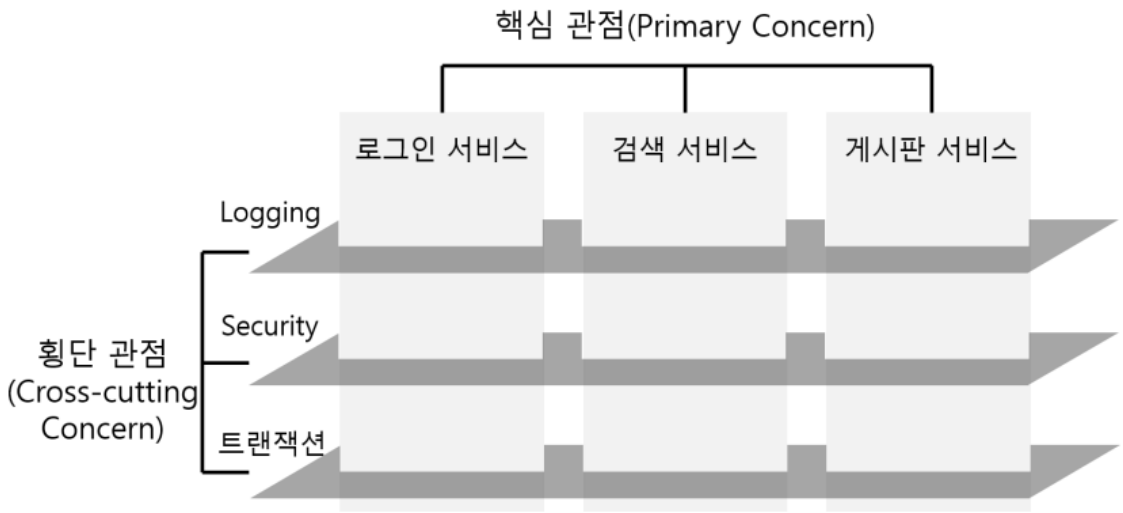
Chap06. AOP

1. AOP

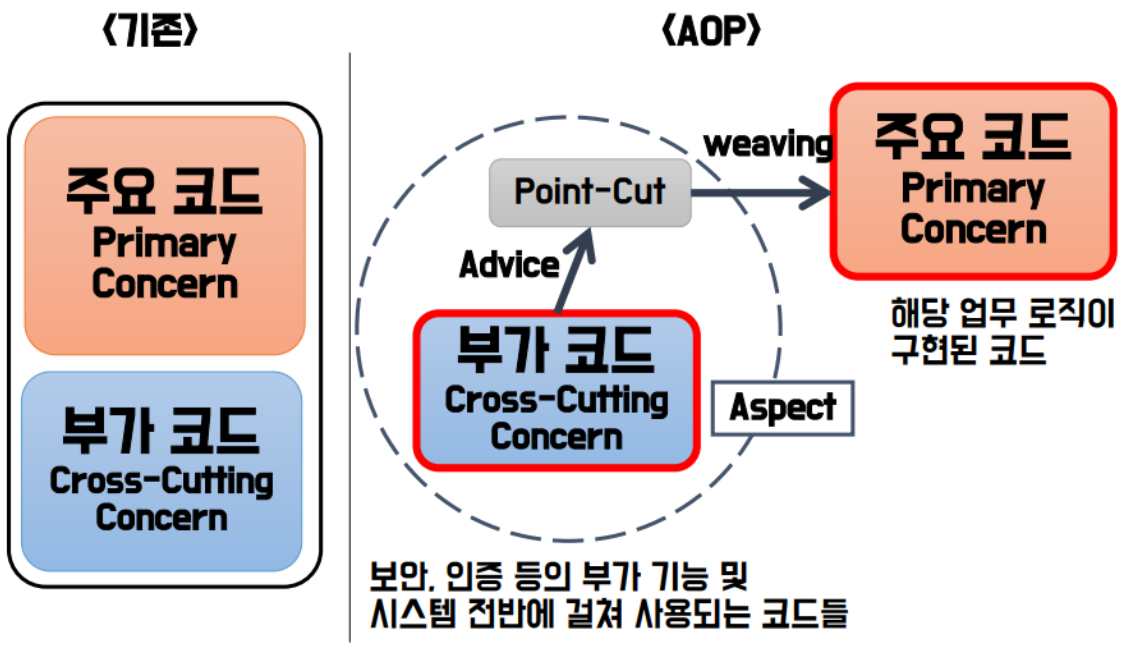
1-1. AOP란?

> 💡 AOP는 관점 지향 프로그래밍(Aspect Oriented Programming)의 약자이다. 중복되는 공통 코드를 분리하고 코드 실행 전이나 후의 시점에 해당 코드를 삽입함으로써 소스 코드의 중복을 줄이고, 필요할 때마다 가져다 쓸 수 있게 객체화하는 기술을 말한다.



