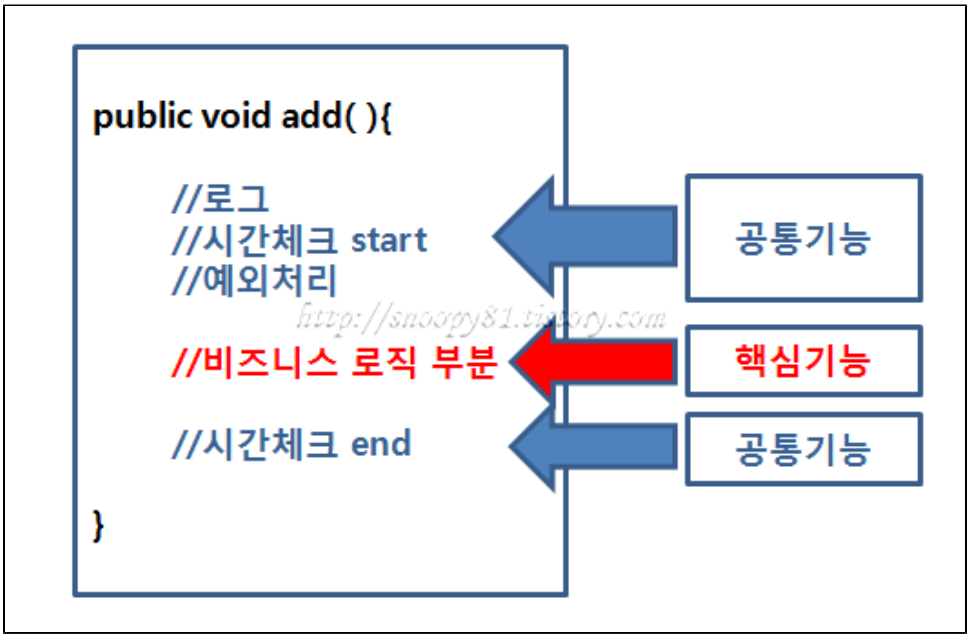
Day 07 (2019-01-28)
1. AOP

(1) AOP 개념

- AOP는 Aspect-Oriented Programming의 약자로 관점지향 프로그래밍을 뜻함
- OOP에선 객체를 기반으로하여 객체와 객체간의 상호작용을 중심사로 프로그래밍 하듯이 AOP에서는 Application을 종단 관심사(핵심 로직)와 횡단 관심사(공통 로직)로 분리하여
  주요 비즈니스 로직은 **종단 관심사**로 분류하고 Application의 전반적인 공통기능은 **횡단 관심사**로 분류



- 횡단 관심(Cross Cutting Concern)의 대표적인 예
  - Logging
  - 권한 검사
  - Transaction
  - 실행시간 측정
  - 대부분의 Application에서 필수적으로 요구하는 사항들

(2) 용어 정리

1) AOP 용어

AOP의 표준화된 용어나 그 설명은 매우 직관적이지 않습니다. 직접 설정해보면서 각각이 의미하는 것을 이해하는게 중요합니당

용어	표준 정의 설명	풀어쓴 것
Aspect(관점)	여러 클래스에 걸친 관심사가 모듈화 된 것	포인트 컷 + 어드바이스 어떤 포인트 컷(메소드)에 어떤 어드바이스(공통 관심 모듈)를 적용할 지

