스프링 AOP

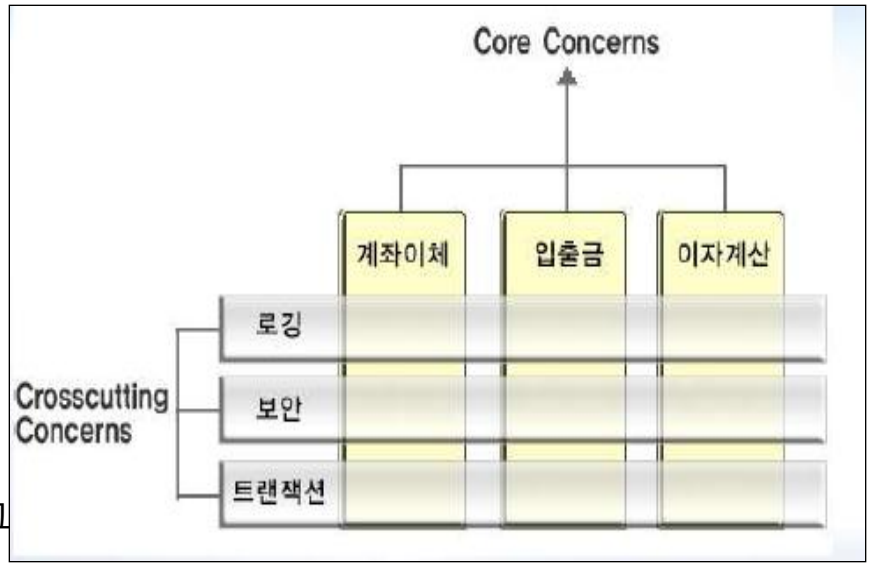
Aspect Oriented Programming

- AOP
  - 장점
  - AOP 주요 용어
  - Proxy
  - Weaving
- Spring에서의 AOP
- XML 기반의 POJO 클래스를 이용한 AOP 구현
- 애노테이션 기반의 AOP 구현
- 표현식

AOP[Aspect Oriented Programming] 개요

- 그림 설명
  - 핵심 비즈니스 로직
    - 계좌이체, 입출금, 이자계산
  - 핵심 로직에 적용되는 공통 로직
    - 로깅, 보안, 트랜잭션

[그



AOP 개요

- 용어 정리
  - Concern
    - 애플리케이션을 개발하기 위하여 관심을 가지고 구현 해야 하는 각각의 기능들을 관심 사항(Concern)이라 함
  - core concern
    - 핵심 관심 사항
    - 해당 애플리케이션이 제공하는 핵심이 되는 비즈니스 로직 의미
  - cross-cutting concern
    - 횡단[공통] 관심 사항
    - 하나의 영역에서만 활용되는 고유한 관심 사항(Concern)이 아니라, 여러 클래스 혹은 여러 계층의 애플리케이션 전반에 걸쳐서 공통적으로 필요로 하는 기능들 의미