1-2. AOP 핵심 용어

용어	설명
Aspect	핵심 비즈니스 로직과는 별도로 수행되는 횡단 관심사를 말한다.
Advice	Aspect의 기능 자체를 말한다.
Join point	Advice가 적용될 수 있는 위치를 말한다.
Pointcut	Join point 중에서 Advice가 적용될 가능성이 있는 부분을 선별한 것을 말한다.
Weaving	Advice를 핵심 비즈니스 로직에 적용하는 것을 말한다.



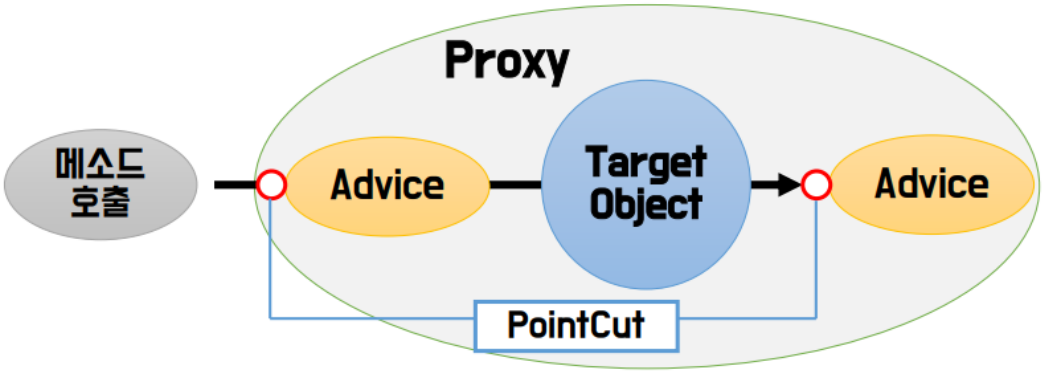
1-3. Adivce의 종류

종류	설명
Before	대상 메소드가 실행되기 이전에 실행되는 어드바이스
After-returning	대상 메소드가 정상적으로 실행된 이후에 실행되는 어드바이스
After-throwing	예외가 발생했을 때 실행되는 어드바이스
After	대상 메소드가 실행된 이후에(정상, 예외 관계없이) 실행되는 어드바이스
Around	대상 메소드 실행 전/후에 적용되는 어드바이스

1-4. Spring AOP

스프링 프레임워크에서 제공하는 AOP는 다음과 같은 특징을 가진다.

- **프록시 기반의 AOP 구현체** : 대상 객체(Target Object)에 대한 프록시를 만들어 제공하며, 타겟을 감싸는 프록시는 서버 Runtime 시에 생성된다.
- **메서드 조인 포인트만 제공** : 핵심기능(대상 객체)의 메소드가 호출되는 런타임 시점에만 부가기능(어드바이스)을 적용할 수 있다.



2. Spring AOP 구현하기

2-1. 로직을 포함하는 코드 작성

1. MemberDTO 클래스 생성

```
@Getter @Setter @ToString
@AllArgsConstructor
public class MemberDTO {

    private Long id;
    private String name;
}
```

2. MemberDAO 클래스 생성

```
@Repository
public class MemberDAO {

    private final Map<Long, MemberDTO> memberMap;

    public MemberDAO(){
        memberMap = new HashMap<>();
        memberMap.put(1L, new MemberDTO(1L, "유관순"));
        memberMap.put(2L, new MemberDTO(2L, "홍길동"));
    }

    public Map<Long, MemberDTO> selectMembers(){

        return memberMap;
    };

    public MemberDTO selectMember(Long id) {

        MemberDTO returnMember = memberMap.get(id);

        if(returnMember == null) throw new RuntimeException("해당하는 id의 회원이 없습니다.");

        return returnMember;
    }
}
```

3. MemberService 클래스 생성

```
@Service
public class MemberService {

    private final MemberDAO memberDAO;

    public MemberService(MemberDAO memberDAO) {
        this.memberDAO = memberDAO;
    }

    public Map<Long, MemberDTO> selectMembers(){
        System.out.println("selectMembers 메소드 실행");
        return memberDAO.selectMembers();
    }

    public MemberDTO selectMember(Long id) {
```

```
        System.out.println("selectMember 메소드 실행");
        return memberDAO.selectMember(id);
    }
}
```

4. Application 클래스 생성

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

        ApplicationContext context
          = new AnnotationConfigApplicationContext("com.ohgiraffers.section01.aop");

        MemberService memberService = context.getBean("memberService", MemberService.class);
        System.out.println("=============== selectMembers ===============");
        System.out.println(memberService.selectMembers());
        System.out.println("=============== selectMember ===============");
        System.out.println(memberService.selectMember(3L));

    }
}
```

▼ 실행 결과

```
=============== selectMembers ===============
{1=MemberDTO(id=1, name=유관순), 2=MemberDTO(id=2, name=홍길동)}
=============== selectMember ===============
Exception in thread "main" java.lang.RuntimeException: 해당하는 id의 회원이 없습니다.
...생략
```

2-2. 라이브러리 의존성 추가

`aspectjweaver` , `aspectjrt` 라이브러리가 있어야 AOP 기능이 동작할 수 있으므로 `build.gradle.kts` 파일에 추가한다.