Join point(조인 포인트)	메소드에 파라미터 바인딩, 메소드의 실행이나 예외 처리 같이 프로그램이 실행되는 중의 특정 지점	언제(When) 횡단 관심 모듈을 삽입할 지 정의 함 Spring AOP에서 조인포인트는 항상 메소드 실행을 의미한다 필드값의 변경 등이 필요한 경우엔 Spring AOP가 아니느 AspectJ 처럼 다양한 JoinPoint를 지원하는 AOP 프레임워크가 필요하다
Advice (어드바이스)	특정 조인 포인트에서 관짐이 취하는 행동으로 여러 어드바이스 타입이 있다	핵심 관심 모듈에 삽입 될횡단 관심 모듈 자체 (What)
Pointcut (포인트 컷)	조인포인트를 매칭하는 것. 어드바이스는 포인트 컷 표현식과 연결되며 포인트 컷이 매치한 조인포인트에서 실행된다	어느 부분(Where)에 횡단 관심사 모듈을 삽입할 것인지 정의 [제어자, 반환타입, 패키지, 클래스, 메소드 이름, 파라미터, 예외]등의 표현식을 통해 정의함 → 특정 조건을 나타내는 표현식에 의해 필터링 된 조인포인트
Introduction (인트로덕션)	타입을 대신하여 메소드나 필드를 추가적으로 선언한다.	
Target object(대상 객체)	하나 이상의 관점으로 어드바이스 된 객체	
AOP Proxy (AOP 프록시)	AOP를 위해 생성된 객체로 Spring의 AOP 프록시는 JDK 다이나믹 프록시 혹은 CGLIB 프록시가 된다	
Weaving (위빙)	어드바이스 된 객체를 관점과 연결하는 것. 컴파일 시점, 로드 시점, 런타임 시점에 수행 될 수 있다	Advice가 실제로 적용되는 행위

2) Advice type 종류

Advice는 공통 처리 로직을 담은 횡단 관심 모듈 자체이며, 아래의 5가지 타입에 따른 Advice가 존재합니다

용어	표준 정의 설명	풀어쓴 것
Before	조인포인트 이전에 실행되는 어드바이스	메소드 실행 이전에 삽입되는 횡단 관심 모듈
After returning	조인포인트가 정상적으로 완료 후 종료되었을 때	메소드가 리턴 된 후에 삽입되는 횡단 관심 모듈
After throwing	조인포인트 실행 중 예외가 던져져 비정상 종료되었을 때	메소드에서 예외가 발생한 후 삽입되는 횡단 관심 모듈

After (finally)	조인포인트가 정상 종료이던 예외가 발생하던 모두가 끝난 후에	메소드의 문맥이 끝난 후 삽입되는 횡단 관심 모듈
Around	조인포인트를 둘러 싼 어드바이스.	메소드의 실행 전후를 기점으로 삽입되는 횡단 관심 모듈

(3) Weaving 방식

1) Compile 타임에 Weaving (CTW: CompileTimeWeaving)

aspectjweaver 라이브러리를 추가 후 build 시 별도의 플러그인을 통해 설정하여 적용가능하다

컴파일 시점에 대상 객체의 byte code를 조작하여 aspect를 삽입한다

2) Class loading 시에 Weaving (LTW: LoadTimeWeaving)

aspectjwearver 라이브러리를 추가 후 VM Option에 아래와 같은 java agent 관련 argument를 추가하여 적용 가능하다

-javaagent:C:₩Users₩taesu.m2₩repository₩org₩springframework₩spring-instrument₩5.0.7.RELEASE₩spring-instrument-5.0.7.RELEASE.jar
-javaagent:C:₩Users₩taesu.m2₩repository₩org₩aspectj₩aspectjweaver₩1.8.13₩aspectjweaver-1.8.13.jar

클래스 로딩 시점에 대상 객체의 byte code를 조작하여 aspect를 삽입한다

3) Run-time에 Weaving

위의 두 방식은 개발자가 작성하여 컴파일 된 bytecode가 직접적으로 변경되는 방식인 반면
Run-time에 weaving 하는 방식은 대상 객체에 대한 프록시를 생성하여 처리한다.

크게 JDK Dynamic Proxy를 이용하는 방법과 CGLIB Proxy를 이용하는 방법이 있다

※ Spring의 선언적 트랜잭션 미리 맛보기

아래 코드의 동작은?

```
////////////////////// BookController.java //////////////////////////
....
this.bookService.nonTransactionalMethod();
....


//////////////////////      BookService.java      ////////////////////////
@Service
@AllArgsConstructor
public class BookService {
    private BookRepository bookRepository;

        @Transactional
    public void nonTransactionalMethod() {
        //..do something
        this.transactionalMethodWillBeCommited();
        this.transactionalMethodWillBeFailed();
    }

    public void transactionalMethodWillBeCommited() {
        List<Book> books = bookRepository.findAll();
        books.forEach(Book::appendDateToAuthor);

        saveBooks(books);
    }

    public void transactionalMethodWillBeFailed() {
        List<Book> books = bookRepository.findAll();
        books.forEach(Book::appendDateToName);

        saveBooks(books);
        if (books.size() > 0) {
            throw new IllegalStateException(" ");
        }
    }

}
```

