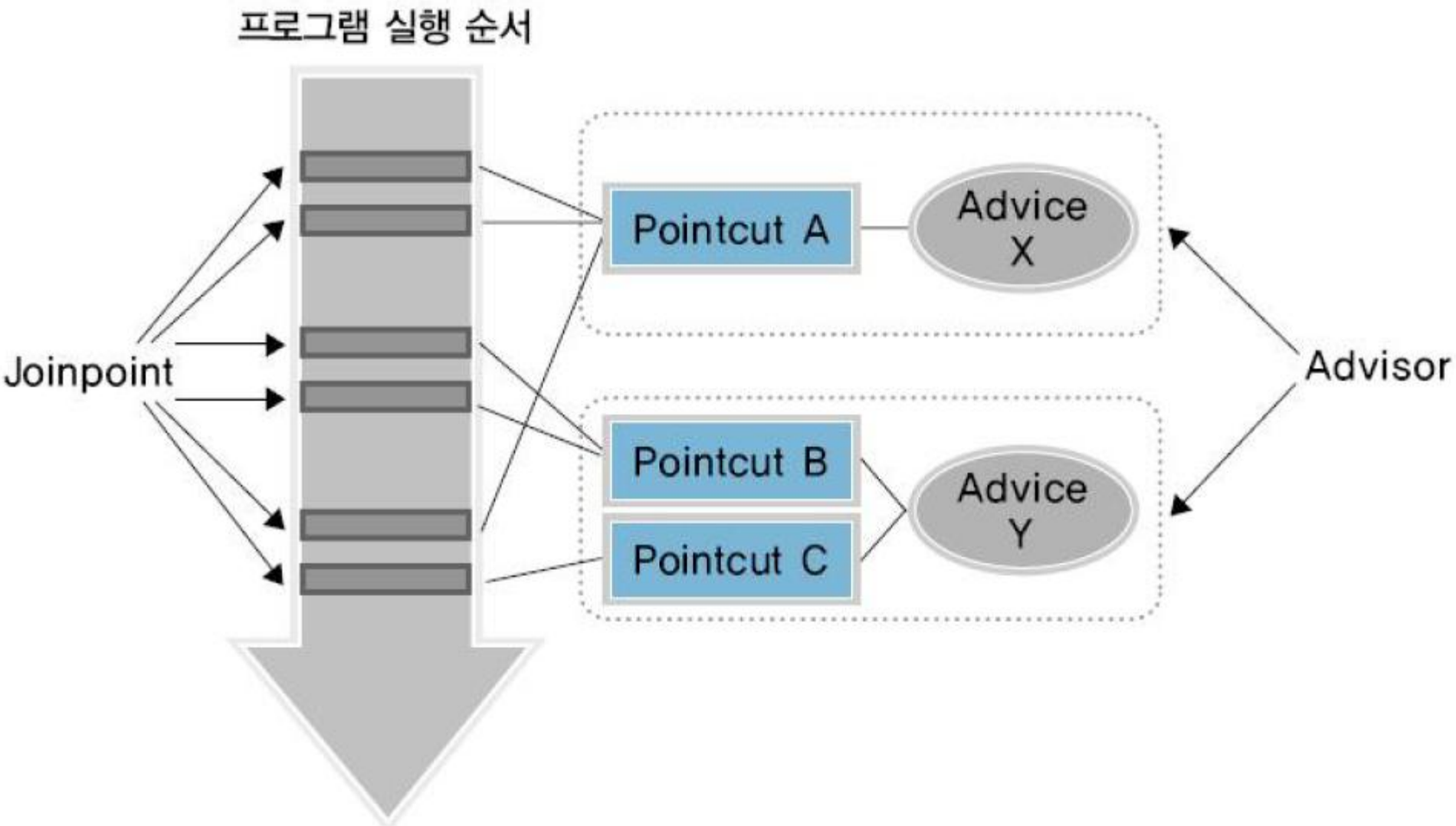
- Target object
  - 하나 또는 그 이상의 Aspect에 의해 Advice되는 객체를 의미한다.
  - 핵심 로직을 구현하는 클래스이다.
  - 스프링에서는 runtime 프록시를 통해서 구현된다.

- Aspect
  - 여러 객체에 공통으로 적용되는 공통 관점 사항을 의미한다.
  - 트랜잭션이나 보안 , 로깅등은 Aspect의 좋은 예이다.

