AOP

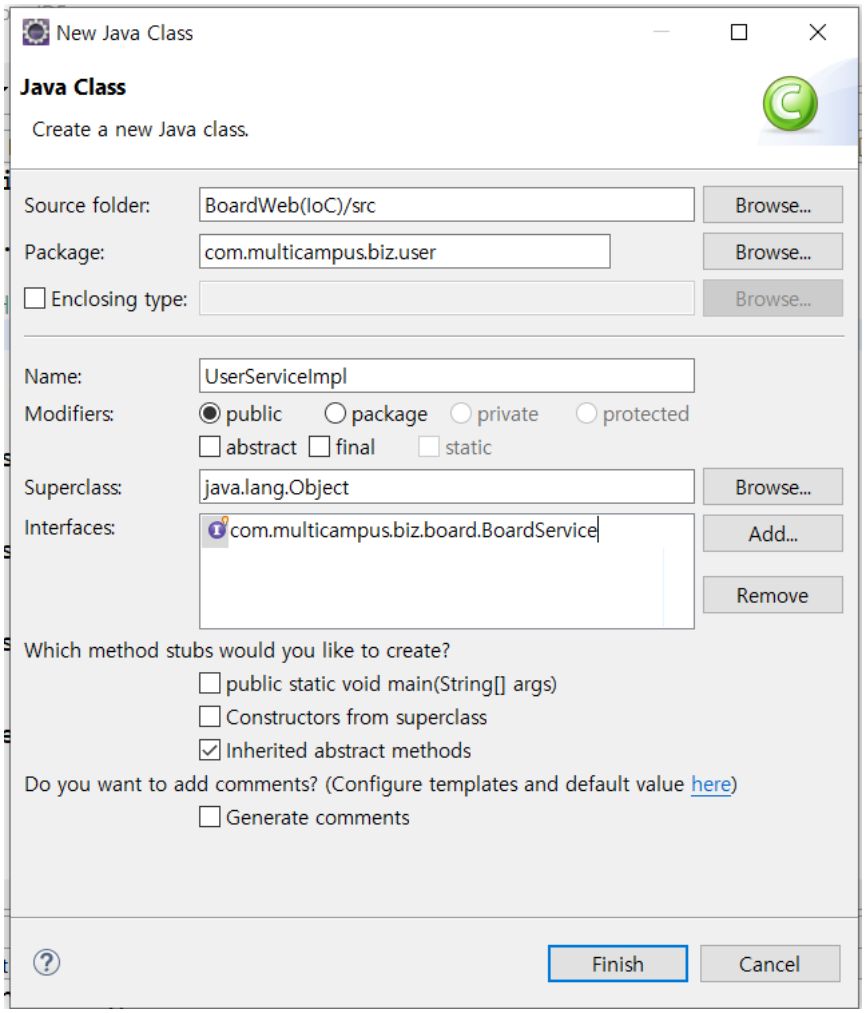
Aspect-Oriented Programming

왜 클라이언트가 인터페이스를 호출하고 서비스 호출하고 DAO호출하고 할까? 그냥 DAO 해도되는데?

답: 트렌젝션때문에

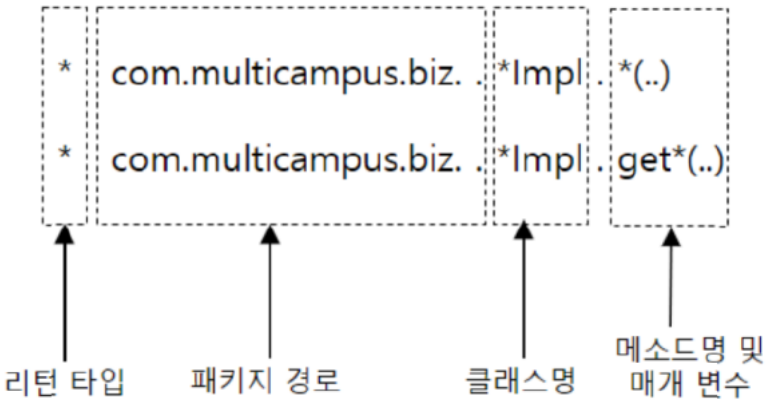
```
클라이언트가 우리은행 DAO 메소드를 해서 인출을 해요.
클라이언트가 신한은행 DAO 메소드를 호출해서 입금을 해요.
근데 신한은행에서 실패했어요. 그럼 돈 날아가요. 왜냐면 DAO하면 커밋되니까.

그래서 이체라는 비지니스 메서드(serviceImpl)가 필요한 것이예요.
이체에서 두번의 디비 연동을 할 것이고 잘 되면 커밋 안되면 롤백해야되니까.
성능상의 차이는 없으니까 그냥 트랜젝션 관리를 위해서도 비지니스 메소드관리는 필요합니다.
```

- Impl class 파일 만들 때, Add 눌러서 service interface 파일 추가해주기.