아래 코드의 동작은?

```
////////////////////// BookController.java //////////////////////////
....
this.bookService.nonTransactionalMethod();
....


//////////////////////      BookService.java      ////////////////////////
@Service
@AllArgsConstructor
public class BookService {
    private BookRepository bookRepository;

    public void nonTransactionalMethod() {
        //..do something
        this.transactionalMethodWillBeCommited();
        this.transactionalMethodWillBeFailed();
    }

    @Transactional
    public void transactionalMethodWillBeCommited() {
        List<Book> books = bookRepository.findAll();
        books.forEach(Book::appendDateToAuthor);

        saveBooks(books);
    }

    @Transactional
    public void transactionalMethodWillBeFailed() {
        List<Book> books = bookRepository.findAll();
        books.forEach(Book::appendDateToName);

        saveBooks(books);
        if (books.size() > 0) {
            throw new IllegalStateException(" ");
        }
    }

}
```

2. Spring AOP

(1) Spring AOP 특징

Spring은 자체적으로 Proxy 기반의 AOP를 지원하므로 메소드 호출에 대한 Join Point를 지원한다

따라서 필드값 변경, 현재 클래스의 메소드 실행 중 AOP에 대상이 되는 메소드 실행은 제대로 처리될 수 없으며
이러한 동작이 정상적으로 처리되도록 하려면 AspectJ와 같이 다양한 Join Point를 지원하는 프레임워크가 필요하다

스프링에서 제공하는 AOP 설정 방식

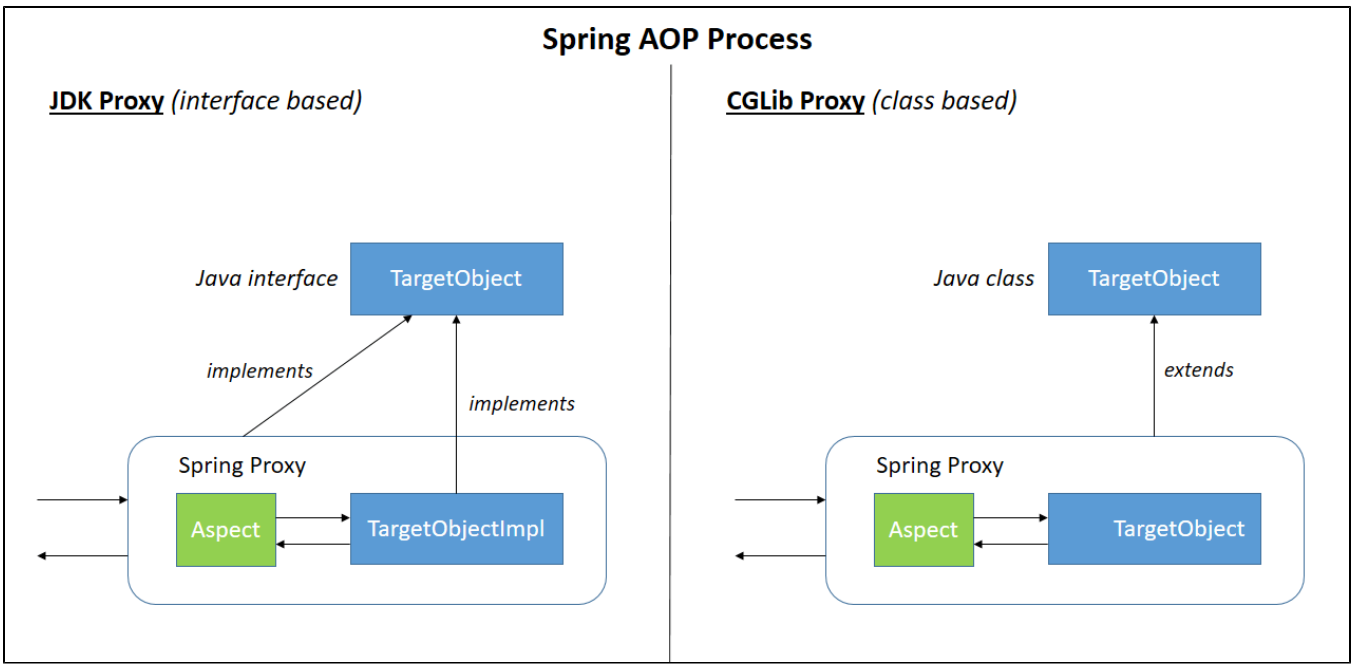
- XML 스키마 기반의 PJOJ 클래스를 이용한 AOP 설정
- AspectJ에서 정의한 @Aspect 어노테이션 기반 AOP 설정
    - AspectJ에서 정의한 설정 방식을 사용하는 것이지 AspectJ에서 지원하는 Join Point를 사용하는 것이 아님을 주의
- Spring API를 이용한 AOP 구현