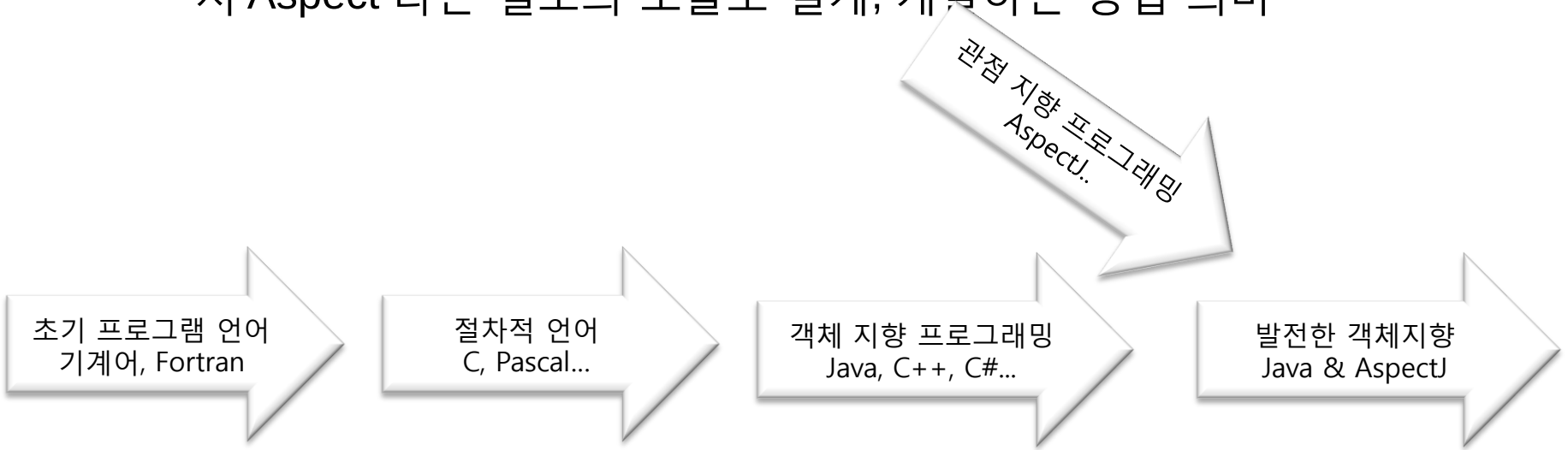
AOP 개요

- Aspect(애스펙트)란?

- 보안과 로깅 모듈처럼 그 자체로 애플리케이션의 핵심 기능을 다루고 있지는 않지만, 애플리케이션을 구성하는 중요한 요소이고, 핵심 기능에 적용되어야만 의미를 갖는 특별한 모듈을 의미

AOP(Aspect-Oriented Programming) 란?

- AOP에서 중요한 개념
  - 문제를 바라보는 관점을 기준으로 프로그래밍하는 기법
  - 「횡단 관심 사항의 분리(Separation of Cross-Cutting Concern)」
  - 로깅, 보안, 트랜잭션처럼 비즈니스 로직의 여러 부분에 공통적으로 사용되는 부가적인 횡단 관심 사항들을 비즈니스 로직에서 분리해서 Aspect 라는 별도의 모듈로 설계, 개발하는 방법 의미



AOP를 적용하지 않은 OOP

AOP를 적용하지 않은 OOP 개발시 발생될 수 있는 문제점

- 중복되고 지저분한 코드
  - 핵심 기능의 코드 사이 사이에 적용되는 횡단 관심의 로직들로 인해 코드의 양도 많아지고 지저분해 지므로 가독성 또한 떨어짐

- 개발 생산성 및 재사용성의 저하
  - 핵심 로직의 코드 개발에만 집중할 수 없고, 전체적으로 많은 부분에 영향을 미치는 횡단 관심 로직도 적용해야 하기 때문에 개발 생산성 및 코드의 재 사용성도 떨어짐

AOP를 적용한 OOP

- 해결책
  - OOP의 단점을 극복하고자 핵심 관심 로직과 횡단 관심 로직을 분리해서 각 모듈로 개발한 이후에 핵심 로직 코드의 수정 없이도 횡단 관심 로직을 적용하도록 하는 것이 AOP의 기술
  - 이처럼 AOP는 OOP의 대체용이 아닌 OOP와 같이 적용하면서 OOP의 단점을 보완해 줄 수 있는 언어

