

DAY - 02

Spring AOP

Spring DAO

AOP 개요

- AOP(Aspect Oriented Programming)

- 관심 분리(Separation of Concerns)
- 횡단 관심(Crosscutting Concerns)
- 핵심 관심(Core Concerns)

```
businessMethod( ) {
```

```
    Logging...
```

```
    비즈니스 로직(3 ~ 5 라인)
```

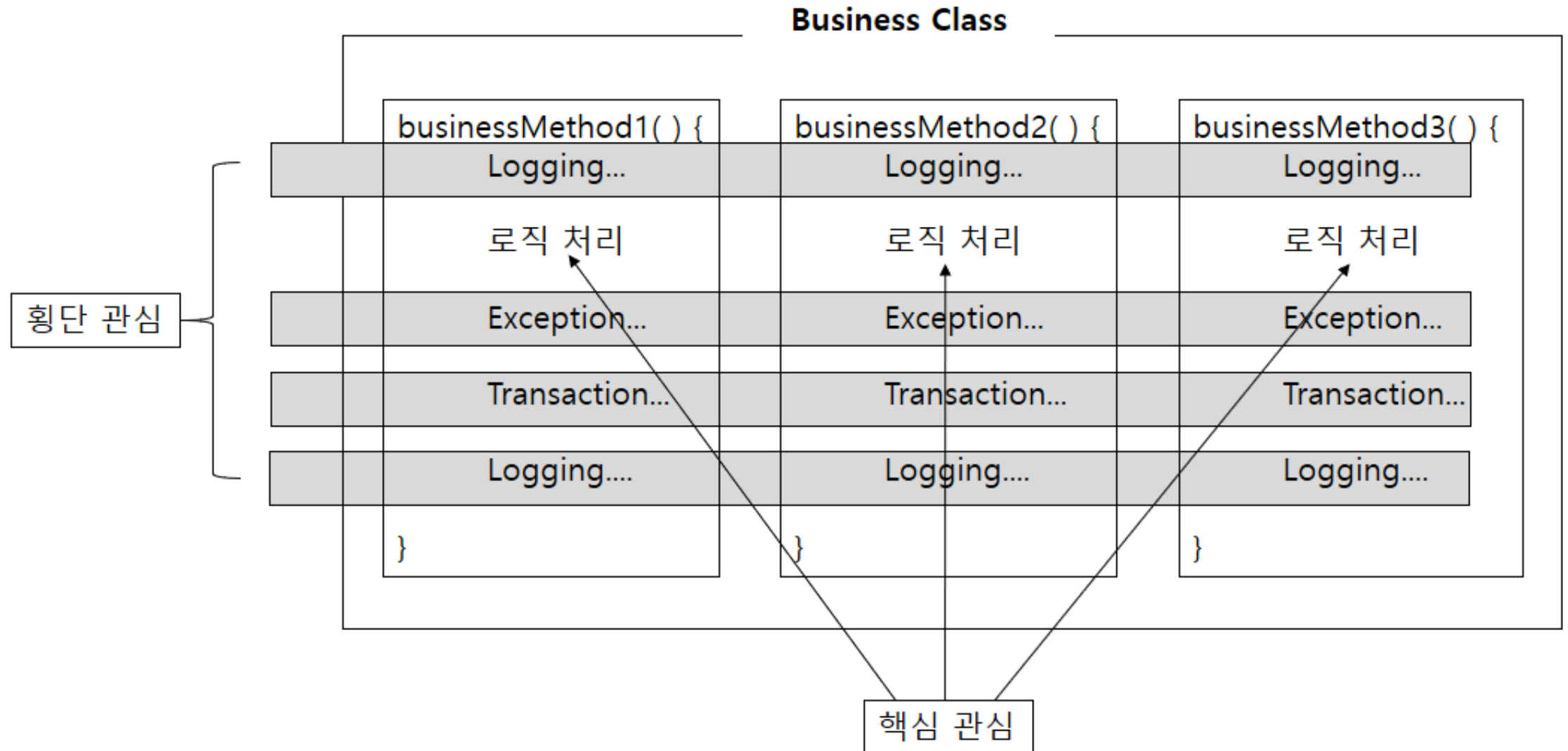
```
    Exception Handle...
```

```
    Transaction Handle...
```

```
    Logging....
```

```
}
```

- 핵심 관심(Core Concerns) 과 횡단 관심(Crosscutting Concerns)



Spring AOP Quick Start

- 실습 순서

- 1. LogAdvice 클래스 작성
- 2. applicationContext.xml 작성
- 3. BoardServiceClient 작성 및 테스트