(2) Spring AOP의 구현 종류

Spring은 Aspect의 적용 대상이 되는 **객체**에 대한 프록시를 만들어 제공한다

즉 Client(Aspect의 적용 대상이 되는 객체를 의존하는 객체)에서 대상 객체를 사용하려고 할 때
대상 객체의 원본에 접근하는것이 아니라 대상 객체에 대한 프록시에 접근합니다.

프록시는 대상 객체에 대한 메소드 실행을 위임받아 필요한 Advice를 실행하는데
프록시 객체의 생성 방식은 대상 객체가 interface를 구현하는지의 여부에 따라 다릅니다



1) JDK Dynamic Proxy를 이용한 구현

- java.lang.reflect.Proxy를 통해 프록시 객체를 생성한다.
- Interface 기반이다.
- 인터페이스를 기반으로 프록시 객체를 생성하므로 반드시 대상 객체는 인터페이스를 구현해야하며
  Join Point도 인터페이스에 정의되어있는 메소드가 되어야 한다

2) CGLIB Proxy를 이용한 구현

- (옵션에 따라 다르지만) 대상객체가 인터페이스를 구현하지 않고있다면 CGLIB를 통해 대상 객체를 상속받아 프록시를 구현한다
- Class 기반이다
- 따라서 대상객체는 반드시 final 클래스가 아니어야 한다

※ Spring과 Spring boot의 AOP Proxy 구현체에 대한 선택

Spring framework에서는 AOP 설정 시 proxy-target-class 옵션이 false인 경우 AOP 대상 객체가 인터페이스를 구현한다면 JDK Dynamic Proxy를 사용하며
그렇지 않은 경우 Classpath에 CGLIB가 존재할 때 CGLIB Proxy를 사용합니다.
(Classpath에 없는경우 Run-time exception이 발생함)

반면 proxy-target-class 옵션이 true인 경우 무조건 CGLIB Proxy를 사용합니다

Spring framework의 proxy-target-class의 default 옵션은 false입니다 (4.3.22 버전 기준)

하지만 Spring boot 2.0 이상의 버전인 경우 proxy-target-class의 default 옵션은 true로 되어있어 CGLIB를 기본으로 사용하게 되어있습니다

이는 CGLIB의 경우 예외가 발생할 가능성이 적고 성능상의 약간의 이득이 있기 때문이라고 합니다(http://wonwoo.ml/index.php/post/1708)

따라서 **Spring 기반의 설정을 진행하는 경우에 proxy-target-class 옵션을 true**로 설정해주는것이 좋습니다

(3) Spring AOP 설정 및 예제

Spring AOP를 사용하기 위해서 아래와 같은 Dependency가 필요

pom.xml

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

우리는 @Aspect 기반의 설정을 진행할 것이므로 아래와 같이 annotation-config 설정을 선언하고
@Aspect 설정이 되어있는 컴포넌트를 Bean으로 등록 하기 위한 component-scan 설정을 선언한다

그 후 Spring AOP의 프록시기반 설정을 위해 aspectj-autoproxy 설정을 선언한다

spring-context.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: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.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

   <context:annotation-config/>
   <context:component-scan base-package="io.crscube.semina.day06" />
   <aop:aspectj-autoproxy />
</beans>
```

아래는 Aspect가 설정 된 Component입니다. Component scan이 될 수 있도록 io.crscube.semina.day06 패키지 하위에 존재하며 @Component 어노테이션이 선언되어있음을 확인.

Aspect Component