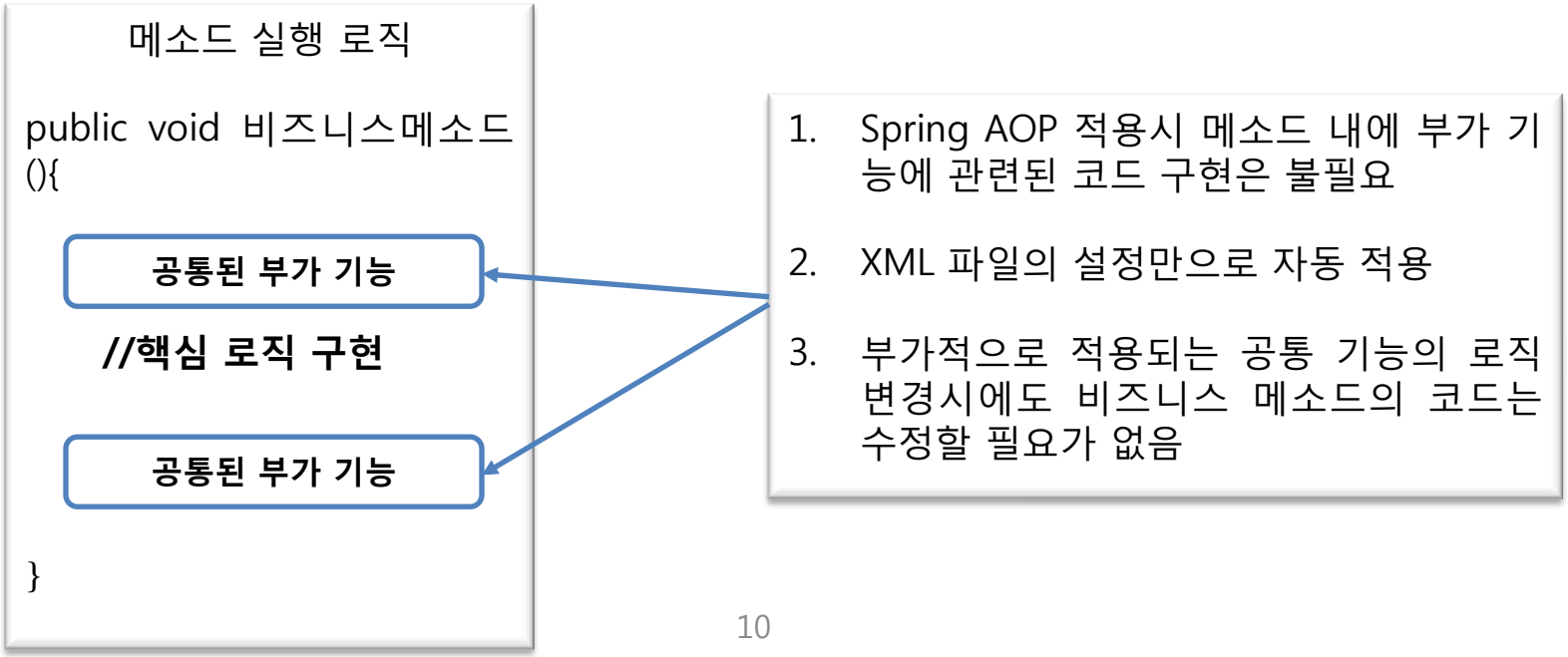
- 애플리케이션을 다양한 측면에서 독립적으로 모델링하고, 설계하고, 개발할 수 있도록 해주며 핵심 관점 및 횡단 관심 관점에서 해당 부분에만 집중해서 설계하고 개발 할 수 있기 때문에 애플리케이션을 특정한 관점을 기준으로 바라 볼 수 있게 해 준다는 의미에서 AOP를 '관점지향 프로그래밍'이라고 함

AOP를 적용한 OOP

- AOP를 적용하게 되면 개발자들은 핵심 관심 사항 코드에도 비즈니스 기능 처리에 대한 구현만 할 수 있음
- 필요에 따라서 횡단관심 사항의 로직들을 언제든지 쉽게 추가, 삭제할 수 있음
- AOP 기술을 프로젝트에 적용하기 위해서는 Spring 또는 AspectJ와 같이 AOP를 지원하는 프레임워크를 주로 사용

AOP 장점 – AOP를 적용한 OOP

- AOP 적용
  - 개발자들은 횡단 관심 모듈을 각각 독립된 모듈로 중복 없이 작성
  - XML 설정 및 애노테이션으로 핵심 관심 모듈과 결합
  - 장점 : 서로 독립성을 가진 다 차원의 모듈로 작성 할 수 있음