```
dependencies {
  ...생략
  implementation 'org.aspectj:aspectjweaver:1.9.19'
  implementation 'org.aspectj:aspectjrt:1.9.19'
}
```

2-3. AutoProxy 설정

`ContextConfiguration` 빈 설정 파일을 생성한다. `aspectj`의 `autoProxy` 사용에 관한 설정을 해 주어야 advice가 동작한다. `proxyTargetClass=true` 설정은 `cglib`를 이용한 프록시를 생성하는 방식인데, Spring 3.2부터 스프링 프레임워크에 포함되어 별도 라이브러리 설정을 하지 않고 사용할 수 있다. 성능 면에서 더 우수하다.

```
@Configuration
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class ContextConfiguration {
}
```

2-4. Aspect 생성

`LoggingAspect` 클래스를 생성하고 빈 스캐닝을 통해 빈 등록을 한다.

`@Aspect` : ponitcut과 advice를 하나의 클래스 단위로 정의하기 위한 어노테이션이다.

```
@Aspect
@Component
public class LoggingAspect {}
```

2-4-1. Pointcut

`LoggingAspect` 클래스에 포인트 컷을 정의한다.

`@Pointcut` : 여러 조인 포인트를 매치하기 위해 지정한 표현식

```
@Pointcut("execution(* com.ohgiraffers.section01.aop.*Service.*(..))")
public void logPointcut() {}
```

▼ `execution` 설명

`execution`은 AOP에서 가장 많이 사용되는 포인트컷 표현식 중 하나이다. `execution` 표현식은 메서드 실행 시점에 일치하는 조인포인트를 정의하는 데 사용된다. `execution`표현식의 기본 구성은 다음과 같다.

```
execution([접근제한자패턴] [리턴타입패턴] [클래스이름패턴] [메서드이름패턴]([파라미터타입패턴]))
```

`com.example.*` 패키지 내의 클래스에서 반환값이 `void`인 메소드 중, 메소드명이 `"get*"`으로 시작하는 메소드를 포함하는 표현식은 다음과 같다.

```
execution(void com.example.*.*.get*(..))
```

- `void` : 리턴 타입 패턴으로 반환값이 `void`인 메소드를 나타낸다.
- `com.example.*.*` : 클래스 이름 패턴으로 `com.example` 패키지 내의 모든 클래스를 나타낸다.
- `get*` : 메소드 이름 패턴으로 `"get"`으로 시작하는 모든 메소드를 나타낸다.
- `..` : 파라미터 타입 패턴으로 모든 파라미터를 나타낸다.

`com.example` 패키지 내의 클래스에서 메소드명이 `"set*"`으로 시작하는 메소드 중, 인자로 `java.lang.String` 타입의 인자를 갖는 메소드를 포함하는 표현식은 다음과 같다.

```
execution(* com.example..set*(java.lang.String))
```

- `*` : 리턴타입 패턴으로 모든 반환값을 나타낸다.
- `com.example..` : 클래스 이름 패턴으로 `com.example` 패키지 내의 모든 클래스를 나타낸다.
- `set*` : 메소드 이름 패턴으로 `"set"`으로 시작하는 모든 메소드를 나타낸다.

- `java.lang.String` : 파라미터 타입 패턴으로 인자로 `java.lang.String` 타입 하나만을 나타낸다.

2-4-2. Before

`Before` 어드바이스는 대상 메소드가 실행되기 이전에 실행되는 어드바이스이다. 미리 작성한 포인트 컷을 설정한다.

`JoinPoint`는 포인트컷으로 패치한 실행 지점이다. 매개변수로 전달한 JoinPoint 객체는 현재 조인 포인트의 메소드명, 인수값 등의 자세한 정보를 엑세스 할 수 있다.

```
@Before("LoggingAspect.logPointcut()")
public void logBefore(JoinPoint joinPoint) {
    System.out.println("Before joinPoint.getTarget() " + joinPoint.getTarget());
    System.out.println("Before joinPoint.getSignature() " + joinPoint.getSignature());
    if(joinPoint.getArgs().length > 0){
        System.out.println("Before joinPoint.getArgs()[0] " + joinPoint.getArgs()[0]);
    }
}
```

▼ 실행 결과

```
=============== selectMembers ===============
Before joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@6ed3f258
Before joinPoint.getSignature() Map com.ohgiraffers.section01.aop.MemberService.selectMembers()
selectMembers 메소드 실행
{1=MemberDTO(id=1, name=유관순), 2=MemberDTO(id=2, name=홍길동)}
=============== selectMember ===============
Before joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@6ed3f258
Before joinPoint.getSignature()
MemberDTO com.ohgiraffers.section01.aop.MemberService.selectMember(Long)
Before joinPoint.getArgs()[0] 3
selectMember 메소드 실행
Exception in thread "main" java.lang.RuntimeException: 해당하는 id의 회원이 없습니다.
...생략
```

`MemberService` 클래스의 `selectMembers` 메소드와 `selectMember` 메소드가 실행 되기 전 `Before` 어드바이스의 실행 내용이 삽입 되어 동작하는 것을 확인할 수 있다.

2-4-3. After

`After` 어드바이스는 대상 메소드가 실행된 이후에(정상, 예외 관계없이) 실행되는 어드바이스이다. 미리 작성한 포인트 컷을 설정한다. 포인트 컷을 동일한 클래스 내에서 사용하는 것이면 클래스명은 생략 가능하다. 단, 패키지가 다르면 패키지를 포함한 클래스명을 기술해야 한다.

`Before` 어드바이스와 동일하게 매개변수로 JoinPoint 객체를 전달 받을 수 있다.