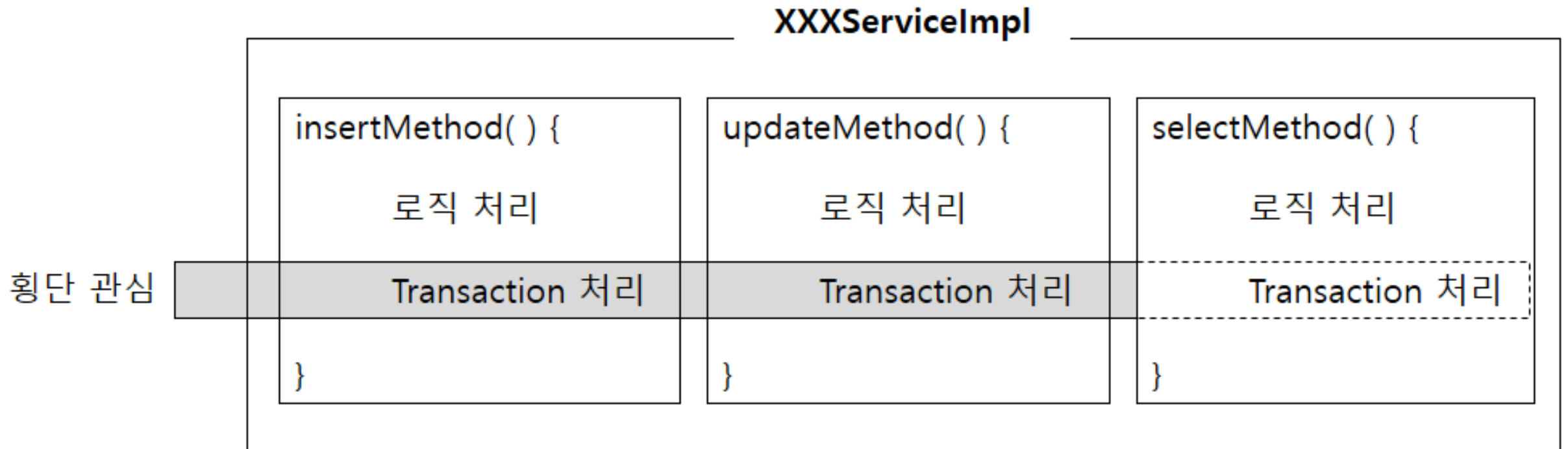
AOP 용어 정리

- 조인포인트 (*Joinpoint*)

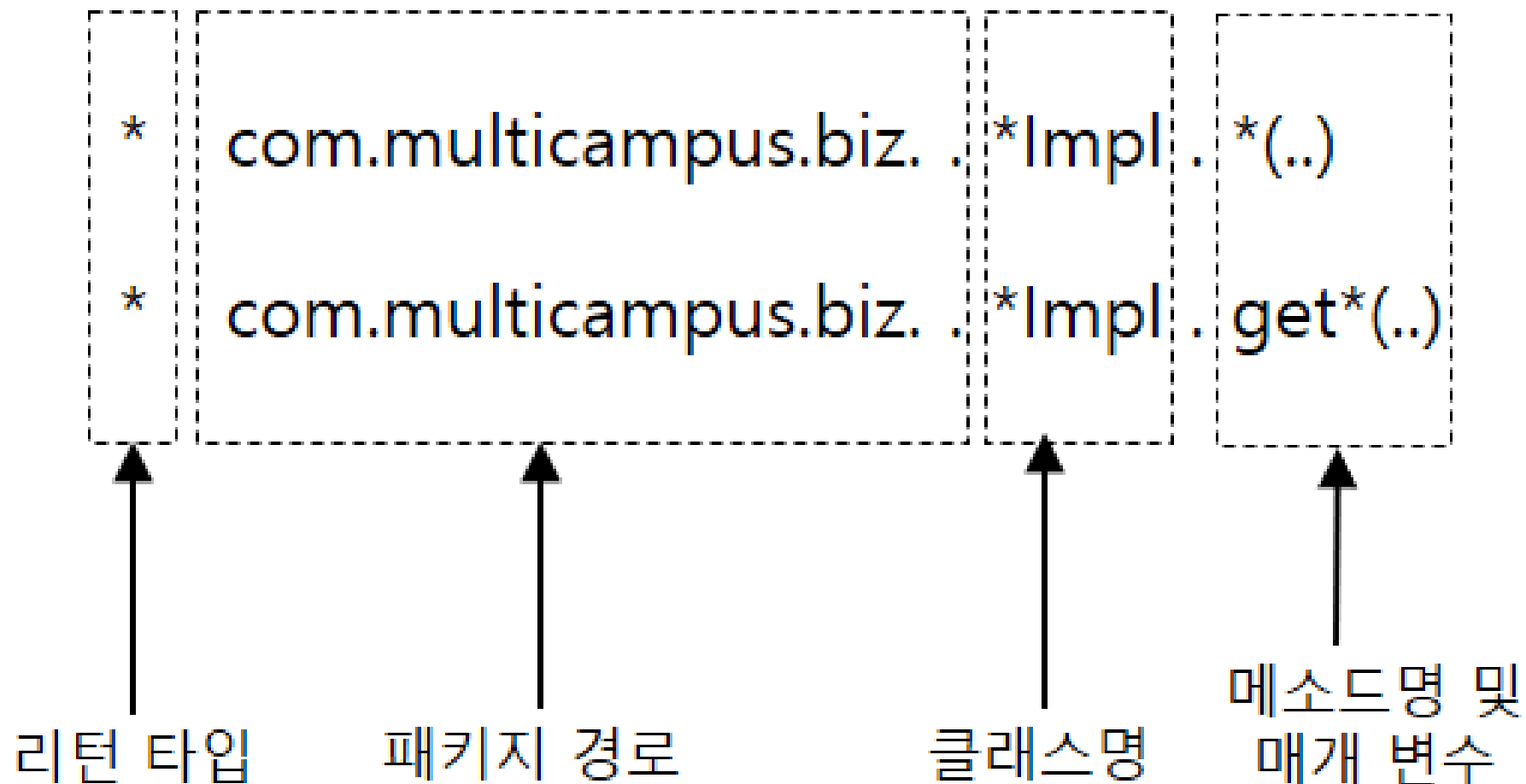
- 클라이언트가 호출하는 모든 비즈니스 메소드

- 포인트컷 (*Pointcut*)