```
package io.crscube.semina.day06.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect            //AOP   Aspect( + )
@Component
public class ApplicationAspect {
   //AOP
   //     Spring AOP

   @Before("execution(* io.crscube.semina.day06.component.impl.*.*(..))")   //AOP   Pointcut  (value    )
   public void onBefore() {                                                                                    //
day06.component.impl
      //AOP   Advice
      //
      System.out.println("Hello onBefore");
   }
}
```

AOP가 적용 될 대상객체로 인터페이스를 구현하고 있으며 현재 proxy-target-class 옵션이 false로 되어있다. 따라서 **JDK Dynamic Proxy에 의해 프록시 객체가 생성**된다

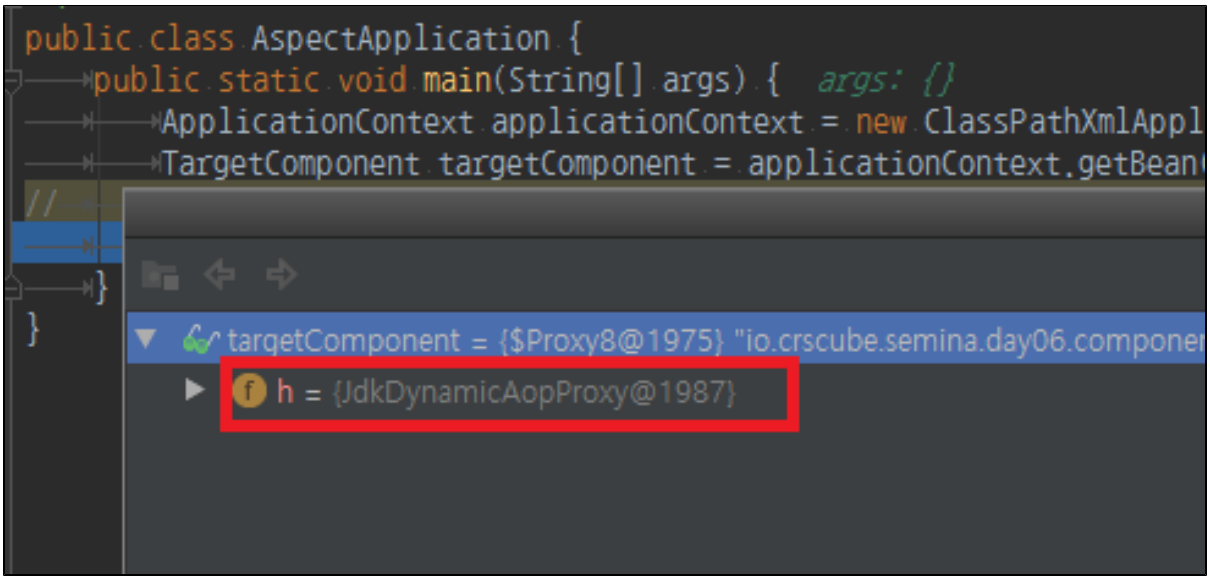
대상객체

```
///////////    TargetComponent.java    //////////////
ppackage io.crscube.semina.day06.component;

public interface TargetComponent {
   void callMethod();
}


////////////// BeforeTargetComponent.java ////////////
package io.crscube.semina.day06.component.impl;

import io.crscube.semina.day06.component.TargetComponent;
import org.springframework.stereotype.Component;