용어정리

- 조인포인트(Joinpoint)
  - 조인포인트는 클라이언트가 시스템을 사용하면서 호출하는 모든 비즈니스 메소드를 의미한다.
- 포인트컷(Pointcut)
  - 클라이언트가 호출하는 모든 비즈니스 메소드가 조인포인트라면, 포인트컷은 필터링된 조인포인트를 의미한다.



  - <aop:pointcut id="allPointcut" expression="execution(* com.multicampus.biz..*Impl*.(..))"/>
  - 리턴 경로 지정
    - 가장 일반적인 반환형 지정은 '*' 캐릭터를 이용하는 것이다.

표현식	의 미
*	모든 반환형 허용
void	반환형이 void인 메소드 선택
!void	반환형이 void가 아닌 메소드 선택

  - 패키지 경로 지정

표현식	의 미
com.multicampus.biz.board	정확하게 com.multicampus.biz.board 패키지만 선택
com.multicampus..	com.multicampus 패키지로 시작하는 모든 패키지를 선택
com.multicampus..board	com.multicampus 패키지로 시작하면서 마지막 패키지 이름이 impl로 끝나는 패키지만 선택

  - 클래스 이름 지정

표현식	의 미
BoardServiceImpl	정확하게 BoardServiceImpl 클래스만 선택
*Impl	클래스 이름이 Impl로 끝나는 모든 클래스를 선택
BoardService+	해당 클래스로부터 파생된 모든 자식 클래스 선택

  - 메소드 지정

표현식	의 미
*	가장 기본 설정으로 모든 메소드 선택
get*	메소드 이름이 get으로 시작하는 모든 메소드 선택

- 매개변수 지정

표현식	의 미
(..)	가장 기본 설정으로서 매개변수의 개수와 타입에 제약이 없음을 의미
(*)	반드시 1개의 매개변수를 가지는 메소드만 선택
(BoardVO)	매개변수로 BoardVO를 가지는 메소드만 선택. 이때 매개 변수로 지정된 클래스는 패키지 경로가 반드시 포함되어야 함. (com.multicampus.biz.board.BoardVO)
(Integer, ..)	한 개 이상의 매개변수를 갖되, 첫 번째 매개변수가 Integer인 메소드만 선택
(Integer, *)	반드시 두 개의 매개변수를 갖되, 첫 번째 매개변수가 Integer인 메소드만 선택

- 어드바이스
  - before와 after 외에도 after-returning, after-throwing, around를 포함하여
    총 5가지의 동작 시점을 제공한다

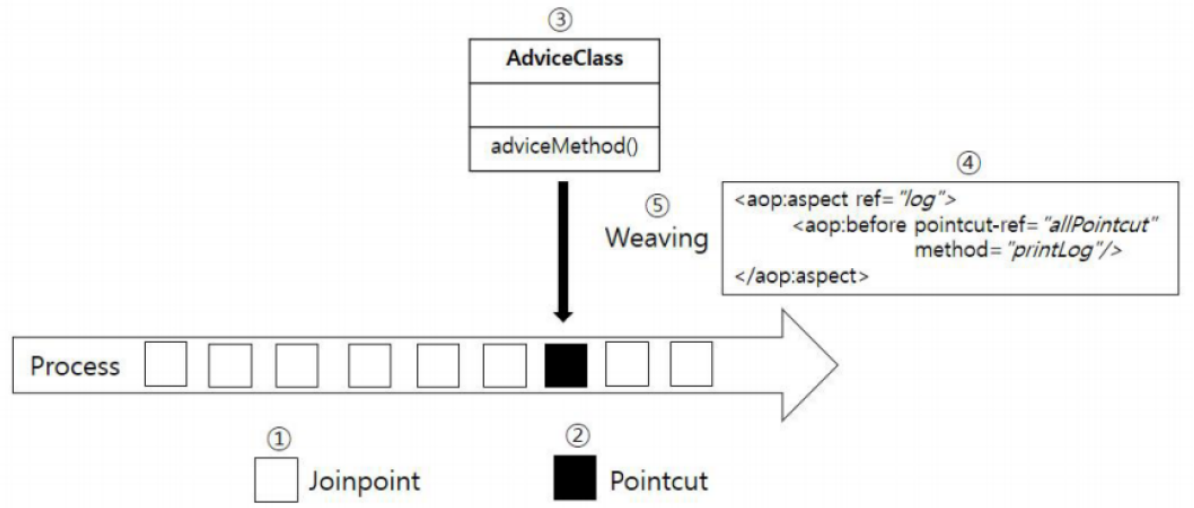
```
<aop:aspect ref="log">
    <aop:before pointcut-ref="allPointcut" method="printLog"/>
</aop:aspect>
```