❑ 3) AOP 주요 용어


핵심 로직
target객체
메소드 A
메소드 B
메소드 C
pointcut
Proxy 객체
삽입시점
(join Point)
삽입을
위빙(weaving).
공통 로직
Before Advice
메소드 로깅
After Advice
메소드 보안
어떤 타겟객체를
선택할지는 설정파일 참고한다.
Aspect (관점)
핵심 로직
메소드 D
Advisor = Advice + pointcut

❑ 3] AOP 주요 용어


프로그램 실행 순서
Joinpoint
Pointcut A
Advice X
Pointcut B
Pointcut C
Advice Y
Advisor

❑ 4] Weaving 방식

❍ Advice을 위빙하는 방식에는 다음과 같이 3가지 방식이 존재한다.

가. 컴파일시에 위빙 하기
- AOP가 적용된 새로운 클래스 파일이 생성된다.

나. 클래스 로딩시에 위빙 하기
- 로딩한 bytecode를 AOP가 변경하여 사용한다.

다. 런타임(runtime)시에 위빙 하기
- 프록시(proxy)를 이용한다.

❑ 5] 스프링에서의 AOP

- 스프링에서는 자체적으로 런타임(runtime)시에 위빙하는 '프록시 기반의 AOP'를 지원한다.
- 프록시 기반의 AOP는 메소드 호출 join point만 지원한다.
- 스프링에서 어떤 대상 객체에 대해 AOP를 적용할 지의 여부는 설정 파일을 통해서 지정한다.

  – 스프링은 설정 정보를 이용하여 런타임에 대상 객체에 대한 프록시 객체를 생성하게 된다.

  따라서 대상 객체를 직접 접근하는 것이 아니라 프록시를 통한 간접 접근을 하게 된다.
- 스프링는 완전한 AOP 기능을 제공하는 것이 목적이 아니라 Enterprise 어플리케이션을 구현하는 데 필요한 기능을 제공하는 것을 목적으로 하고 있다.
- 필드값 변경등과 같은 다양한 joinpoint를 이용하려면 AspectJ와 같은 다른 AOP프레임워크를 이용해야 된다.

❑ 6] 스프링에서의 AOP 구현 방법

- 스프링에서는 다음 3가지 방법으로 AOP구현을 지원한다.

  가. 스프링 API를 이용한 AOP 구현

  나. XML 기반의 POJO 클래스를 이용한 AOP 구현

  다. AspectJ 에서 정의한 @Aspect 어노테이션 기반의 AOP 구현.

*개발자가 직접 스프링 AOP API를 사용해서 AOP를 구현하는 경우는 많지 않음.

❑ 8) XML 기반의 POJO 를 이용한 AOP 방법1

- 스프링2 버전부터 POJO클래스를 이용하여 Advice를 개발하고 적용할 수 있는 방법이 추가되었다.

- 스프링2 버전의 xml 스키마 확장 기법을 통해 설정 파일도 보다 쉽게 설정이 가능하다.

❍ Aspect 작성

LogAspect.java

```
public class LogAspect {
    public void beforeLogging(){
        System.out.println("** 메서드 호출 전**");
    }
    public void afterLogging(Object returnValue){
        System.out.println("** 메서드 호출 후**");
    }
    public void throwingLogging(Exception ex){
        System.out.println("** 예외 발생  : "+ex.getMessage()+"**");
    }
    public void alwaysLogging(){
        System.out.println("** 항상 실행 **");
    }
}
```

❍ Aspect 작성

PerformanceAspect.java

```
public class PerformanceAspect {

    public Object timeCheck(ProceedingJoinPoint joinPoint) throws Throwable{

        Signature s= joinPoint.getSignature();
        String methodName = s.getName();
        long startTime = System.nanoTime();
        System.out.println("[Log]METHOD Before : " + methodName+" time check start");

        Object obj = null;
        try{
            ojb = joinPoint.proceed();
        }catch(Exception e){
            System.out.println("[Log]METHOD error : "+ methodName);
        }

        long endTime = System.nanoTime();
        System.out.println("[Log]METHOD After : " + methodName+" time check end");
        System.out.println("[Log] "+ methodName + " Processing time is "+(endTime - startTime)+"ns");
        return ojb;
    }
}
```

❑ 8) XML 기반의 POJO 를 이용한 AOP 방법2

- Aspect 설정
  - 설정파일에 aop 네임스페이스 및 네임스페이스와 관련된 스키마를 추가한다.
  - <aop:config> 태그를 이용하여 AOP관련정보를 설정한다.