- 필터링된 조인포인트

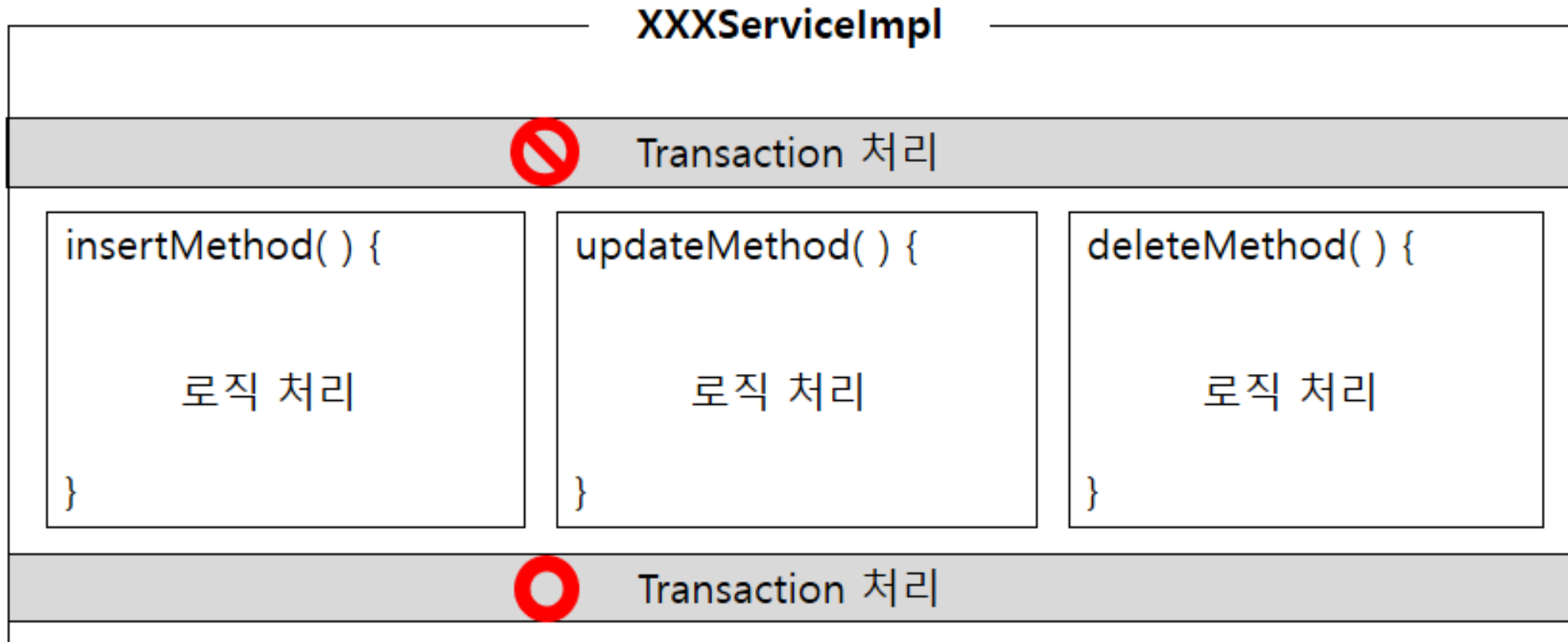


- Pointcut 설정 예시



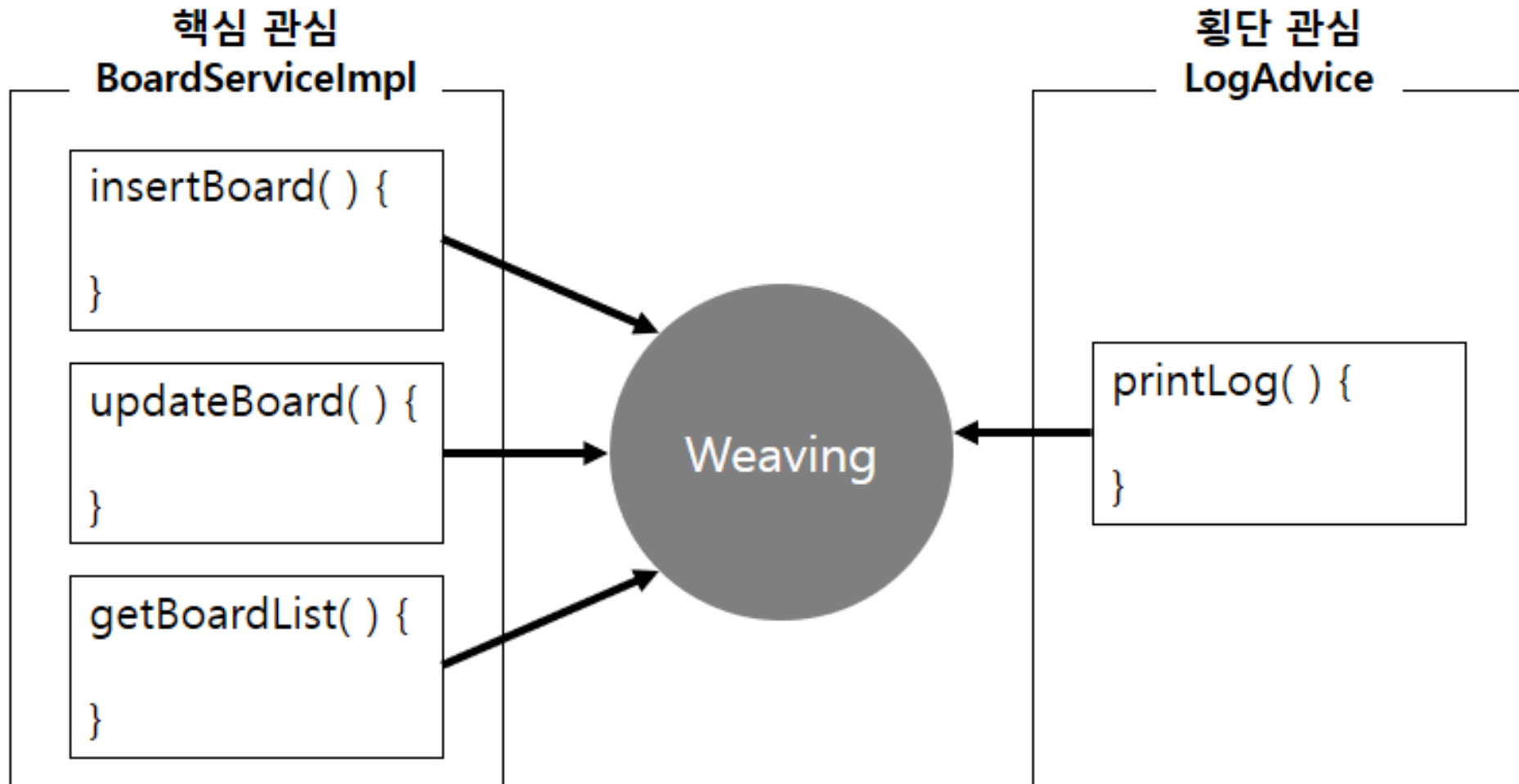
- 어드바이스 (*Advice*)