- 애스팩트(Aspect) or 어드바이저(Advisor)
  - 애스팩트는 포인트컷과 어드바이스의 결합

AOP 결론✨

```
<!-- 횡단관심에 해당하는 Advice 클래스를 등록 -->
<bean id="log" class="com.multicampus.biz.common.LogAdvice"></bean>

<!-- AOP 설정 -->
<aop:config>

    <aop:pointcut id="allPointcut" expression="execution(* com.multicampus.biz..*Impl.*(..))"/>
    <aop:pointcut id="getPointcut" expression="execution(* com.multicampus.biz..*Impl.get*(..))"/>

    <aop:aspect ref="log">
        <aop:before pointcut-ref="allPointcut" method="printLog"/>
    </aop:aspect>
```

```
컨테이너야 allPointcut 필터링 된 비지니스 메서드가 실행되기 이전에 aspect인 log 객체가
가진 printLog 실행해줘.

‼ aspect: 포인트컷과 어드바이스의 연결고리 ‼
```



어드바이스 동작 시점

before

→ 사전

after

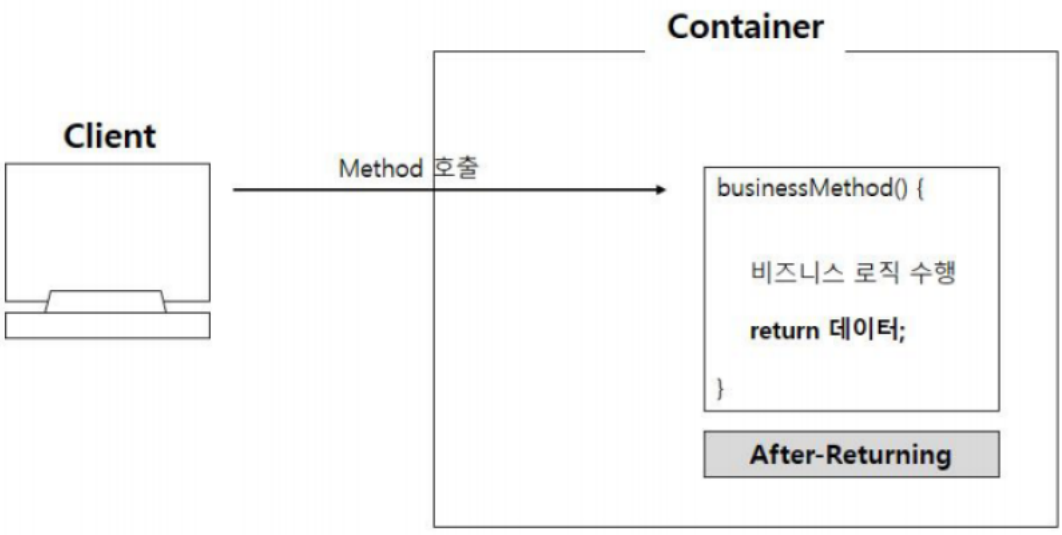
→ 사후

after-returning

→ after과 차이 : returning 속성을 사용할 수 있다.

→ 즉, after은 log출력밖에 못하는데 returning은 비지니스 메서드 리턴값을 받아서 사후처리 할 수 있다.