```
@After("logPointcut()")
public void logAfter(JoinPoint joinPoint) {
    System.out.println("After joinPoint.getTarget() " + joinPoint.getTarget());
    System.out.println("After joinPoint.getSignature() " + joinPoint.getSignature());
    if(joinPoint.getArgs().length > 0){
        System.out.println("After joinPoint.getArgs()[0] " + joinPoint.getArgs()[0]);
    }
}
```

▼ 실행 결과

```
=============== selectMembers ===============
Before joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@5443d039
Before joinPoint.getSignature() Map com.ohgiraffers.section01.aop.MemberService.selectMembers()
selectMembers 메소드 실행
After joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@5443d039
After joinPoint.getSignature() Map com.ohgiraffers.section01.aop.MemberService.selectMembers()
{1=MemberDTO(id=1, name=유관순), 2=MemberDTO(id=2, name=홍길동)}
=============== selectMember ===============
Before joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@5443d039
Before joinPoint.getSignature()
MemberDTO com.ohgiraffers.section01.aop.MemberService.selectMember(Long)
Before joinPoint.getArgs()[0] 3
selectMember 메소드 실행
After joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@5443d039
After joinPoint.getSignature()
MemberDTO com.ohgiraffers.section01.aop.MemberService.selectMember(Long)
After joinPoint.getArgs()[0] 3
Exception in thread "main" java.lang.RuntimeException: 해당하는 id의 회원이 없습니다.
...생략
```

`MemberService` 클래스의 `selectMembers` 메소드와 `selectMember` 메소드가 실행 된 후에 `After` 어드바이스의 실행 내용이 삽입 되어 동작하는 것을 확인할 수 있다. `Exception` 발생 여부와 무관하게 항상 실행된다.

2-4-4. AfterReturning

`AfterReturning` 어드바이스는 대상 메소드가 정상적으로 실행된 이후에 실행되는 어드바이스이다. 미리 작성한 포인트 컷을 설정한다.

`returning` 속성은 리턴값으로 받아올 오브젝트의 매개변수 이름과 동일해야 한다. 또한 joinPoint는 반드시 첫 번째 매개변수로 선언해야 한다. 이 어드바이스에서는 반환 값을 가공할 수도 있다.

```
@AfterReturning(pointcut="logPointcut()", returning="result")
public void logAfterReturning(JoinPoint joinPoint, Object result) {
    System.out.println("After Returning result " + result);
    /* 리턴할 결과값을 변경해 줄 수 도 있다. */
    if(result != null && result instanceof Map) {
        ((Map<Long, MemberDTO>) result).put(100L, new MemberDTO(100L, "반환 값 가공"));
    }
}
```

▼ 실행 결과

```
=============== selectMembers ===============
Before joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@2a62b5bc
Before joinPoint.getSignature() Map com.ohgiraffers.section01.aop.MemberService.selectMembers()
selectMembers 메소드 실행
After Returning result {1=MemberDTO(id=1, name=유관순), 2=MemberDTO(id=2, name=홍길동)}
After joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@2a62b5bc
After joinPoint.getSignature() Map com.ohgiraffers.section01.aop.MemberService.selectMembers()
{1=MemberDTO(id=1, name=유관순), 2=MemberDTO(id=2, name=홍길동),
100=MemberDTO(id=100, name=반환 값 가공)}
=============== selectMember ===============
Before joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@2a62b5bc
Before joinPoint.getSignature()
MemberDTO com.ohgiraffers.section01.aop.MemberService.selectMember(Long)
Before joinPoint.getArgs()[0] 3
selectMember 메소드 실행
```

```
After joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@2a62b5bc
After joinPoint.getSignature()
MemberDTO com.ohgiraffers.section01.aop.MemberService.selectMember(Long)
After joinPoint.getArgs()[0] 3
Exception in thread "main" java.lang.RuntimeException: 해당하는 id의 회원이 없습니다.
...생략
```

`MemberService` 클래스의 `selectMembers` 메소드가 실행 된 후에 `AfterReturning` 어드바이스의 실행 내용이 삽입 되어 동작하는 것을 확인할 수 있다. `Exception` 이 발생한 `selectMember` 메소드에서는 동작하지 않는다.

2-4-5. AfterThrowing

`AfterThrowing` 어드바이스는 예외가 발생했을 때 실행되는 어드바이스이다. 미리 작성한 포인트 컷을 설정한다.

throwing 속성의 이름과 매개변수의 이름이 동일해야 한다. 이 어드바이스에서는 `Exception` 에 따른 처리를 작성할 수 있다.