@Component("targetComponent")
public class BeforeTargetComponent implements TargetComponent{
   public void callMethod() {
      System.out.println("callMethod()");
   }
}
```

Aspect가 설정 된 객체를 통해 포인트 컷 메소드를 실행하도록 하여 디버깅을 하면 JdkDynamicAopProxy 객체가 반환되었음을 알 수 있다.

JDK Dynamic Proxy 객체 생성 테스트

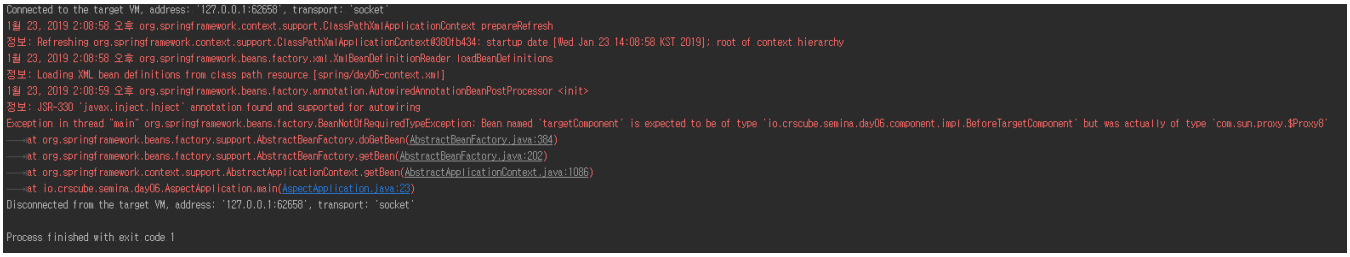
```
public class AspectApplication {
   public static void main(String[] args) {
      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("classpath:spring/day06-
context.xml");
      TargetComponent targetComponent = applicationContext.getBean("targetComponent", TargetComponent.class);
      Objects.requireNonNull(targetComponent).callMethod();
   }
}
```



이 상태에서 아래와 같이 반환 받을 Bean의 타입을 BeforeTargetComponent로 요구하면 Exception이 발생한다

JDK Dynamic Proxy 객체 생성 테스트

```
public class AspectApplication {
   public static void main(String[] args) {
      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("classpath:spring/day06-
context.xml");
      BeforeTargetComponent targetComponent = applicationContext.getBean("targetComponent",
BeforeTargetComponent.class);
      Objects.requireNonNull(targetComponent).callMethod();
   }
}
```



반환되는 객체는 인터페이스를 구현하는 JDK Dynamic Proxy 객체이다.
따라서 프록시 객체는 TargetComponent 인터페이스의 구현체이지만 우리가 요구하는 **BeforeTargetComponent를 상속하지는 않으므로** Type 관련 에러가 발생한다.

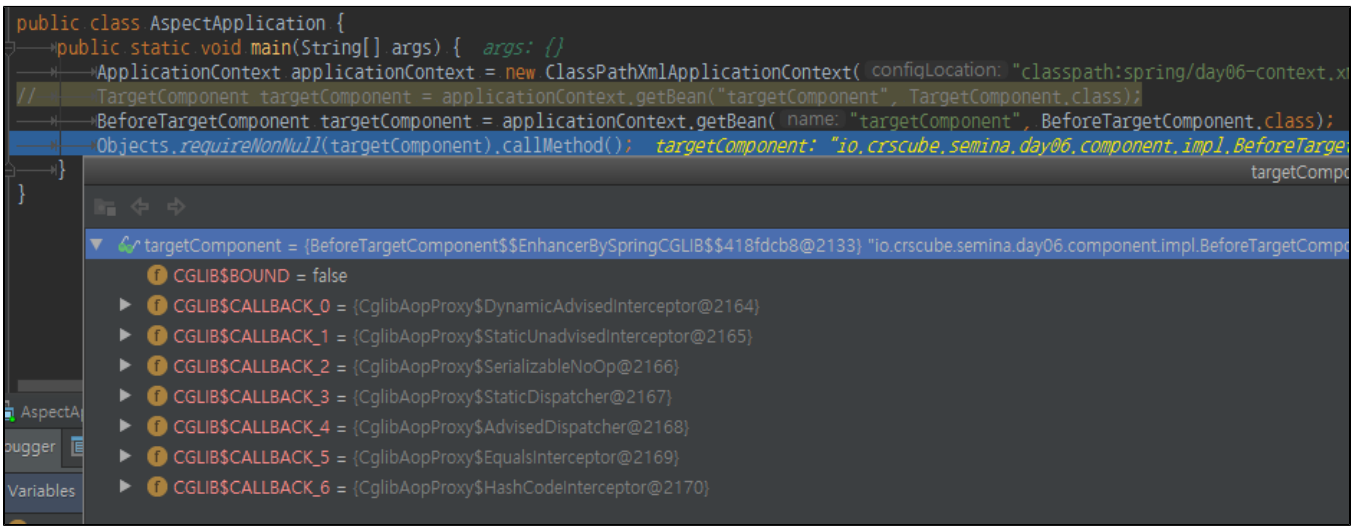
정상적으로 처리가 되도록 하려면?

① 인터페이스를 구현하는 관계를 끊는다

이 상태에서 BeforeTargetComponent가 구현하는 인터페이스 관계를 끊어버린후 다시 실행하면 정상적으로 처리된다.

TargetComponent 인터페이스 구현을 제거한 후