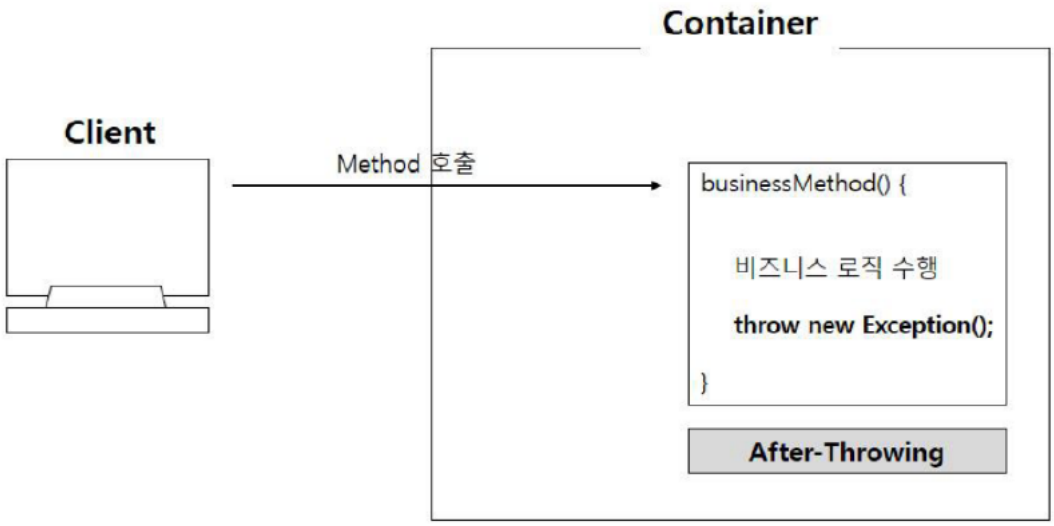
```
<aop:aspect ref="afterReturning">
    <aop:after-returning pointcut-ref="getPointcut" method="afterLog" returning="returnobj"/>
</aop:aspect>
```



after-throwing

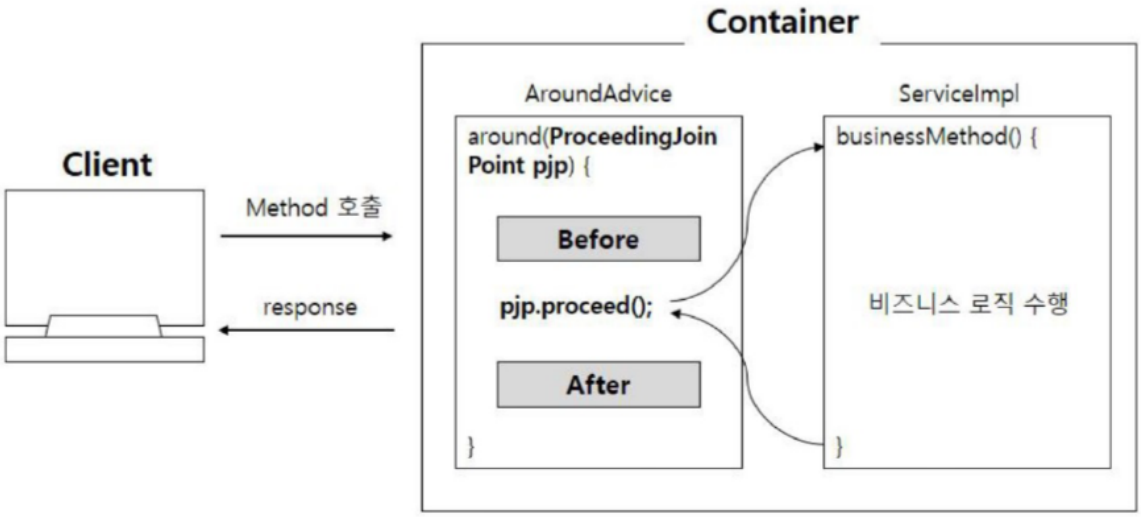
→ throwing 이란 예외가 발생했을 때, After-Throwing으로 점프해서 발생한 예외처리를 매개변수로 받아서 분기 시킬 수 있다.

```
<aop:aspect ref="afterThrowing">
    <aop:after-throwing pointcut-ref="allPointcut" method="exceptionLog" throwing="exceptionobj"/>
</aop:aspect>
```



around

→ 비지니스 메소드를 기준으로 사전 처리와 사후 처리를 모두 하고 싶을 때



정리

어노테이션	설 명
@Before	비즈니스 메소드 수행 전에 동작
@AfterReturning	비즈니스 메소드의 리턴 데이터를 받아서 동작
@AfterThrowing	비즈니스 메소드 실행 중 예외가 발생하면 동작
@After	비즈니스 메소드가 실행된 후, 무조건 동작
@Around	비즈니스 메소드 호출을 가로채서 사전처리 사후처리로 동작