```
@AfterThrowing(pointcut="logPointcut()", throwing="exception")
public void logAfterThrowing(Throwable exception) {
    System.out.println("After Throwing exception " + exception);
}
```

▼ 실행 결과

```
=============== selectMembers ===============
Before joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@24111ef1
Before joinPoint.getSignature() Map com.ohgiraffers.section01.aop.MemberService.selectMembers()
selectMembers 메소드 실행
After Returning result {1=MemberDTO(id=1, name=유관순), 2=MemberDTO(id=2, name=홍길동)}
After joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@24111ef1
After joinPoint.getSignature() Map com.ohgiraffers.section01.aop.MemberService.selectMembers()
{1=MemberDTO(id=1, name=유관순), 2=MemberDTO(id=2, name=홍길동),
100=MemberDTO(id=100, name=반환 값 가공)}
=============== selectMember ===============
Before joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@24111ef1
Before joinPoint.getSignature()
MemberDTO com.ohgiraffers.section01.aop.MemberService.selectMember(Long)
Before joinPoint.getArgs()[0] 3
selectMember 메소드 실행
After Throwing exception java.lang.RuntimeException: 해당하는 id의 회원이 없습니다.
After joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@24111ef1
After joinPoint.getSignature()
MemberDTO com.ohgiraffers.section01.aop.MemberService.selectMember(Long)
After joinPoint.getArgs()[0] 3
Exception in thread "main" java.lang.RuntimeException: 해당하는 id의 회원이 없습니다.
...생략
```

`MemberService` 클래스의 `selectMember` 메소드가 실행 된 후에 `AfterThrowing` 어드바이스의 실행 내용이 삽입 되어 동작하는 것을 확인할 수 있다. `Exception` 이 발생하지 않은 `selectMembers` 메소드에서는 동작하지 않는다.

2-4-6. Around

`Around` 어드바이스는 대상 메소드 실행 전/후에 적용되는 어드바이스이다. 미리 작성한 포인트 컷을 설정한다.

Around Advice는 가장 강력한 어드바이스이다. 이 어드바이스는 조인포인트를 완전히 장악하기 때문에 앞에 살펴 본 어드바이스 모두 Around 어드바이스로 조합할 수 있다.

AroundAdvice의 조인포인트 매개변수는 ProceedingJoinPoint로 고정되어 있다. JoinPoint의 하위 인터페이스로 원본 조인포인트의 진행 시점을 제어할 수 있다.

조인포인트 진행하는 호출을 잊는 경우가 자주 발생하기 때문에 주의해야 하며 최소한의 요건을 충족하면서도 가장 기능이 약한 어드바이스를 쓰는게 바람직하다.

```
@Around("logPointcut()")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    System.out.println("Around Before " + joinPoint.getSignature().getName());
    /* 원본 조인포인트를 실행한다. */
    Object result = joinPoint.proceed();
    System.out.println("Around After " + joinPoint.getSignature().getName());
    /* 원본 조인포인트를 호출한 쪽 혹은 다른 어드바이스가 다시 실행할 수 있도록 반환한다. */
    return result;
}
```

▼ 실행 결과

```
=============== selectMembers ===============
Around Before selectMembers
Before joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@21a21c64
Before joinPoint.getSignature() Map com.ohgiraffers.section01.aop.MemberService.selectMembers()
selectMembers 메소드 실행
After Returning result {1=MemberDTO(id=1, name=유관순), 2=MemberDTO(id=2, name=홍길동)}
After joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@21a21c64
After joinPoint.getSignature() Map com.ohgiraffers.section01.aop.MemberService.selectMembers()
Around After selectMembers
{1=MemberDTO(id=1, name=유관순), 2=MemberDTO(id=2, name=홍길동),
100=MemberDTO(id=100, name=반환 값 가공)}
=============== selectMember ===============
Around Before selectMember
Before joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@21a21c64
Before joinPoint.getSignature()
MemberDTO com.ohgiraffers.section01.aop.MemberService.selectMember(Long)
Before joinPoint.getArgs()[0] 3
selectMember 메소드 실행
After Throwing exception java.lang.RuntimeException: 해당하는 id의 회원이 없습니다.
After joinPoint.getTarget() com.ohgiraffers.section01.aop.MemberService@21a21c64
After joinPoint.getSignature()
MemberDTO com.ohgiraffers.section01.aop.MemberService.selectMember(Long)
After joinPoint.getArgs()[0] 3
Exception in thread "main" java.lang.RuntimeException: 해당하는 id의 회원이 없습니다.
...생략
```

`MemberService` 클래스의 `selectMember` 메소드가 실행 된 후에 `AfterThrowing` 어드바이스의 실행 내용이 삽입되어 동작하는 것을 확인할 수 있다. `Exception` 이 발생하지 않은 `selectMembers` 메소드에서는 동작하지 않는다.

3. Reflection

> 💡 자바 리플렉션(Reflection)은 실행 중인 자바 프로그램 내부의 클래스, 메소드, 필드 등의 정보를 분석하여 다루는 기법을 말한다. 이를 통해 프로그램의 동적인 특성을 구현할 수 있다. 예를 들어, 리플렉션을 이용하면 실행 중인 객체의 클래스 정보를 얻어오거나, 클래스 내부의 필드나 메소드에 접근할 수 있다. 이러한 기능들은 프레임워크, 라이브러리, 테스트 코드 등에서 유용하게 활용된다.

⇒ 스프링에서는 `Reflection`을 사용해서 런타임 시 등록한 빈을 애플리케이션 내에서 사용할 수 있게 한다.

3-1. 로직을 포함하는 코드 작성

리플렉션 테스트의 대상이 될 `Account` 클래스를 생성한다.