```
@Component("targetComponent")
public class BeforeTargetComponent {
   public void callMethod() {
      System.out.println("callMethod()");
   }
}
```



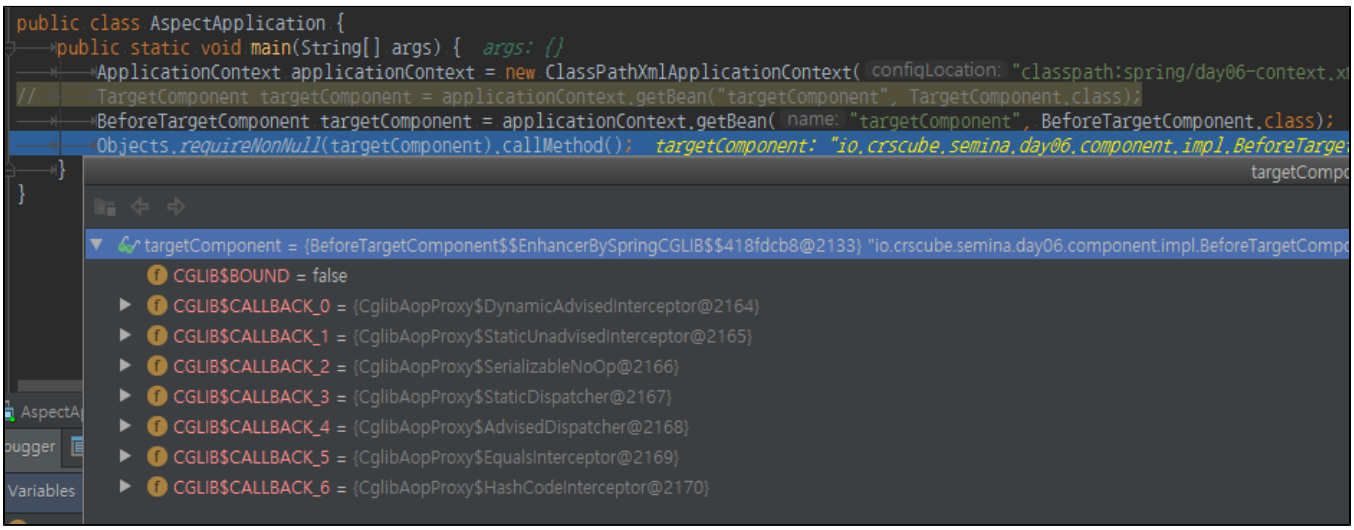
인터페이스를 구현하지 않기 때문에 CGLIB 기반으로 프록시가 생성되었고 해당 프록시는 대상객체(BeforeTargetComponent)를 상속하기 때문에 Type 관련 에러가 발생하지 않는 것.

② proxy-target-class 옵션을 true로 설정한다

인터펭이스는 상속하도록 두되 proxy-target-class 옵션을 true로 설정하여 무조건 **CGLIB 프록시를 사용하도록** 처리한다.

proxy-target-class true로 설정

```
<?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.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

   <context:annotation-config/>
   <context:component-scan base-package="io.crscube.semina.day06" />
   <aop:aspectj-autoproxy proxy-target-class="true" />
</beans>
```



마찬가지로 BeforeTargetComponent 타입으로 형 변환도 가능하고 TargetComponent 타입으로도 형 변환이 가능하다
또한 인터페이스를 구현하던 구현하지 않던 CGLIB 프록시 객체가 만들어짐을 확인할 수 있다.

3. 참고

(1) 포인트컷 표현식