- 횡단 관심에 해당하는 공통 기능의 코드
- 어드바이스 동작 시점을 5가지로 지정



- 위빙 (*Weaving*)

- 포인트컷으로 지정한 핵심 관심 메소드가 호출될 때, 어드바이스에 해당하는 횡단 관심 메소드를 결합하는 것



- 애스팩트 (*Aspect*) 또는 어드바이저 (*Advisor*)
 - 포인트컷과 어드바이저의 결합
 - 애스팩트 설정에 따라 AOP의 동작 방식이 결정

- Aspect

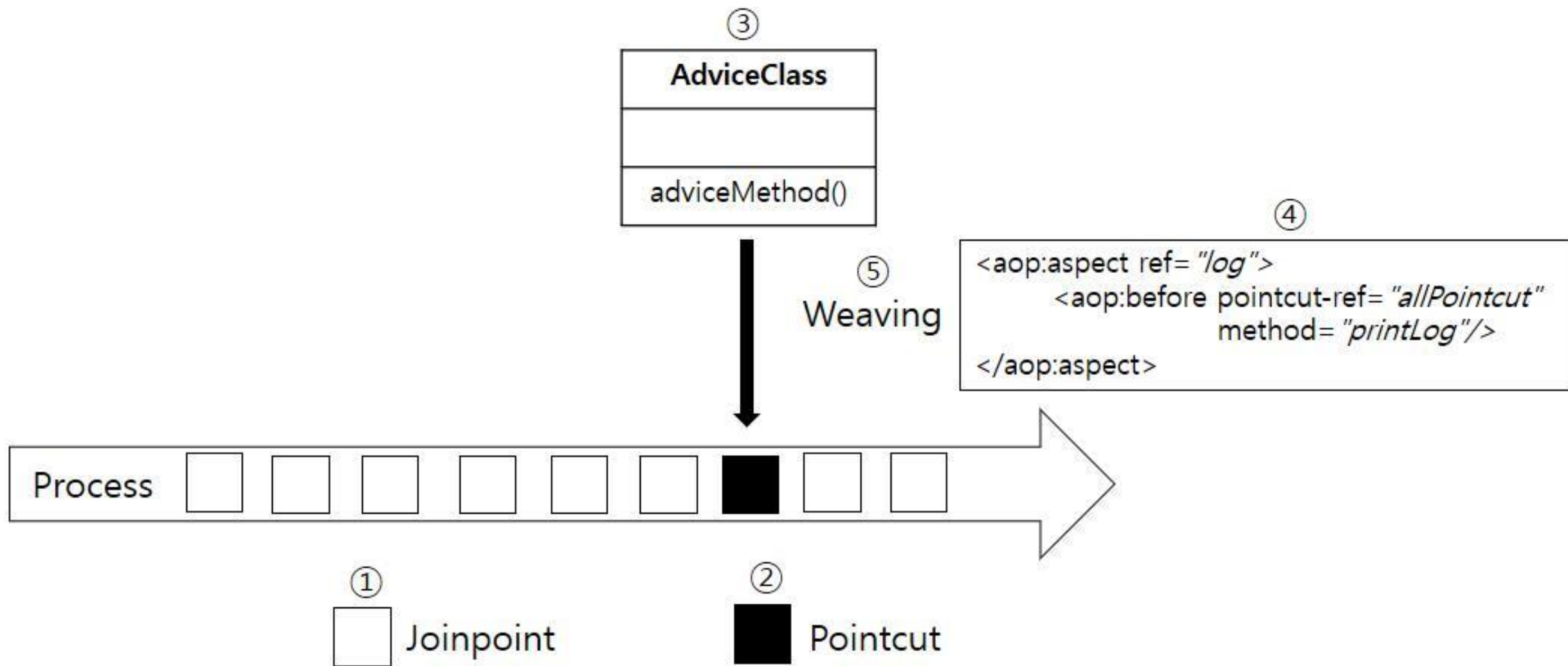
LogAdvice.java

```
public class LogAdvice {  
    public void printLog() {  
        System.out.println("[공통 로그] 비즈니스 로직 수행 전 동작");  
    }  
}
```

applicationContext.xml

```
<bean id="log" class="com.springbook.biz.common.LogAdvice"></bean>  
  
<aop:config>  
    <aop:pointcut id="allPointcut" expression="execution(* com.springbook.biz..*Impl.*(..))"/>  
  
    <aop:pointcut id="getPointcut" expression="execution(* com.springbook.biz..*Impl.get*(..))"/>  
    <aop:aspect ref="log">  
        <aop:before pointcut-ref="getPointcut" method="printLog"/>  
  
    </aop:aspect>  
</aop:config>
```

- AOP 용어 정리



- AOP 설정 Elements

applicationContext.xml

```
<beans xmlns="http://www.springframework.org/schema/beans" ...>
```

```
  <aop:config>
```

```
    <aop:pointcut .../>
```

```
    <aop:aspect ...></aop:aspect>
```