문법

- Before, After Returning, After Throwing, After 어드바이스에서는 JoinPoint를 사용해야 하고, 유일하게 Around 어드바이스에서만 ProceedingJoinPoint를 매개변수로 사용해야 한다.
- 이는 Around 어드바이스가 proceed 메소드를 필요로 하기 때문이다.
- JoinPoint와 ProceedingJoinPoint 모두 반드시 첫 번째 매개변수로 선언되어야 한다.

자바는 느리지만 유지보수가 쉽다. 빠르고 메모리 적게 이용하는 거 쓸라면 C++하셈

```
public void deleteBoard(BoardVO vo) {
    boardDAO.deleteBoard(vo);
}
```

```
public void deleteBoard(int seq, String password) {
    boardDAO.deleteBoard(seq, password);
}
```

- 값이 몇개만 필요하다 해도 객체 자체를 가져와서 메모리 낭비를 하며 처리하는 이유는 나중에 계속 필요한 매개변수가 계속 변할 수 있기 때문이다. - 귀찮아짐.

Annotation 사용을 위한 XML 설정🎨

- 빈 등록 하지말고 component 하자

```
<!-- 횡단관심에 해당하는 Advice 클래스를 등록 -->
<bean id="log"            class="com.multicampus.biz.common.LogAdvice"></bean>
<bean id="afterReturning" class="com.multicampus.biz.common.AfterReturningAdvice"></bean>
<bean id="afterThrowing"  class="com.multicampus.biz.common.AfterThrowingAdvice"></bean>
<bean id="aruond"         class="com.multicampus.biz.common.AroundAdvice"></bean>
```

- 이것도 없앨 수 있음.

```xml
<!-- 횡단관심에 해당하는 Advice 클래스를 등록 -->
<!--
<bean id="log"              class="com.multicampus.biz.common.LogAdvice"></bean>
<bean id="afterReturning"   class="com.multicampus.biz.common.AfterReturningAdvice"></bean>
<bean id="afterThrowing"    class="com.multicampus.biz.common.AfterThrowingAdvice"></bean>
<bean id="aruond"           class="com.multicampus.biz.common.AroundAdvice"></bean>
-->

<!-- AOP 설정 -->
<!-- <aop:config>

    <aop:pointcut id="allPointcut" expression="execution(* com.multicampus.biz..*Impl.*(..))"/>
    <aop:pointcut id="getPointcut" expression="execution(* com.multicampus.biz..*Impl.get*(..))"/>

    <aop:aspect ref="log">
        <aop:before pointcut-ref="allPointcut" method="printLog"/>
    </aop:aspect>

    <aop:aspect ref="afterReturning">
        <aop:after-returning pointcut-ref="getPointcut" method="afterLog" returning="returnobj"/>
    </aop:aspect>

    <aop:aspect ref="afterThrowing">
        <aop:after-throwing pointcut-ref="allPointcut" method="exceptionLog" throwing="exceptionobj"/>
    </aop:aspect>

    <aop:aspect ref="aruond">
        <aop:around pointcut-ref="allPointcut" method="aroundLog"/>
    </aop:aspect>

</aop:config> -->
```

- 바로 이런 식으로 포인트컷 설정

```java
@Service
// aspect = Pointcut(필터링 된 핵심관심) + Advice(공통분리 된 횡단관심)
@Aspect
public class LogAdvice {

    @Pointcut("execution(* com.multicampus.biz..*Impl.*(..))")
    public void allPointcut () {}

    @Before("allPointcut()")
    public void printLog(JoinPoint jp) {
        String methodName = jp.getSignature().getName();
        Object[] args = jp.getArgs();

        System.out.println("[사전 처리] " + methodName + " 메소드 ARGS 정보 : " + args[0].toString());
    }
```

```java
@Service
@Aspect
public class AfterThrowingAdvice {

    @Pointcut("execution(* com.multicampus.biz..*Impl.*(..))")
    public void allPointcut () {}

    @AfterThrowing(pointcut = "allPointcut()", throwing = "exceptionobj")
    public void exceptionLog(JoinPoint jp, Exception exceptionobj) {
        String methodName = jp.getSignature().getName();
        System.out.println("[예외 처리]" + methodName + " 메소드 수행 중 예외 발생!!!");

        // 발생된 예외의 종류에 따른 분기처리
```

```java
@Service
@Aspect
public class AroundAdvice {

    @Pointcut("execution(* com.multicampus.biz..*Impl.*(..))")
    public void allPointcut () {}

    // Around로 등록되는 메소드는 리턴타입(Object)과 매개변수(ProceedingJoinPoint)가 고정된다.
    @Around("allPointcut()")
    public Object aroundLog(ProceedingJoinPoint jp) throws Throwable {
        String methodName = jp.getSignature().getName();
        Object obj = null;
        StopWatch watch = new StopWatch();
```