횡단 관점의 분리를 위한 개발 방법 - AOP

- AOP에서는 핵심 로직을 구현한 코드에서 공통 기능을 직접적으로 호출하지 않음
- AOP에서는 분리한 공통 기능의 호출까지도 관점으로 다룸
- 이러한 각 모듈로 산재한 관점을 **횡단 관점**이라 부름
- AOP에서는 이러한 횡단 관점까지 분리함으로써 각 모듈로부터 공통 로직으로 인한 중복 코드를 완전히 제거하는 것을 목표로 함

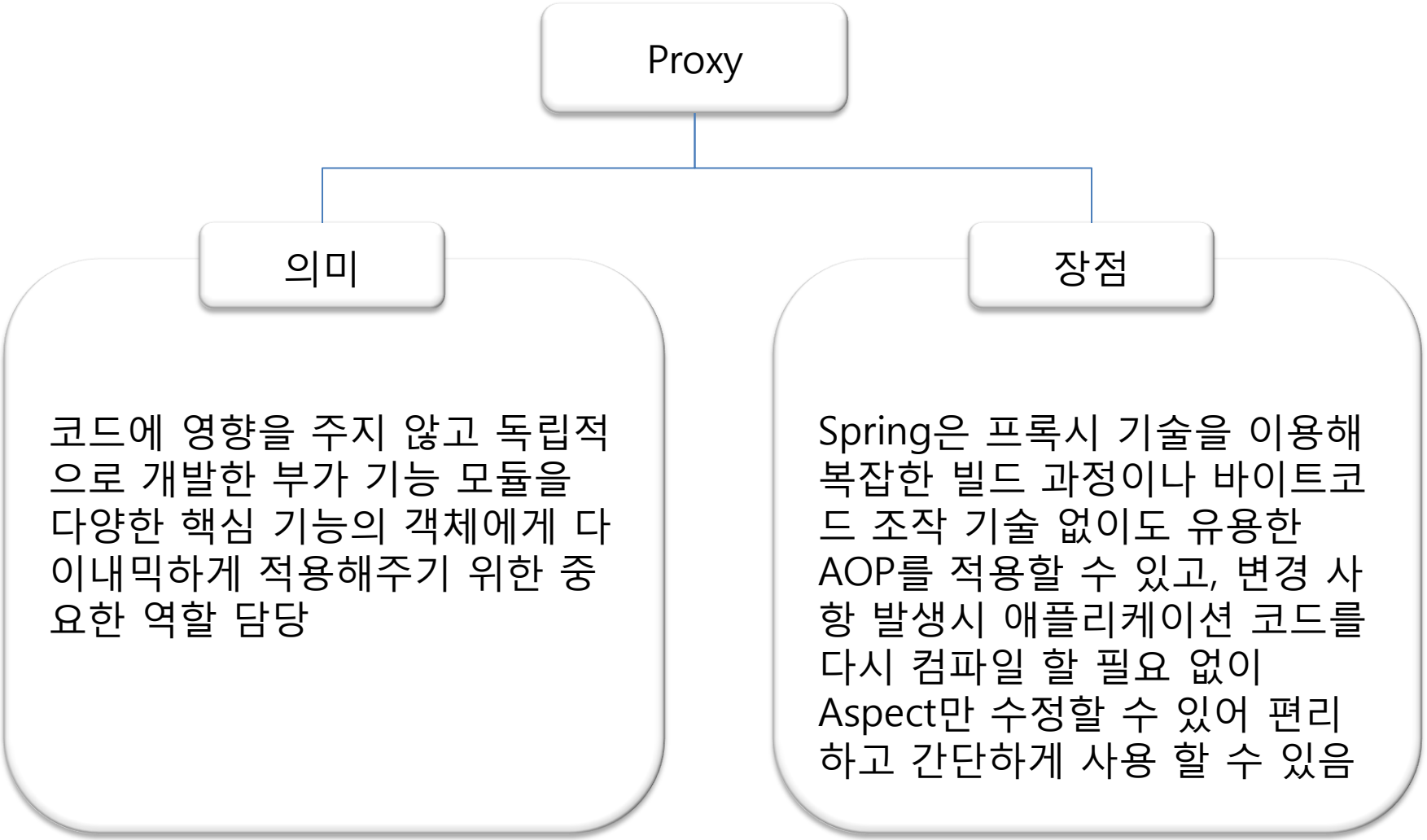
AOP 주요 용어

용어	설명
Target	핵심 로직을 구현하는 클래스 공통 관심 사항을 적용 받게 되는 대상으로 어드바이스가 적용되는 객체
Aspect	여러 객체에 공통으로 적용되는 공통 관심 사항
Advice	조인 포인트에 삽입되어 동작할 수 있는 공통 관심 사항의 코드 *동작시점 : Spring에서는 조인포인트 실행 전, 후로 before, after, after returning, after throwing, around로 구분
joinpoint	「클래스의 인스턴스 생성 시점」,「메소드 호출 시점」 및「예외 발생 시점」과 같이 애플리케이션을 실행할 때 특정 작업이 시작되는 시점으로 Advice를 적용 가능한 지점 즉 어드바이스가 적용될 수 있는 위치
pointcut	여러 개의 Joinpoint를 하나로 결합한(묶은) 것
Advisor	Advice와 Pointcut를 하나로 묶어 취급한 것
weaving	공통 관심 사항의 코드인 Advice를 핵심 관심 사항의 로직에 적용 하는 것을 의미

Weaving방식- Advice를 위빙하는 방식

용어	설명
컴파일시 위빙하기	AOP가 적용된 새로운 클래스 파일이 생성됨 AspectJ에서 사용 하는 방식