applicationContext.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-2.5.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
  …
</beans>
```

❍ Aspect 설정 – Advice 정의 관련 태그

태그	설명
<aop:before>	메서드를 실행하기 전에 적용되는 Advice를 정의한다
<aop:after-returning>	메서드가 정상적으로 실행된 이후에 적용되는 Advice를 정의한다.
<aop:after-throwing>	메서드가 실행하는 도중 예외가 발생할 경우 적용되는 Advice를 정의한다.
<aop:after>	메서드가 정상적으로 실행되던지, 예외를 발생시키던지 여부에 상관없이 적용되는 Advice를 정의한다.
<aop:around>	메서드 실행 전, 후, 예외 발생시 적용 가능한 Advice를 정의한다.

❑ 9] @Aspect 기반의 POJO 를 이용한 AOP 방법3

- AspectJ 5 버전에 추가된 기능이다.
- 스프링에서는 2버전부터 @Aspect 어노테이션을 지원한다.

- 구현과정
  - XML기반의 POJO를 이용한 방법과 유사하다.
    차이점은 다음과 같다.

  가. @Aspect 어노테이션을 이용해서 Aspect 클래스를 구현한다.
  이때 Aspect 클래스는 Advice를 구현한 메소드와 Pointcut을 포함한다.

  나. XML 설정파일에서 <aop:aspectj-autoproxy />을 설정한다.

- 어노테이션
  - @Aspect
  - @Pointcut
  - @Around
  - @Before
  - @AfterReturning
  - @AfterThrowing

❑ 9) @Aspect 기반의 POJO 를 이용한 AOP 방법3

PerformanceAspect.java

```
@Aspect
public class PerformanceAspect {

    @Pointcut("execution(public * com.consolution.test.aop..*sayHello(..))")
    private void profileTarget() {}

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

        Signature s= joinPoint.getSignature();
        String methodName = s.getName();
        long startTime = System.nanoTime();

        System.out.println("[Log]METHOD Before --> " + methodName+" time check start");

        Object o = null;
        try{
            o= joinPoint.proceed();
        }catch(Exception e){
             System.out.println("[Log]METHOD error --> "+ methodName);
        }

        long endTime = System.nanoTime();
        System.out.println("[Log]METHOD After --> " + methodName+" time check end");
        System.out.println("[Log] "+ methodName + " Processing time is "+(endTime - startTime)+"ns");
        return o;

    }
}
```

❑ 9) @Aspect 기반의 POJO 를 이용한 AOP 방법3

applicationContext.xml

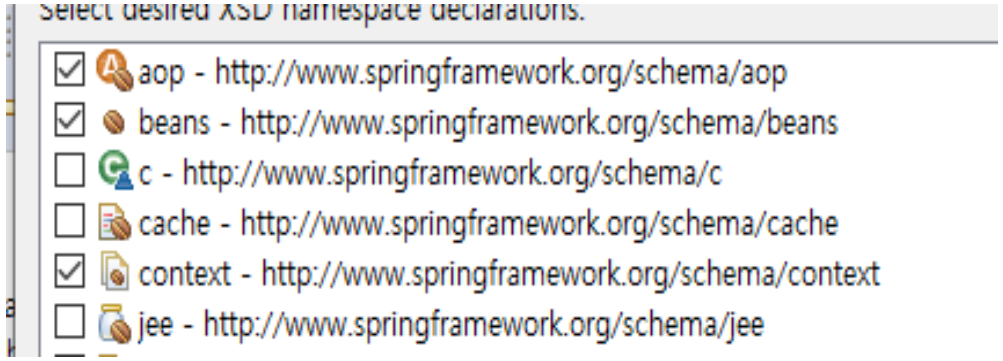
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.1.xsd">

    <aop:aspectj-autoproxy />

    <bean id="greetingTarget" class="com.consolution.test.aop.GreetingServiceImpl">
        <property name="greeting">
            <value>Hello_annot</value>
        </property>
    </bean>

    <bean id="performanceAspect" class="com.consolution.test.aop.annot.PerformanceAspect" />
</beans>
```

❑Aop config.xml 사용시 namespace 설정



Pom.xml의 설정 http://mavenrepository.com

<scope>runtime</scope>삭제 실행시 사용- 컴파일 안됨

```
<!--
https://mvnrepository.com/artifact/org.aspectj
/aspectjweaver Aop-->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.1</version>
</dependency>
```

```
<!-- Spring and Transactions -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>${spring-framework.version}</version>
</dependency>
```