```
public class Account {

  private String backCode;
  private String accNo;
  private String accPwd;
  private int balance;

  public Account() {}

  public Account(String bankCode, String accNo, String accPwd) {
    this.backCode = bankCode;
    this.accNo = accNo;
    this.accPwd = accPwd;
  }

  public Account(String bankCode, String accNo, String accPwd, int balance) {
    this(bankCode, accNo, accPwd);
    this.balance = balance;
  }

  /* 현재 잔액을 출력해주는 메소드 */
  public String getBalance() {

    return this.accNo + " 계좌의 현재 잔액은 " + this.balance + "원 입니다.";
  }

  /* 금액을 매개변수로 전달 받아 잔액을 증가(입금) 시켜주는 메소드 */
  public String deposit(int money) {

    String str = "";

    if(money >= 0) {
      this.balance += money;
      str = money + "원이 입금되었습니다.";
    }else {
      str = "금액을 잘못 입력하셨습니다.";
    }

    return str;
  }

  /* 금액을 매개변수로 받아 잔액을 감소(출금) 시켜주는 메소드 */
  public String withDraw(int money) {

    String str = "";

    if(this.balance >= money) {
      this.balance -= money;
      str = money + "원이 출금되었습니다.";
```

```
    }else {
      str = "잔액이 부족합니다. 잔액을 확인해주세요.";
    }

    return str;
  }
}
```

3-2. 리플렉션 테스트

3-2-1. Class

Class타입의 인스턴스는 해당 클래스의 메타정보를 가지고 있는 클래스이다.

```
/* .class 문법을 이용하여 Class 타입의 인스턴스를 생성할 수 있다. */
Class class1 = Account.class;
System.out.println("class1 : " + class1);

/* Object 클래스의 getClass() 메소드를 이용하면 Class 타입으로 리턴받아 이용할 수 있다. */
Class class2 = new Account().getClass();
System.out.println("class2 : " + class2);

/* Class.forName() 메소드를 이용하여 런타임시 로딩을 하고 그 클래스 메타정보를 Class 타입으로 반환받을 수 있다. */
try {
  Class class3 = Class.forName("com.ohgiraffers.section02.reflection.Account");
  System.out.println("class3 : " + class3);

  /* Double자료형 배열을 로드할 수 있다. */
  Class class4 = Class.forName("[D");
  Class class5 = double[].class;

  System.out.println("class4 : " + class4);
  System.out.println("class5 : " + class5);

  /* String자료형 배열을 로드할 수 있다. */
  Class class6 = Class.forName("[Ljava.lang.String;");
  Class class7 = String[].class;
  System.out.println("class6 : " + class6);
  System.out.println("class7 : " + class7);

} catch (ClassNotFoundException e) {
  e.printStackTrace();
}

/* 원시 자료형을 사용하면 컴파일 에러 발생 */
//    double d = 1.0;
//    Class class8 = d.getClass();

/* TYPE 필드를 이용하여 원시형 클래스를 반환받을 수 있다. */
Class class8 = Double.TYPE;
System.out.println("class8 : " + class8);

Class class9 = Void.TYPE;
System.out.println("class9 : " + class9);

/* 클래스의 메타 정보를 이용하여 여러 가지 정보를 반환받는 메소드를 제공한다. */
/* 상속된 부모 클래스를 반환한다. */
Class superClass = class1.getSuperclass();
System.out.println("superClass : " + superClass);
```

▼ 실행 결과

```
class1 : class com.ohgiraffers.section02.reflection.Account
class2 : class com.ohgiraffers.section02.reflection.Account
class3 : class com.ohgiraffers.section02.reflection.Account
class4 : class [D
class5 : class [D
class6 : class [Ljava.lang.String;
class7 : class [Ljava.lang.String;
class8 : double
class9 : void
superClass : class java.lang.Object
```

3-2-2. field

field 정보에 접근할 수 있다.