클래스 로딩시 위빙하기	로딩한 바이트 코드를 AOP가 변경하여 사용 AOP 라이브러리는 JVM이 클래스를 로딩할때 클래스 정보를 변경할 수 있는 에이전트 제공 (원본 클래스 파일 변경 없이 클래스를 로딩할 때 JVM이 변경된 바이트 코드를 사용하도록 함)
런타임시 위빙하기	소스 코드나 클래스 정보 자체를 변경하지 않음 프록시를 이용하여 AOP적용 프록시 기반의 AOP는 핵심 로직을 구현한 프록시를 통해서 핵심 로직을 구현한 객체에 access 프록시 기반의 제한사항 : 메소드 호출할 경우에만 Advice를 적용

Proxy


Proxy
의미
코드에 영향을 주지 않고 독립적으로 개발한 부가 기능 모듈을 다양한 핵심 기능의 객체에게 다이내믹하게 적용해주기 위한 중요한 역할 담당
장점
Spring은 프록시 기술을 이용해 복잡한 빌드 과정이나 바이트코드 조작 기술 없이도 유용한 AOP를 적용할 수 있고, 변경 사항 발생시 애플리케이션 코드를 다시 컴파일 할 필요 없이 Aspect만 수정할 수 있어 편리하고 간단하게 사용 할 수 있음

Spring에서의 AOP

- Spring에서는 자체적으로 런타임시에 위빙하는 “프록시 기반의 AOP”를 지원
- 프록시 기반의 AOP는 메소드 호출 조인포인트만 지원
- Spring에서 어떤 대상 객체에 대해 AOP를 적용할 지의 여부는 설정 파일을 통해서 지정
- Spring은 설정 정보를 이용하여 런타임에 대상 객체에 대한 프록시 객체를 생성 따라서, 대상 객체를 직접 접근하는 것이 아니라 프록시를 통한 간접 접근을 하게 됨

Spring에서의 AOP

- Spring은 완전한 AOP 기능을 제공하는 것이 목적이 아니라 Enterprise Application을 구현하는데 필요한 기능을 제공하는 것을 목적으로 하고 있음