❍ Aspect 설정

```java
public class Person {

    //핵심기능
   public void getInfo() {
       System.out.println("getInfo");
   }
}
```

```java
public class PersonBeforeAspect {
        //공통로직 구현부
        public void mesgBefore() {

            System.out.println("공통로직 aop의 mesgBefore()+++++++++++++++");
        }
}
```

```xml
 <!-- 공통기능 제공 AOP bean 등록 -->
<bean id="beforeAspect" class="com.aspect.PersonBeforeAspect"/>
<!--  aspect 설정 : advice를 어떤 pointcut에 적용할 지 설정-->
<aop:config>
    <aop:aspect id="mesgAspect" ref="beforeAspect">
    <!-- 공통로직함수에 pointcut등록 -->
        <aop:pointcut id="publicMethod"
            expression="execution(public * com..*.*(..))"
        ></aop:pointcut>
        <!--수식어(public 리턴타입 * com..패키지아래*모든클래스.*모든함수(..)매개변수 -->
        <aop:before pointcut-ref="publicMethod"  method="mesgBefore"/>
        <!-- 실행할 공통로직함수 등록 -->
    </aop:aspect>
</aop:config>
  <!-- target class 생성(핵심로직)-->
    <bean id="xxx" class="com.spring.Person" />
</beans>
```

```
[20-09-26 08:20:57] [DEBUG] [o.s.b.f.s.DefaultListableBeanF
공통로직 aop의 mesgBefore()+++++++++++++++
getInfo
[20-09-26 08:20:57] [DEBUG] [o.s.b.f.s.DefaultListableBeanF
공통로직 aop의 mesgBefore()+++++++++++++++
mesg
```

❑ 8) XML 기반의 POJO 를 이용한 AOP 방법2

- 빈 객체 사용
  - 스프링 API로 구현한 경우처럼 프록시 객체를 설정하지 않으므로 빈 객체를 얻어와 사용하면 된다.
  - 포인트컷에 expression에 설정한 정보로 자동으로 Advice가 해당 포인트컷에 적용된다.

```
7  import org.aspectj.lang.annotation.Before;
8  import org.aspectj.lang.annotation.Pointcut;
9
10 @Aspect
11 public class PersonAfterAspect {
12     //advice와 poing cut의 분리
13     @Pointcut("execution(* mesg(..))")
14     public void xx() {
15
16     }
17     @Before("xx()")
18     public void mesgBefore(JoinPoint x) {
19         Signature s= x.getSignature();
20         System.out.println(">>> targetClass의 클래스명"+ s.getClass());//
21         System.out.println(">>> targetClass의 메소드명"+ s.getName());//
22         System.out.println("mesgBefore==================================
23     }
24     //advice와 pointcut을 같이 표현
25     @After("execution(* getInfo(..))")
26     public void getInfoAfter() {
27         System.out.println("getInfoAfter================================
28     }
```

```
3  public class Person {
4
5      //핵심기능
6      public void getInfo() {
7          System.out.println("getInfo");
8      }
9
10     public void mesg() {
11         System.out.println("mesg");
12     }
```

❑ 8) XML 기반의 POJO 를 이용한 AOP 방법2

❍ Config.xm설정

Select XSD namespaces to use in the configuration file

- ☑ aop - http://www.springframework.org/schema/aop
- ☑ beans - http://www.springframework.org/schema/beans
- ☐ c - http://www.springframework.org/schema/c
- ☐ cache - http://www.springframework.org/schema/cache
- ☑ context - http://www.springframework.org/schema/context
- ☐ jee - http://www.springframework.org/schema/jee
- ☐ lang - http://www.springframework.org/schema/lang
- ☐ p - http://www.springframework.org/schema/p
- ☐ task - http://www.springframework.org/schema/task
- ☐ tx - http://www.springframework.org/schema/tx
- ☐ util - http://www.springframework.org/schema/util

```
8           http://www.springframework.org/schema/aop http://www.spring
9
10  <!-- AOP활성화 -->
11    <aop:aspectj-autoproxy />
12    <!-- target class -->
13      <bean id="xxx" class="com.spring.Person" />
14
15  <!-- AOP aspect -->
16      <bean id="afterAspect" class="com.aspect.PersonAfterAspect" />
17
18  </beans>
19
```