- 실수주의
  - returning 일 때는 뭔가를 뱉어야 하니까 뭔가 리턴되는 메소드에서 해야되는거 잘 생각하기
  - ex) get인 거

```java
@Service
@Aspect
public class AfterReturningAdvice {

    @Pointcut("execution(* com.multicampus.biz..*Impl.get*(..))")
    public void getPointcut () {}

    @AfterReturning(pointcut = "getPointcut()", returning = "returnobj")
    public void afterLog(JoinPoint jp, Object returnobj) {
        String methodName = jp.getSignature().getName();
        System.out.println("[사후 처리] " + methodName + " 비즈니스 메소드 리턴값 : " + returnobj.toString());
```

- xml

```xml
<!-- Annotation 기반의 AOP 설정 -->
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

- 이거 간략하게 하기

```java
@Service
@Aspect
public class AfterReturningAdvice {

    @Pointcut("execution(* com.multicampus.biz..*Impl.get*(..))")
    public void getPointcut () {}

    @AfterReturning(pointcut = "getPointcut()", returning = "returnobj")
    public void afterLog(JoinPoint jp, Object returnobj) {
        String methodName = jp.getSignature().getName();
```

- class 생성 이후

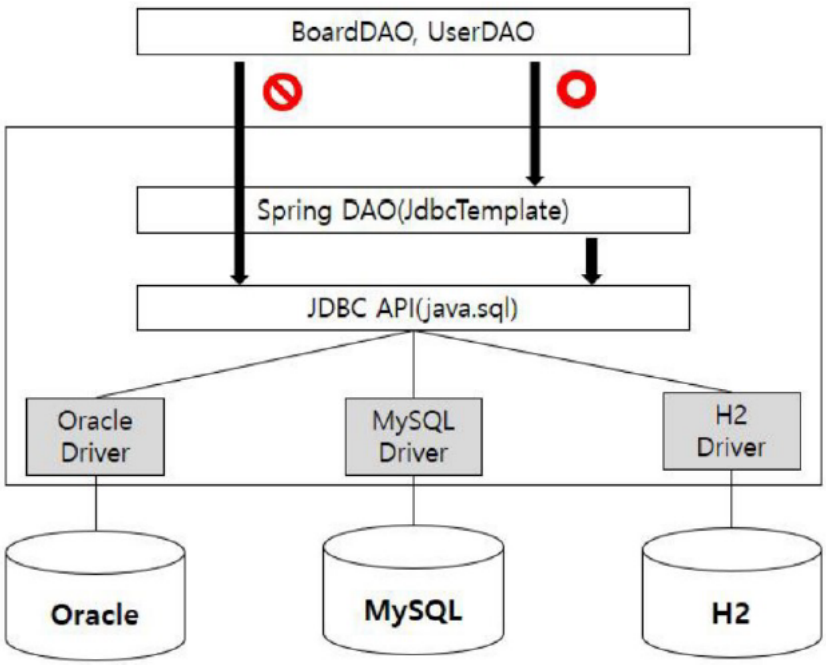
BoardPointcut.java | AroundAdvice.java | LogAdvice.java

```java
package com.multicampus.biz.common;

import org.aspectj.lang.annotation.Aspect;

@Aspect
public class BoardPointcut {

    @Pointcut("execution(* com.multicampus.biz..*Impl.*(..))")
    public void allPointcut() {}

    @Pointcut("execution(* com.multicampus.biz..*Impl.get*(..))")
    public void getPointcut() {}

    @Pointcut("execution(* com.multicampus.biz.board.*Impl.*(..))")
    public void boardPointcut() {}