- 필드값 변경 등 다양한 조인포인트를 이용하려면 AspectJ와 같은 다른 AOP솔루션을 이용해야 함

Spring AOP 프록시의 종류

- 핵심 로직 구현 방법에 따른 proxy 종류

핵심 로직의 클래스 구현시 interface가 정의되어 있는 경우

JDK 동적 프록시 사용

JDK 프록시는 오직 인터페이스의 프록시만을 생성
대상 객체가 인터페이스를 구현하고 있다면,
Spring은 자바 리플렉션 API가 제공하는 java.lang.reflect.Proxy 를 이용하여 프록시 객체를 생성

interface가 정의 되어 있지 않은 핵심 로직의 클래스인 경우

CGlib 사용

바이트코드 생성 라이브러리를 이용하여 동적으로 각각의 클래스에 대한 프록시 객체를 생성
이미 생성한 클래스가 있을 때는 그것을 재사용

Spring에서의 AOP

- AOP 구현을 위한 개발 방법

XML 스키마 기반의
POJO 클래스를 이용한 AOP 구현

애노테이션 기반의 AOP 구현

Spring에서의 AOP – Advice 개요

- Advice란?
  - 조인 포인트에 삽입되어 동작할 수 있는 공통 관심 사항의 코드
  - 동작시점 : Spring에서는 조인포인트 실행 전, 후로 before, after, after returning, after throwing, around로 구분

Spring에서의 AOP - Advice 종류

Advice 종 류	XML 스키마 기반의 POJO 클래스를 이용	@Aspect 애노테이션 기반	설 명
Before	<aop:before>	@Before	target 객체의 메소드 호출시 호출 전에 실행
AfterReturning	<aop:after-returning>	@AfterRetuning	target 객체의 메소드가 예외 없이 실행된 후 호출
AfterThrowing	<aop:after-throwing>	@AfterThrowing	target 객체의 메소드가 실행하는중 예외가 발생한 경우에 실행
After	<aop:after>	@After	target 객체의 메소드를 정상 또는 예외 발생 유무와 상관없이 실행 try의 finally와 흡사
Around	<aop:around>	@Around	target 객체의 메소드 실행 전, 후 또는 예외 발생 시점에 모두 실행해야 할 로직을 담아야 할 경우

XML 기반의 POJO 클래스를 이용한 AOP 구현

- Spring API를 사용하지 않은 POJO 클래스를 이용하여 어드바이스를 개발하고 적용할 수 있는 방법이 추가

- XML 확장을 통해 설정 파일도 보다 쉽게 작성 할 수 있음

XML 기반의 POJO 클래스를 이용한 AOP 구현

개발 단계

Advice 클래스를 구현

aop 네임스페이스가 있는 XML 설정 파일에 Advice 클래스를 빈으로 등록

<aop:config> 태그를 이용해서 Advice를 어떤 포인트컷에 적용할지 지정

XML 기반의 POJO 클래스를 이용한 AOP 구현

Advice 클래스를 구현

```
 4 import org.aspectj.lang.ProceedingJoinPoint;
 5
 6 public class CommonAdvice {
 7
 8    /* 1. around advice
 9     * 2. 실행 로직  - Advice가 적용될 target 객체를 호출하기 전화 후를 구해서 target 객체의
10     *                메소드 호출 실행 시간 출력*/
11    public void around(ProceedingJoinPoint point) throws Throwable{
12       String methodName = point.getSignature().getName();
13       String targetName = point.getTarget().getClass().getName();
14       System.out.println("target 클래스명 : " + targetName + " 및 메소드명 : " + methodName);
15
16       long startTime = System.nanoTime();
17       System.out.println("[Log]핵심 로직 메소드 호출전 --> "+ methodName+" time check start");
18
19       Object obj = point.proceed();
20
21       long endTime = System.nanoTime();
22       System.out.println("[Log]핵심 로직 메소드 호출 후 --> " + methodName+" time check end");
23       System.out.println("[Log] "+ methodName+" 실행 소요 시간 "+(endTime - startTime)+"ns");
24
25       System.out.println(targetName + " 메서드 실행후 리턴된 데이터 : " + obj);
26    }
27
```