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

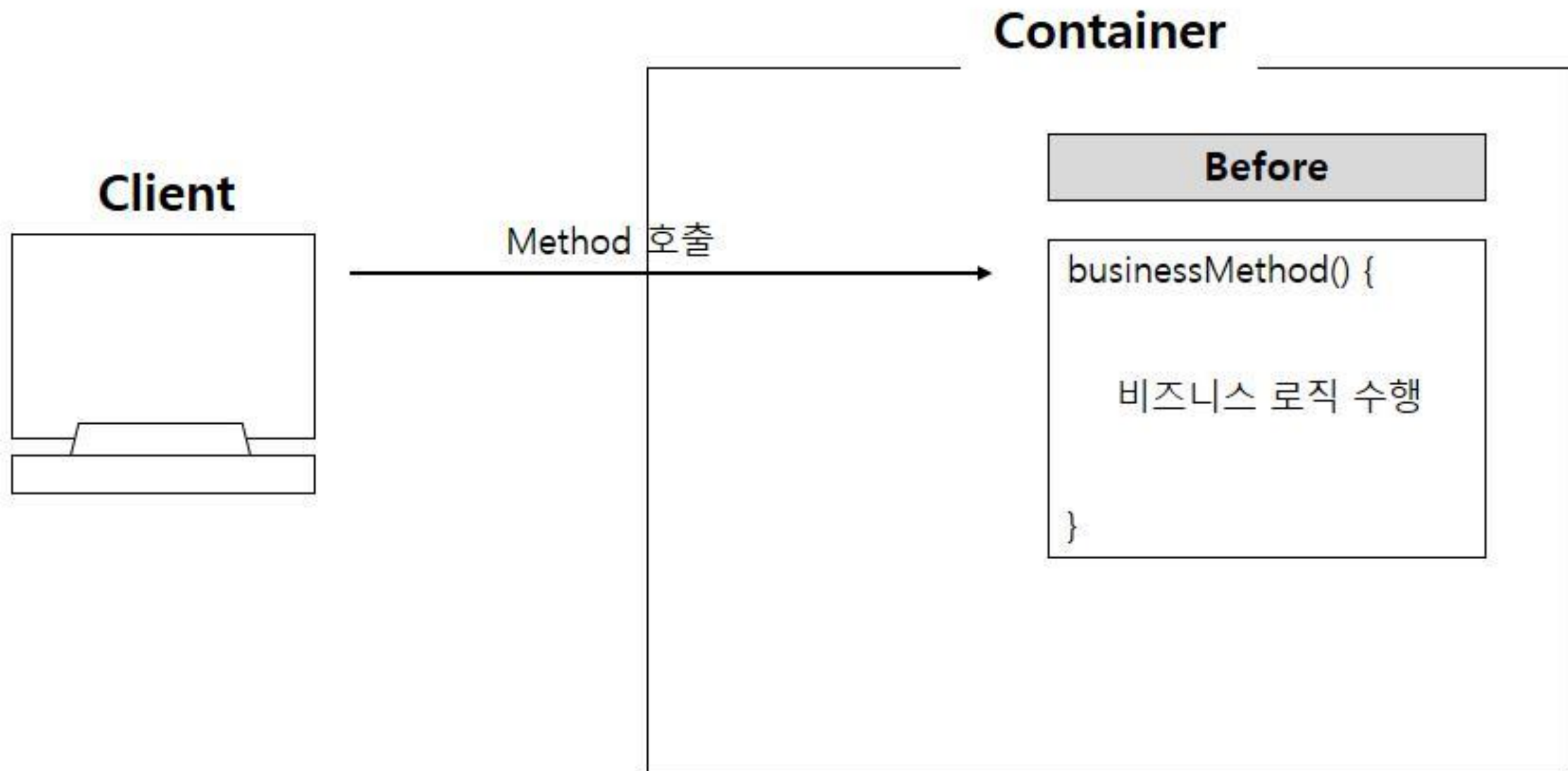
```
</beans>
```

Advice 동작 시점

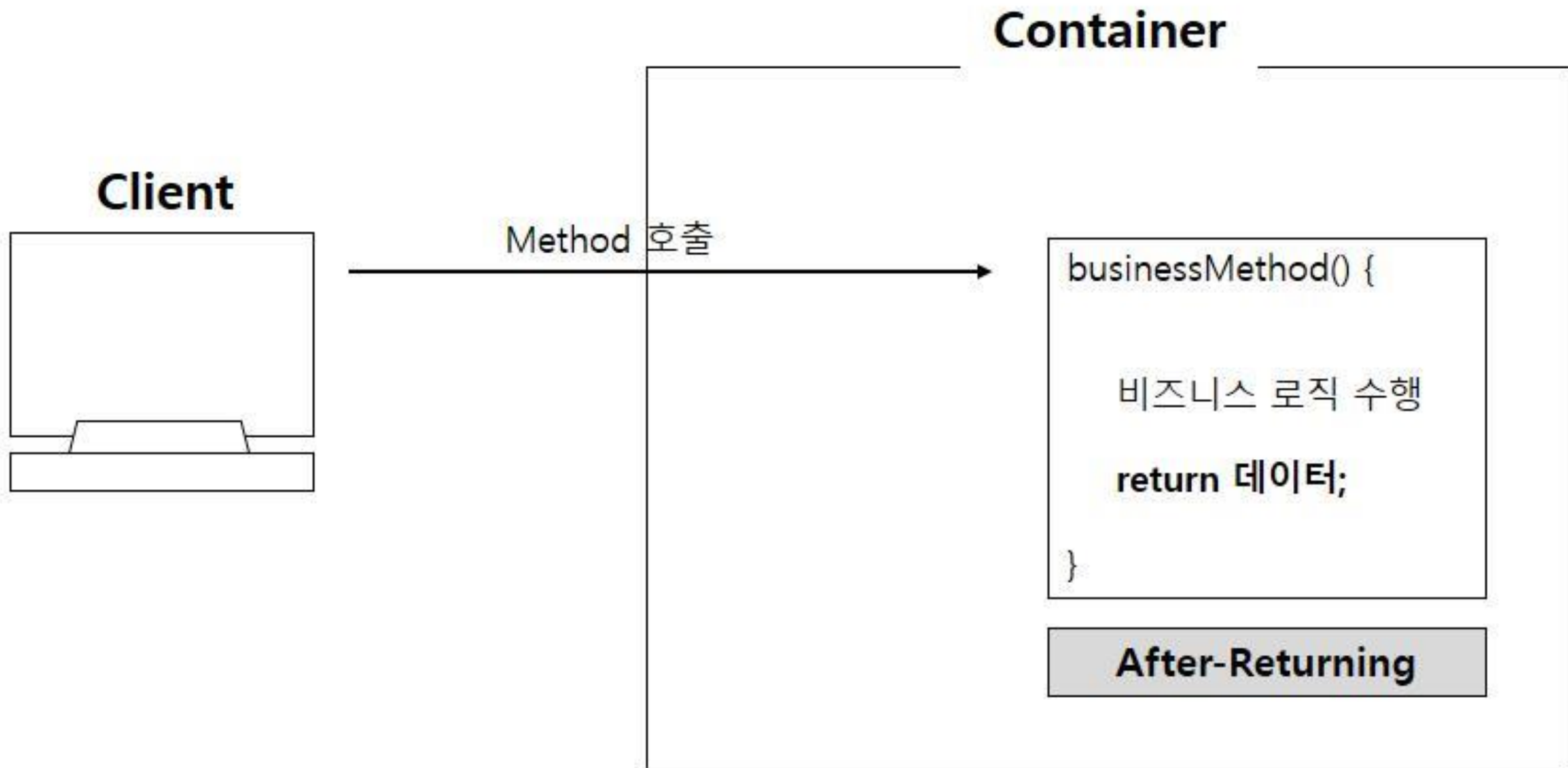
- Advice 동작 시점

동작 시점	설 명
Before	비즈니스 메소드 실행 전 동작
After	<ul style="list-style-type: none">- After Returning: 비즈니스 메소드가 성공적으로 반환되면 동작- After Throwing: 비즈니스 메소드 실행 중 예외가 발생하면 동작 (try~catch 블록에서 catch 블록에 해당)- After: 비즈니스 메소드가 실행된 후, 무조건 실행 (try~catch~finally 블록에서 finally 블록에 해당)
Around	Around는 메소드 호출 자체를 가로채 비즈니스 메소드 실행 전후에 처리할 로직을 삽입할 수 있음