    @Pointcut("execution(* com.multicampus.biz.user.*Impl.*(..))")
    public void userPointcut() {}
}
```

- class 이름 적어주면 끗

```java
@Service
@Aspect
public class AfterReturningAdvice {

    @AfterReturning(pointcut = "BoardPointcut.getPointcut()", returning = "returnobj")
    public void afterLog(JoinPoint jp, Object returnobj) {
```

JDBC🛍️

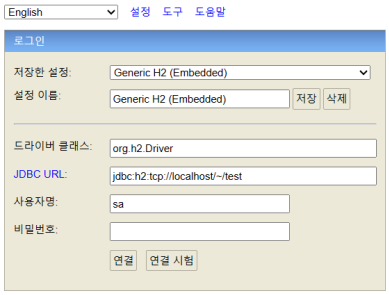


jdbc 메서드만 호출하면 코드가 엄청 줄어든다.

사용법

```
<!-- DataSource 등록 -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName"    value="org.h2.Driver"></property>
    <property name="url"                value="jdbc:h2:tcp://localhost/~/test"></property>
    <property name="username"           value="sa"></property>
    <property name="password"       value=""></property>
</bean>
```

```
<!-- JdbcTemplate 등록 -->
<bean class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"></property>
</bean>
```



코드

```
package com.multicampus.biz.board;

import java.util.ArrayList;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

// 2. DAO(Data Access Object) 클래스
@Repository
public class BoardDAOSPRING {

  @Autowired
  private JdbcTemplate spring;

  // BOARD 테이블 관련 SQL 명령어들
  private final String BOARD_INSERT = "insert into board(seq, title, writer, content) values((select nvl(max(seq), 0)+1 from board),?,
  private final String BOARD_UPDATE = "update board set title=?, content=? where seq=?";
  private final String BOARD_DELETE = "delete board where seq=?";
  private final String BOARD_GET = "select * from board where seq=?";
  private final String BOARD_LIST = "select * from board order by seq desc";

  public BoardDAOSPRING() {
    System.out.println("===> BoardDAOSPRING 생성");
  }

  // CRUD 기능의 메소드 구현
  // 글 등록
  public void insertBoard(BoardVO vo) {
    System.out.println("===> SPRING 기반으로 insertBoard() 기능 처리");
    spring.update(BOARD_INSERT, vo.getTitle(), vo.getWriter(), vo.getContent());
  }

  // 글 수정
  public void updateBoard(BoardVO vo) {
    System.out.println("===> SPRING 기반으로 updateBoard() 기능 처리");
    spring.update(BOARD_UPDATE, vo.getTitle(), vo.getContent(), vo.getSeq());
  }

  // 글 삭제
  public void deleteBoard(BoardVO vo) {
    System.out.println("===> SPRING 기반으로 deleteBoard() 기능 처리");
    spring.update(BOARD_DELETE, vo.getSeq());
  }

  // 글 상세 조회
  public BoardVO getBoard(BoardVO vo) {
    System.out.println("===> SPRING 기반으로 getBoard() 기능 처리");
    BoardVO board = null;
    return board;
  }

  // 글 목록 검색
  public List<BoardVO> getBoardList(BoardVO vo) {
    System.out.println("===> SPRING 기반으로 getBoardList() 기능 처리");
    return spring.query(BOARD_LIST, new BoardRowMapper());
  }
}
```

- 등록 수정 삭제 전부 update로 함.

```java
// CRUD 기능의 메소드 구현
// 글 등록
public void insertBoard(BoardVO vo) {
    System.out.println("===> SPRING 기반으로 insertBoard() 기능 처리");
    spring.update(BOARD_INSERT, vo.getTitle(), vo.getWriter(), vo.getContent());
}

// 글 수정
public void updateBoard(BoardVO vo) {
    System.out.println("===> SPRING 기반으로 updateBoard() 기능 처리");
    spring.update(BOARD_UPDATE, vo.getTitle(), vo.getContent(), vo.getSeq());
}

// 글 삭제
public void deleteBoard(BoardVO vo) {
    System.out.println("===> SPRING 기반으로 deleteBoard() 기능 처리");
    spring.update(BOARD_DELETE, vo.getSeq());
}
```

select를 하는 경우 저장해둔 정보가 필요하므로 모든 테이블 마다 BoardRowMapper 같은 class만들어줘야함.