XML 기반의 POJO 클래스를 이용한 AOP 구현

aop 네임스페이스가 있는 XML 설정 파일에 Advice 클래스를 빈으로 등록

<aop:config> 태그를 이용해서 Advice를 어떤 포인트컷에 적용할지 지정

설정 파일에 aop 네임스페이스 및 네임스페이스와 관련된 스키마 추가

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4
5     xmlns:aop="http://www.springframework.org/schema/aop"
6
7     xsi:schemaLocation="http://www.springframework.org/schema/beans
8        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
9
10       http://www.springframework.org/schema/aop
11       http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
12
13    <!-- 중략 -->
14
15 </beans>
16
```

XML 기반의 POJO 클래스를 이용한 AOP 구현

aop 네임스페이스가 있는 XML 설정 파일에 Advice 클래스를 빈으로 등록

<aop:config> 태그를 이용해서 Advice를 어떤 포인트컷에 적용할지 지정

<aop:config>태그를 이용하여 AOP관련정보를 설정

```
12
13    <bean id="common" class="spring.study.aop.CommonAdvice" />
14
15    <bean id="data" class="spring.study.aop.DataImpl"></bean>
16
17    <aop:config>
18       <aop:aspect id="commonAdvice" ref="common">
19
20          <aop:pointcut id="aroundPush" expression="within(spring.study.aop.*)"/>
21
22          <aop:around pointcut-ref ="aroundPush" method="around" />
23
24       </aop:aspect>
25    </aop:config>
26
```

spring.study.aop package의 모든 메소드 호출 전후로 공통 로직으로 CommonAdvice의 around()를 호출 의미

빈 객체 사용

```
1  package spring.study.aop;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  public class AopTest {
7
8      public static void main(String[] args) {
9
10         ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
11         Data data = (Data)ctx.getBean("data");
12
13         data.around("김혜경");
14
15     }
16 }
```

외부 Advice 설정

- 외부에서 정의한 트랜잭션 Advice를 포인트컷과 연결할 수 있는 <aop:advisor>
- <aop:advisor>는 <aop:aspect> 와 달리 외부에서 정의한 어드바이스를 포인트컷으로 연결할 목적으로 사용
- <tx:advice> 태그를 이용하여 선언한 txAdvice 라는 이름의 외부에 정의된 어드바이스를 포인트컷으로 지정하기 위해서 <aop:config> 태그 내에 <aop:advisor> 태그를 사용

```
<aop:config>
    <aop:pointcut id=" businessLogic" expression="within(spring.study.aop.*)"/>
    <aop:advisor pointcut-ref="businessLogic" advice-ref="txAdvice"/>
</aop:config>
<tx:advice id="txAdvice">
    <tx:attributes>
        <tx:method name=="*" propagation="REQUIRED" />
    </tx:attributes>
</tx:advice>
```

애노테이션 기반의 AOP 구현

개발 단계

@Aspect 애노테이션을 이용하여 Advice 클래스를 구현
(Advice를 구현한 메서드와 PointCut 포함)

XML 설정 파일에는 <aop:aspectj-autoproxy /> 설정

애노테이션 기반의 AOP 구현

@Aspect 애노테이션을 이용하여 Advice 클래스를 구현

```
3  import org.aspectj.lang.ProceedingJoinPoint;
4  import org.aspectj.lang.annotation.Aspect;
5  import org.aspectj.lang.annotation.Pointcut;
6  import org.aspectj.lang.annotation.Around;
7  @Aspect
8  public class CommonAdvice {
9      @Pointcut("within(spring.study.aop.*)")
10     private void adviceMethod(){}
11
12     @Around("adviceMethod()")
13     public void around(ProceedingJoinPoint point) throws Throwable{
14         String methodName = point.getSignature().getName();
15         String targetName = point.getTarget().getClass().getName();
16         System.out.println("target 클래스명 : " + targetName + " 및 메소드명 : " + methodName);
17
18         long startTime = System.nanoTime();
19         System.out.println("[Log]핵심 로직 메소드 호출전 --> "+ methodName+" time check start");
20
21         Object obj = point.proceed();
22
23         long endTime = System.nanoTime();
24         System.out.println("[Log]핵심 로직 메소드 호출 후 --> " + methodName+" time check end");
25         System.out.println("[Log] "+ methodName+" 실행 소요 시간 "+(endTime - startTime)+"ns");
26
27         System.out.println(targetName + " 메서드 실행후 리턴된 데이터 : " + obj);
28     }
29 }
```

proxy대상의 실제 메소드 호출

애노테이션 기반의 AOP 구현