```
execution(* io.crscube.semina.day06.component.impl.*.*(..))
```

포인트컷은 지정자와 타겟 명세로 나뉜다

"**execution**"은 지정자, "**(* io.crscube.semina.day06.component.impl.*.*(..))**"는 타겟 명세

※ 지정자 종류

지정자	설명
args()	• 메소드의 인자가 타겟 명세에 포함된 타입일 경우 • ex) args(java.io.Serializable) : 하나의 파라미터를 갖고, 그 인자가 Serializable 타입인 모든 메소드
@args	• 메소드의 인자가 타겟 명세에 포함된 어노테이션 타입을 갖는 경우 • ex) @args(com.blogcode.session.User) : 하나의 파라미터를 갖고, 그 인자의 타입이 @User 어노테이션을 갖는 모든 메소드 (@User User user 같이 인자 선언된 메소드)

execution()	• 접근제한자, 리턴타입, 인자타입, 클래스/인터페이스, 메소드명, 파라미터타입, 예외타입 등을 전부 조합가능한 가장 세심한 지정자 • 이전 예제와 같이 풀패키지에 메소드명까지 직접 지정할 수도 있으며, 아래와 같이 특정 타입내의 모든 메소드를 지정할 수도 있다. • ex) execution(* com.blogcode.service.AccountService.*(..) : AccountService 인터페이스의 모든 메소드
within()	• execution 지정자에서 클래스/인터페이스까지만 적용된 경우 • 즉, 클래스 혹은 인터페이스 단위까지만 범위 지정이 가능하다. • ex) within(com.blogcode.service.*) : service 패키지 아래의 클래스와 인터페이스가 가진 모든 메소드 • ex) within(com.blogcode.service..*) : service 아래의 모든 *하위패키지까지** 포함한 클래스와 인터페이스가 가진 메소드
@within()	• 주어진 어노테이션을 사용하는 타입으로 선언된 메소드
this()	• 타겟 메소드가 지정된 빈 타입의 인스턴스인 경우
target()	• this와 유사하지만 빈 타입이 아닌 타입의 인스턴스인 경우
@target()	• 타겟 메소드를 실행하는 객체의 클래스가 타겟 명세에 지정된 타입의 어노테이션이 있는 경우
@annotation()	• 타겟 메소드에 특정 어노테이션이 지정된 경우 • ex) @annotation(org.springframework.transaction.annotation.Transactional) : Transactional 어노테이션이 지정된 메소드 전부

※ 타겟명세에서 사용하는 와일드 카드

와일드 카드	설명
*	• All을 나타냄
..	• 0개 이상의 의미를 나타냄

(2) AOP 주의사항

→ AOP는 최대한 단순하고 명시적일수록 좋으며, AOP에 비즈니스로직을 담지 말아야 함을 잊지 말 것

(화려한 포인트컷은 아무리 잘 짜놔도 테스트가 불가하다)

특정 Exception이 발생 했을 때 최대 5번 재수행 하도록 처리하는 AOP를 작성해보기

실패시 5번 재수행 가능한 Advice의 적용 대상객체

```
@Component
public class UserMapper {
   private int callCount = 1;

   public String gerUserName() {
      System.out.println("getUserName call count is [" + callCount + "]");
      if(callCount >= 5) {
         return "Hello lee";
      }
      callCount++;

      throw new RedoableException("");
   }
}
```

특정 Exception 예제

```
public class RedoableException extends NestedRuntimeException {
   public RedoableException(String msg) {
      super(msg);
   }
}
```

메인 실행코드

```
public class RedoApplication {
   public static void main(String[] args) {
      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("classpath:spring/day06-
context.xml");
      UserMapper userMapper = applicationContext.getBean("userMapper", UserMapper.class);
      System.out.println(userMapper.gerUserName());
   }
}
```

실행 결과

```
1월 23, 2019 4:01:17 오후 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
정보: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@38Ofb434: startup date [Wed Jan 23 16:01:16 KST 2019]; root of context hierarchy
1월 23, 2019 4:01:17 오후 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
정보: Loading XML bean definitions from class path resource [spring/day06-context.xml]
1월 23, 2019 4:01:17 오후 org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor <init>
정보: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
getUserName call count is [1]
getUserName call count is [2]
getUserName call count is [3]
getUserName call count is [4]
getUserName call count is [5]
Hello lee
Disconnected from the target VM, address: '127.0.0.1:57644', transport: 'socket'

Process finished with exit code 0
```