❑ 8) XML 기반의 POJO 를 이용한 AOP 방법2

- ❑ @AfterReturning(pointcut = "execution(* getInfo(..))", returning="x")
- ❑ 리턴값의 사용

```
8         http://www.springframework.org/schema/aop http://www.spring
9
10  <!-- AOP활성화 -->
11    <aop:aspectj-autoproxy />
12    <!-- target class -->
13      <bean id="xxx" class="com.spring.Person" />
14
15  <!-- AOP aspect -->
16      <bean id="afterAspect" class="com.aspect.PersonAfterAspect" />
17
18  </beans>
19
```

```
1  package com.spring;
2
3  public class Person {
4
5      //핵심기능
6      public String getInfo() {
7          System.out.println("getInfo");
8          return "홍길동";
9      }
```

```
7  @Aspect
8  public class PersonAfterAspect{
9
10     @AfterReturning(pointcut = "execution(* getInfo(..))",
11         returning="x")  //pointcut 지정 함수 정의 , 실행 후 return 값의 저장
12     public void getInfoAfterReturning(JoinPoint point, Object x) {
13         System.out.println("aftereReturn JoinPoint=====  "+ point.getTarget());
14         System.out.println("aftereReturn JoinPoint returning data x=====  "+ x);
15     }
16
```

❑ 8) XML 기반의 POJO 를 이용한 AOP 방법2

@AfterThrowing(pointcut = "execution(* getInfo(..))",throwing = "x")
예외발생시

```java
package com.spring;

public class Person {

    //핵심기능
    public void getInfo() throws Exception{
        throw new Exception("에러발생");
//      System.out.println("getInfo");
//      int n = 10/0;
    //    return "홍길동";
    }
```

```java
@Aspect
public class PersonAfterAspect {


    @AfterThrowing(pointcut="execution(* getInfo(..))",
            throwing="x")
    public void getInfoAfterReturning(JoinPoint point, Exception x) {
        System.out.println("getInfoAfterReturning:" +x.getMessage());
    }
```

```java
public static void main(String[] args) {

    GenericXmlApplicationContext ctx=
            new GenericXmlApplicationContext("classpath:com/spring/person.xml");
    Person p = ctx.getBean("xxx",Person.class);
try {
    p.getInfo();
    //String name= p.getInfo();
  // System.out.println(name);
}catch (Exception e) {
    System.out.println("catch getMessgage()"+e.getMessage());
}
System.out.println("end");
```

❑ @Around – ProceedingJoinPoint 객체사용하여 공통로직 실행

```java
package com.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class PersonAroundAspect {


    @Around("execution(* getInfo(..))")
    public Object  getInfoAfterReturning(ProceedingJoinPoint  pjp)
            throws Throwable {
        //before
        System.out.println("before");
        Object retVal = pjp.proceed(); //핵심로직함수 실행
        //after
        System.out.println("after:" + retVal);
        return retVal;
    }

}
```

❍ AspectJ 의 Pointcut 표현식

- execution(수식어패턴? 리턴타입패턴 클래스이름패턴?이름패턴(파라미터패턴))
  - 수식어패턴 : 생략 가능, public, protected 등이 옴
  - 리턴타입패턴 : 리턴 타입 명시
  - 클래스이름패턴 : 생략 가능, 클래스 이름 명시
  - 이름패턴 : 메서드 이름 명시
  - 파라미터패턴 : 매칭될 파라미터에 대해서 명시
- 각 패턴은 * 을 이용해 모든 값을 표현 가능
- .. 을 이용하면 0개 이상이라는 의미를 표현
  - 클래스이름패턴에서 패키지 이름 뒤에 ..을 쓰면 서브패키지도 찾는다.
  - 파라미터패턴에 .. 을 쓰면 파라미터가 0개 이상
    - (*) 으로 하면 파라미터를 1개 포함해야 함
    - (* , *) 으로 하면 파라미터가 2개 이어야 함
    - (Integer, ..)으로 하면 첫 번째 파라미터는 Integer형이며, 1개 이상의 파라미터를 가짐
- 예)
  - execution(* com.myapp.aop..*.select*(..))
    - com.myappa.aop패키지 및 하위 패키지에 있는 파라미터가 0개 이상인 메서드 이름이 select 로 시작하는 메서드 호출(리턴타입과도 무관함)

- execution(public void set*(..))

리턴타입void이며 set으로 시작하고 파라미터가 0개 이상임

- execution(* chap07.*.*())

Chap07패키지의 모든 파라미터가 없는 모든 메소드

- execution(* chap07..*.*(..))

Chap07하위 패키지의 파라미터가 0개 이상인 메소드

- execution(int chap07.Cal.factorial(..))

Chap07의 리턴타입이 int인 factorial

- execution(* get*(*,*))

Get으로 시작하고 매개변수가 2개인 메소드

- AOP 개요
- AOP 주요 용어
- 스프링에서의 AOP 적용
- 스프링 API를 이용한 AOP 적용
- POJO클래스를 이용한 AOP 적용
- @AspectJ 어노테이션을 이용한 AOP적용

Thank you