```
Field[] fields = Account.class.getDeclaredFields();
for(Field field : fields) {
  System.out.println("modifiers : " + Modifier.toString(field.getModifiers()) +
      ", type : " + field.getType() +
      ", name : " + field.getName() );
}
```

▼ 실행 결과

```
modifiers : private, type : class java.lang.String, name : backCode
modifiers : private, type : class java.lang.String, name : accNo
modifiers : private, type : class java.lang.String, name : accPwd
modifiers : private, type : int, name : balance
```

3-2-3. 생성자

생성자 정보에 접근할 수 있다.

```
Constructor[] constructors = Account.class.getConstructors();
for(Constructor con : constructors) {
  System.out.println("name : " + con.getName());

  Class[] params = con.getParameterTypes();
  for(Class param : params) {
    System.out.println("paramType : " + param.getTypeName());
  }
}
```

▼ 실행 결과

```
name : com.ohgiraffers.section02.reflection.Account
paramType : java.lang.String
paramType : java.lang.String
paramType : java.lang.String
paramType : int
name : com.ohgiraffers.section02.reflection.Account
paramType : java.lang.String
paramType : java.lang.String
paramType : java.lang.String
name : com.ohgiraffers.section02.reflection.Account
```

생성자를 이용하여 인스턴스를 생성할 수 있다.

```
try {
  Account acc = (Account) constructors[0].newInstance("20", "110-223-123456", "1234", 10000);
  System.out.println(acc.getBalance());
} catch (InstantiationException | IllegalAccessException | IllegalArgumentException
    | InvocationTargetException e) {
    e.printStackTrace();
}
```

▼ 실행 결과

```
110-223-123456 계좌의 현재 잔액은 10000원 입니다.
```

3-2-4. 생성자

메소드 정보에 접근할 수 있다.

```
Method[] methods = Account.class.getMethods();
Method getBalanceMethod = null;
for(Method method : methods) {
  System.out.println(Modifier.toString(method.getModifiers()) + " " +
          method.getReturnType().getSimpleName() + " " +
          method.getName());

  if("getBalance".equals(method.getName())) {
    getBalanceMethod = method;
  }
}
```

▼ 실행 결과

```
public String getBalance
public String withDraw
public String deposit
public final native void wait
public final void wait
public final void wait
public boolean equals
public String toString
public native int hashCode
public final native Class getClass
public final native void notify
public final native void notifyAll
```

invoke 메소드로 메소드를 호출할 수 있다.

```
try {
  System.out.println(getBalanceMethod.invoke(((Account) constructors[2].newInstance())));
} catch (IllegalAccessException | IllegalArgumentException | InvocationTargetException e) {
  e.printStackTrace();
} catch (InstantiationException e) {
  e.printStackTrace();
}
```

▼ 실행 결과

```
null 계좌의 현재 잔액은 0원 입니다.
```

4. Proxy

Java에서 프록시(Proxy)는 대리자를 의미한다. 프록시는 기존의 객체를 감싸서 그 객체의 기능을 확장하거나 변경할 수 있게 해준다. 예를 들어, 프록시 객체를 사용하면 객체에 대한 접근을 제어하거나, 객체의 메소드 호출 전후에 로깅 작업 등을 수행할 수 있다. 또한, 프록시 객체를 사용하여 원격으로 실행되는 객체를 호출할 수도 있다. 프록시는 주로 AOP(Aspect Oriented Programming)에서 사용된다.

프록시 생성은 크게 두 가지 방식이 제공된다.

1. JDK Dynamic Proxy 방식

   리플렉션을 이용해서 proxy클래스를 동적으로 생성해주는 방식으로 타겟의 인터페이스를 기준으로 proxy를 생성해준다. 사용자의 요청이 타겟을 바라보고 실행될 수 있도록 타겟 자체에 대한 코드 수정이 아닌 리플렉션을 이용한 방식으로, 타겟의 위임 코드를 InvocationHandler를 이용하여 작성하게 된다. 하지만 사용자가 타겟에 대한 정보를 잘못 주입하는 경우가 발생할 수 있기 때문에 내부적으로 주입된 타겟에 대한 검증 코드를 거친 후 invoke가 동작하게 된다.

2. CGLib 방식

   동적으로 Proxy를 생성하지만 바이트코드를 조작하여 프록시를 생성해주는 방식이다. 인터페이스 뿐 아니라 타겟의 클래스가 인터페이스를 구현하지 않아도 프록시를 생성해준다. CGLib(Code Generator Library)의 경우에는 처음 메소드가 호출된 당시 동적으로 타켓 클래스의 바이트 코드를 조작하게 되고, 그 이후 호출 시부터는 변경된 코드를 재사용한다. 따라서 매번 검증 코드를 거치는 1번 방식보다는 invoke시 더 빠르게 된다. 또한 리플렉션에 의한 것이 아닌 바이트코드를 조작하는 방식이기 때문에 성능면에서는 더 우수하다.

하지만 CGLib 방식은 스프링에서 기본적으로 제공되는 방식은 아니었기에 별도로 의존성을 추가하여 개발해야 했고, 파라미터가 없는 default 생성자가 반드시 필요했으며, 생성된 프록시의 메소드를 호출하면 타겟의 생성자가 2번 호출되는 등의 문제점들이 있었다.

스프링 4.3, 스프링부트 1.3 이후부터 CGLib의 문제가 된 부분이 개선되어 기본 core 패키지에 포함되게 되었고, 스프링에서 기본적으로 사용하는 프록시 방식이 CGLib 방식이 되었다.

4-1. 로직을 포함하는 코드 작성

1. Student 인터페이스 작성