```java
package com.multicampus.biz.board;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class BoardRowMapper implements RowMapper<BoardVO> {

    @Override
    public BoardVO mapRow(ResultSet rs, int rowNum) throws SQLException {
        // ResultSet에 있는 검색 결과를 VO 객체에 매핑한다.
        BoardVO board = new BoardVO();
        board.setSeq(rs.getInt("SEQ"));
        board.setTitle(rs.getString("TITLE"));
        board.setWriter(rs.getString("WRITER"));
        board.setContent(rs.getString("CONTENT"));
        board.setRegDate(rs.getDate("REGDATE"));
        board.setCnt(rs.getInt("CNT"));
        return board;
    }

}
```

→ 이것이 귀찮아서 생긴 게 myBatis, iBatis

- Impl의 객체는 인터페이스여야한다.

UserServiceImpl.java | business-layer.xml | UserDAOSPRING.java | UserRowMapper.java

```java
package com.multicampus.biz.user;

import java.util.List;

@Service("userService")
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDAO userDAO;

    public void insertUser(UserVO vo) {}
    public void updateUser(UserVO vo) {}
    public void deleteUser(UserVO vo) {}
    public UserVO getUser(UserVO vo) {
        return userDAO.getUser(vo);
    }
    public List<UserVO> getUserList(UserVO vo) {
```

- 인터페이스로 만든 애들은 메모리에 올리면 안되니까 @Repository 주석처리해줘야함.

```java
// 2. DAO(Data Access Object) 클래스
// @Repository
public class UserDAOSPRING implements UserDAO {

    @Autowired
    private JdbcTemplate spring;

    // USERS 테이블 관련 SQL 명령어들
    private final String USER_GET = "select * from use
```

```java
// 2. DAO(Data Access Object) 클래스
// @Repository
public class UserDAOJDBC implements UserDAO {
    // JDBC 관련 변수 선언
    private Connection conn = null;
    private PreparedStatement stmt = null;
```

DataSource 설정하는 방법

→ properties 파일 이용하기.

datasource.properties

```
# DataSource Setting
board.jdbc.driverClassName=org.h2.Driver
board.jdbc.jdbcUrl=jdbc:h2:tcp://localhost/~/test
board.jdbc.username=sa
board.jdbc.password=
```

*business-layer.xml

```
    <!-- DataSource 등록 -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName"    value="org.h2.Driver"></property>
        <property name="url"                value="jdbc:h2:tcp://localhost/~/test"></property>
        <property name="username"           value="sa"></property>
        <property name="password"           value=""></property>
    </bean>
```

→ XML에 있던 거 옮기고

```
<!-- DataSource 등록 -->
<context:property-placeholder location="classpath:datasoure.properties" />

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName"    value="${board.jdbc.driverClassName}"></property>
    <property name="url"                value="${board.jdbc.jdbcUrl}"></property>
    <property name="username"           value="${board.jdbc.username}"></property>
    <property name="password"           value="${board.jdbc.password}"></property>
</bean>
```

→ 바꿔줌

▼ 자바는 다중 상속 허용 안함.

따라서 DAO는 최대한 독립된 클래스로 만들어서 언제든지 부모를 지정할 수 있게 해야한다.

뭔 말이냐면 DAO를 개많이 만들게 되면 SuperDAO를 만들어서 모두를 상속하고 싶어질 수 있는데 이미 누군갈 상속하고 있는 놈은 상속 받지 못한다.

트렌젝션 설정

```
<!-- Transaction 설정 -->
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="*" rollback-for="Exception"/>
    </tx:attributes>
</tx:advice>
```

- txAdvice 객체가 생기고 트렌젝션을 txManager로 한다. (커밋과 롤백)
- 만약 모든 메소드에서 예외가 발생하면 롤백해라 라는 뜻
- 어드바이스가 매니저를 통해서 커밋과 롤백을 실행할 것이다.
- 포인트컷(비지니스 서비스)과 어드바이스를 연결하는걸 aspect라고 한다.
    - 근데 트렌젝션은 동적이라 메소드 이름을 모르니까 aspect대신 advisor을 사용

```xml
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="*" rollback-for="Exception"/>
    </tx:attributes>
</tx:advice>

<aop:config>
    <aop:pointcut id="txPointcut" expression="execution(* com.multicampus.biz..*Impl.*(..))"/>

    <aop:advisor pointcut-ref="txPointcut" advice-ref="txAdvice"/>
</aop:config>
```