```
 8 @Aspect
 9 public class CommonAdvice {
10    @Pointcut("within(spring.study.aop.*)")
11    private void adviceMethod(){}
12
13    @Around("adviceMethod()")
14    public void around(ProceedingJoinPoint point) throws Throwable{
15       //... 중략 ...
16
17       String methodName = point.getSignature().getName();
```

@Aspect
: XML 설정 파일에 <aop:aspect> 태그 설정 없이도 클래스 자체가 어드바이스가 됨
@Pointcut
: 모든 클래스들의 메소드들을 포인트컷으로 설정하는 문법
  이 애노테이션이 적용된 메소드의 리턴 타입은 void여야 하며, 일반적으로 메소드 body에는 특별한 로직의 코드를 갖지 않으며, 이 메소드의 이름을 이용해서 다른 애노테이션에서 해서 활용하게 됨
@Around
: target 객체의 메소드 실행 전, 후 또는 예외 발생 시점에 모두 실행 할수 있게 설정

애노테이션 기반의 AOP 구현

XML 설정 파일에는 <aop:aspectj-autoproxy /> 설정

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4
5     xmlns:aop="http://www.springframework.org/schema/aop"
6
7     xsi:schemaLocation="http://www.springframework.org/schema/beans
8        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
9
10       http://www.springframework.org/schema/aop
11       http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
12
13    <aop:aspectj-autoproxy />
14
15    <bean id="common" class="spring.study.aop.CommonAdvice" />
16
17    <bean id="data" class="spring.study.aop.DataImpl"></bean>
18
19 </beans>
```

<aop:aspectj-autoproxy />
이 태그를 사용하게 되면 @Aspect 애노테이션이 적용된 빈 객체를 Aspect로 사용하게 됨

XML 기반 : 애노테이션 기반 비교해 보기

Advice

개발 방법 1

```
public void afterThrowing(){
        System.out.println("메서드 실행 중 예외 발생, 예외”);
}
```

개발 방법 2

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

설정 파일

```
<aop:aspect id="commonAspect" ref="common">
        <aop:pointcut   id=" exceptionPush" expression="within(spring.study.aop.*)"/>

         <!– 개발 방법 1 -->
        <aop: after-throwing pointcut-ref="exceptionPush" method="afterThrowing”/>

        <!– 개발 방법 2 -->
        <aop: after-throwing pointcut-ref=”exceptionPush” method="afterThrowing”
                                throwing=”ex”/>

</aop:aspect>
```

XML 기반 : 애노테이션 기반 비교해 보기

Advice

*** 개발 방법 1 - Advice 대상 객체의 메소드가 리턴한 값이 없을 경우 설정하는 메소드**

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

*** 개발 방법 2 - Advice 대상 객체의 메소드가 리턴한 값이 있을 경우 설정하는 메소드**

```
@ AfterThrowing (pointcut=" adviceMethod ()", throwing="ex")
public void afterThrowing (Throwable ex){
        System.out.println("메서드 실행 중 예외 발생, 예외=" + ex.getMessage());
}
```

설정 파일

*** 개발방법 1 -** @Aspect가 선언된 클래스에 @AfterThrowing("adviceMethod()") 애노테이션을 설정하게 되면 target 클래스의 메소드가 예외 발생시 afterThrowing() 메소드는 자동 실행됨

*** 개발 방법 2 –** target 객체의 메소드가 발생시킨 예외 객체를 받기 위한 파라미터를 throwing 속성값으로 지정하게 됩니다. 이 때 어드바이스 클래스의 해당 메소드 파라미터 명과 동일해야 함