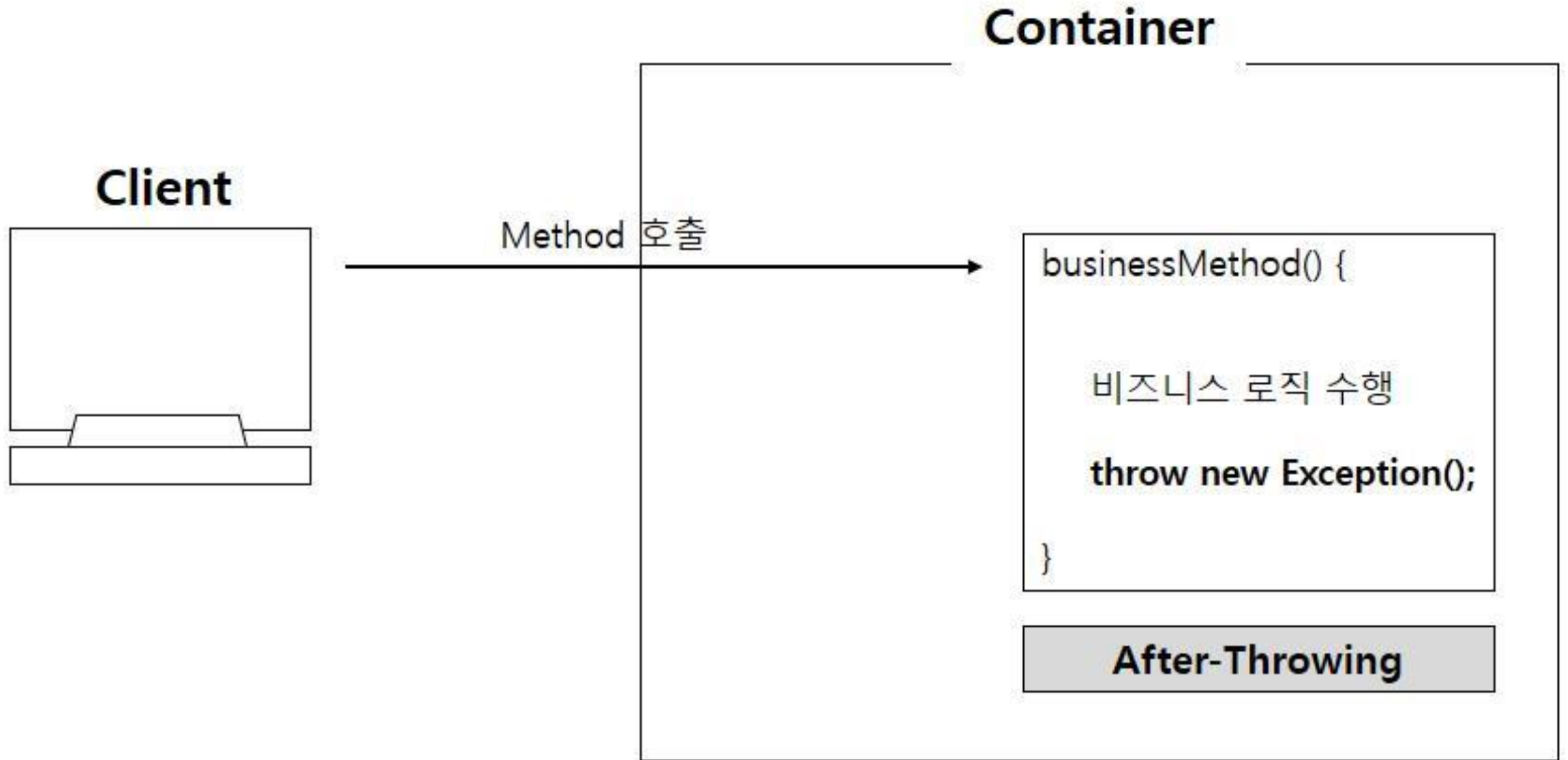
- Before 어드바이스



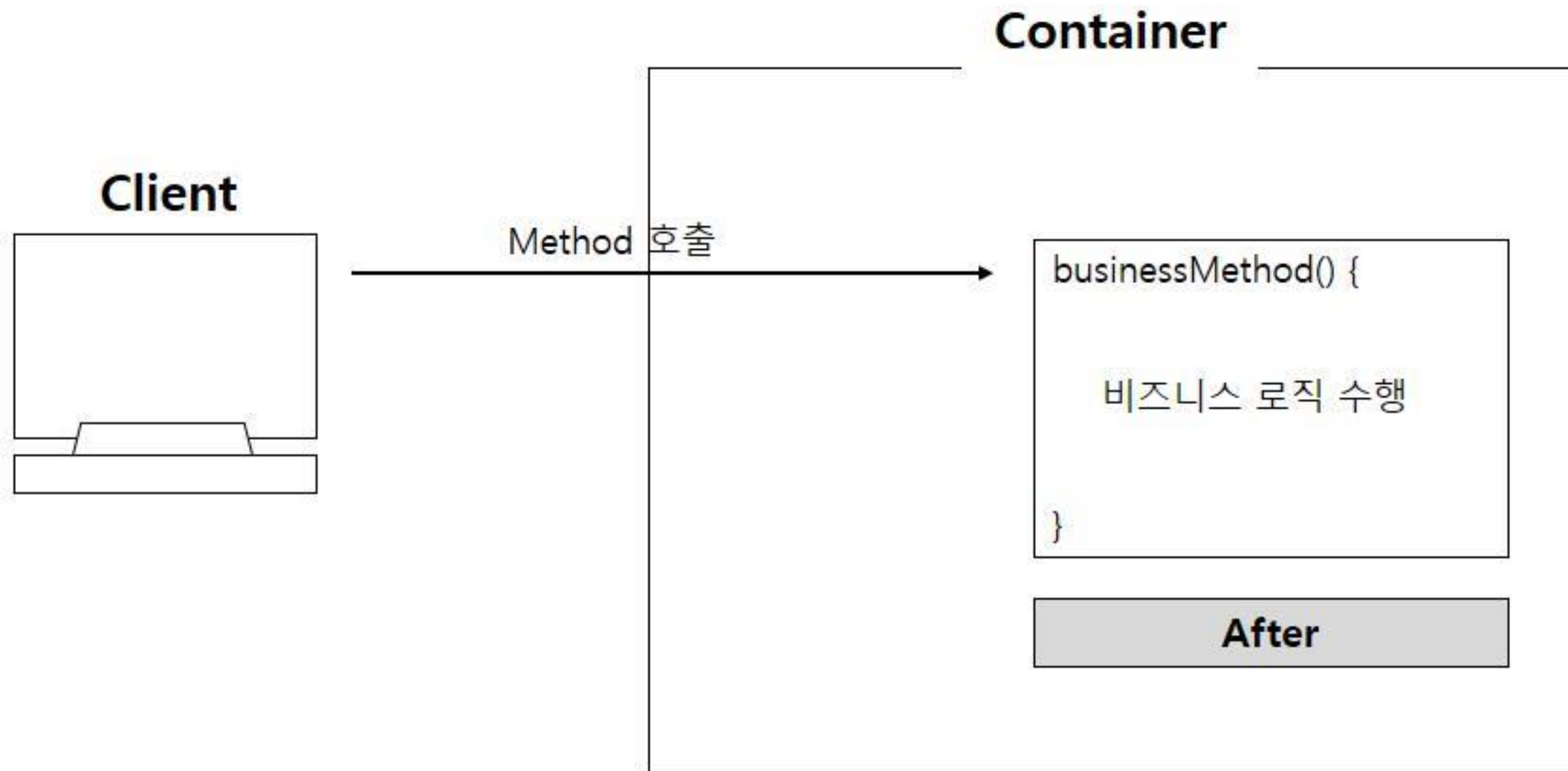
- After Returning 어드바이스



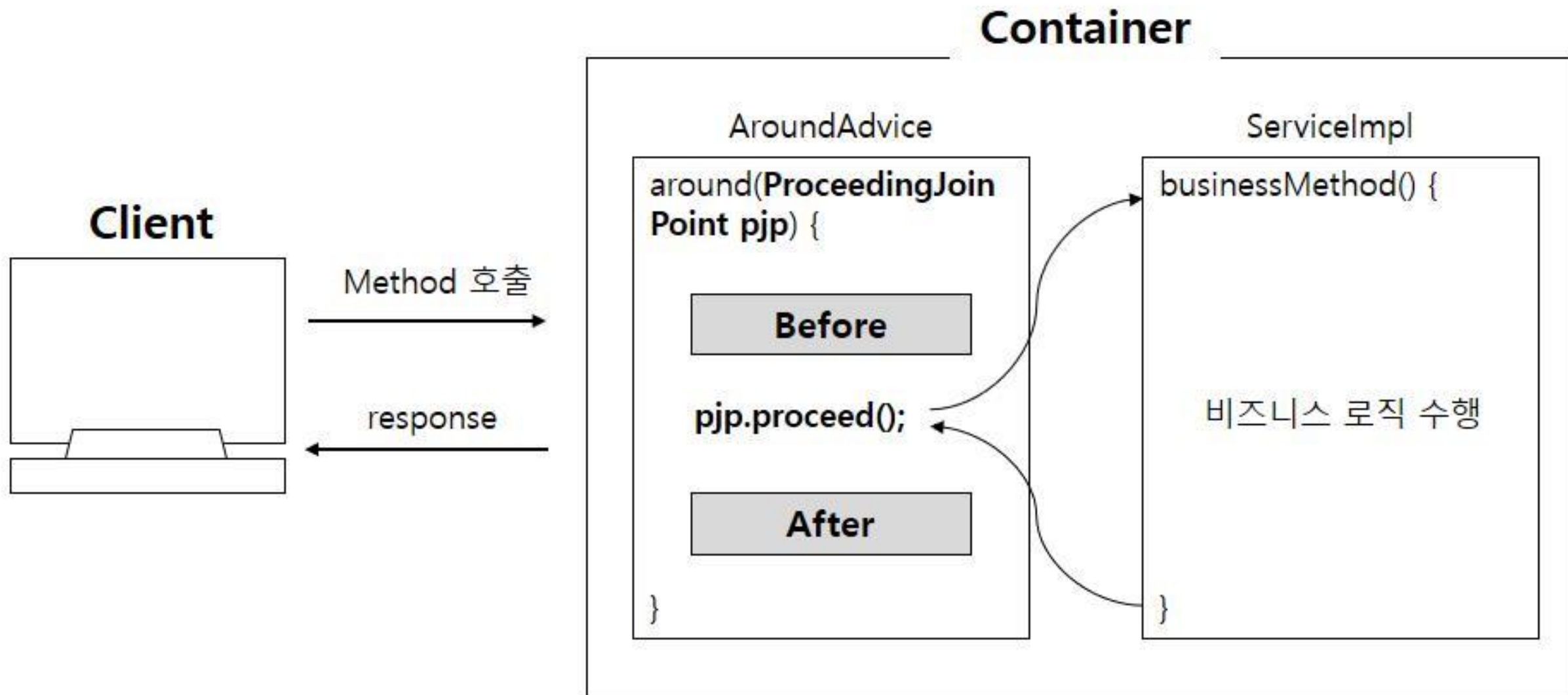
- After Throwing 어드바이스



- After 어드바이스



- Around 어드바이스



- JoinPoint 메소드

메소드	설 명
Signature getSignature()	클라이언트가 호출한 메소드의 시그니처(반환형, 이름, 매개 변수) 정보가 저장된 Signature 객체 반환
Object getTarget()	클라이언트가 호출한 비즈니스 메소드를 포함하는 비즈니스 객체 반환
Object[] getArgs()	클라이언트가 메소드를 호출할 때 넘겨준 인자 목록을 Object 배열로 반환

- ProceedingJoinPoint

- ProceedingJoinPoint는 JoinPoint를 상속했다.
- ProceedingJoinPoint는 proceed() 메소드가 추가되어 있다.
- Around로 동작하는 어드바이스 메소드는 반드시 ProceedingJoinPoint를 매개 변수로 받아야 한다.

AOP 설정

(by Annotation)

- Annotation 설정

```
<beans xmlns="http://www.springframework.org/schema/beans"  
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
        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-4.2.xsd">
```

```
    <aop:aspectj-autoproxy> </aop:aspectj-autoproxy>
```

```
</beans>
```

- Pointcut 설정

@Service

```
public class LogAdvice {
```

```
    @Pointcut("execution(* com.springbook.biz..*Impl.*(..))")
```

```