```
public interface Student {

  void study(int hours);
}
```

2. OhgiraffersStudent 클래스 작성 (Student 인터페이스 구현)

```
public class OhgiraffersStudent implements Student {

   @Override
     public void study(int hours) {
        System.out.println(hours + "시간 동안 열심히 공부합니다.");
     }
}
```

4-2. dynamic

4-2-1. Handler 클래스 작성

`java.lang.reflect.InvocationHandler`를 구현한 클래스를 작성한다.

`Student` 클래스를 타겟 인스턴스로 설정하고 `invoke` 메소드를 정의한다.

```
public class Handler implements InvocationHandler {

  /* 메소드 호출을 위해 타겟 인스턴스가 필요하다 */
  private final Student student;

  public Handler(Student student) {
    this.student = student;
  }

  /* 생성된 proxy 인스턴스와 타겟 메소드, 전달받은 인자를 매개변수로 한다. */
  @Override
  public Object invoke(Object proxy, Method method, Object[] args)
    throws IllegalAccessException, IllegalArgumentException, InvocationTargetException {

    System.out.println("============ 공부가 너무 하고 싶습니다. ==============");
    System.out.println("호출 대상 메소드 : " + method);
    for(Object arg : args) {
      System.out.println("전달된 인자 : " + arg);
    }

   /* 타켓 메소드를 호출한다. 타겟 Object와 인자를 매개변수로 전달한다.
    * 여기서 프록시를 전달하면 다시 타겟을 호출할 때 다시 프록시를 생성하고 다시 또 전달하는 무한 루프에 빠지게 된다.
    * */
   method.invoke(student, args);

   System.out.println("============ 공부를 마치고 수면 학습을 시작합니다. ============");

   return proxy;

  }
}
```

4-2-2. Application 실행 클래스 작성

```
Student student = new OhgiraffersStudent();
Handler handler = new Handler(student);

/* 클래스로더, 프록시를 만들 클래스 메타 정보(인터페이스만 가능), 프록시 동작할 때 적용될 핸들러 */
Student proxy
= (Student) Proxy.newProxyInstance(Student.class.getClassLoader(), new Class[] {Student.class}, handler);
```

```
/* 프록시로 감싸진 인스턴스의 메소드를 호출하게 되면 핸들러에 정의한 메소드가 호출된다. */
proxy.study(16);
```

▼ 실행 코드

```
============ 공부가 너무 하고 싶습니다. ==============
호출 대상 메소드 : public abstract void com.ohgiraffers.section03.proxy.common.Student.study(int)
전달된 인자 : 16
16시간 동안 열심히 공부합니다.
============ 공부를 마치고 수면 학습을 시작합니다. ============
```

`study` 메소드 호출 시 `proxy` 객체의 동작을 확인할 수 있다.

4-3. cglib

4-3-1. Handler 클래스 작성

`org.springframework.cglib.proxy.InvocationHandler`를 구현한 클래스를 작성한다.

`OhgiraffersStudent` 클래스를 타겟 인스턴스로 설정하고 `invoke` 메소드를 정의한다.

```
public class Handler implements InvocationHandler {

  private final OhgiraffersStudent student;

  public Handler(OhgiraffersStudent student) {
   this.student = student;
  }

  @Override
  public Object invoke(Object proxy, Method method, Object[] args)
    throws IllegalAccessException, IllegalArgumentException, InvocationTargetException {

    System.out.println("============ 공부가 너무 하고 싶습니다. ==============");
    System.out.println("호출 대상 메소드 : " + method);
    for(Object arg : args) {
      System.out.println("전달된 인자 : " + arg);
    }

    method.invoke(student, args);

    System.out.println("============ 공부를 마치고 수면 학습을 시작합니다. ============");

    return proxy;

  }
}
```

4-3-2. Application 실행 클래스 작성

```
OhgiraffersStudent student = new OhgiraffersStudent();
Handler handler = new Handler(student);

/* Enhancer 클래스의 create static 메소드는 타겟 클래스의 메타정보와 핸들러를 전달하면 proxy를 생성해서 반환해준다. */
OhgiraffersStudent proxy
= (OhgiraffersStudent) Enhancer.create(OhgiraffersStudent.class, new Handler(new OhgiraffersStudent()));
```

```
proxy.study(20);
```

▼ 실행 코드

```
============ 공부가 너무 하고 싶습니다. ==============
호출 대상 메소드 : public void com.ohgiraffers.section03.proxy.common.OhgiraffersStudent.study(int)
전달된 인자 : 20
20시간 동안 열심히 공부합니다.
============ 공부를 마치고 수면 학습을 시작합니다. ============
```

`study` 메소드 호출 시 `proxy` 객체의 동작을 확인할 수 있다.