    public void allPointcut() {}
```

```
    @Pointcut("execution(* com.springbook.biz..*Impl.get*(..))")
```

```
    public void getPointcut() {}
```

```
}
```

참조용 메소드

- Advice 설정

어노테이션	설 명
@Before	비즈니스 메소드 실행 전에 동작
@AfterReturning	비즈니스 메소드가 성공적으로 반환되면 동작
@AfterThrowing	비즈니스 메소드 실행 중 예외가 발생하면 동작 (마치 try~catch 블록에서 catch 블록에 해당).
@After	비즈니스 메소드가 실행된 후, 무조건 실행 (try~catch~finally 블록에서 finally 블록에 해당)
@Around	호출 자체를 가로채 비즈니스 메소드 실행 전후에 처리할 로직을 삽입할 수 있음

- Aspect 설정

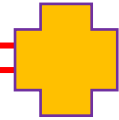
```
@Service
```

```
@Aspect // Aspect = Pointcut + Advice
```

```
public class LogAdvice {
```

```
@Pointcut("execution(* com.springbook.biz..*Impl.*(..))")
```

```
public void allPointcut() {}
```



```
@Before("allPointcut()")
```

```
public void printLog() {
```

```
    System.out.println("[공통 로그] 비즈니스 로직 수행 전 동작");
```

```
}
```

```
}
```

- Before Advice

@Service

@Aspect

```
public class BeforeAdvice {
```

```
    @Pointcut("execution(* com.springbook.biz..*Impl.*(..))")
```

```
    public void allPointcut() {}
```

```
    @Before("allPointcut()")
```

```
    public void beforeLog(JoinPoint jp) {
```

```
        String method = jp.getSignature().getName();
```

```
        Object[] args = jp.getArgs();
```

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

```
    }
```

```
}
```

- AfterReturning Advice

@Service

@Aspect

```
public class AfterReturningAdvice {
```

```
    @Pointcut("execution(* com.springbook.biz.*Impl.get*(..))")
```

```
    public void getPointcut() {}
```

```
    @AfterReturning(pointcut="getPointcut()", returning="returnObj")
```

```
    public void afterLog(JoinPoint jp, Object returnObj) {
```

```
        String method = jp.getSignature().getName();
```

```
        System.out.println("[사후 처리] " + method +
```

```
            "() 메소드 리턴값 : " + returnObj.toString());
```

```
    }
```

```
}
```

- AfterThrowing Advice

@Service

@Aspect

```
public class AfterThrowingAdvice {
```

```
    @Pointcut("execution(* com.springbook.biz..*Impl.*(..))")
```

```
    public void allPointcut() {}
```

```
    @AfterThrowing(pointcut="allPointcut()", throwing="exceptObj")
```

```
    public void exceptionLog(JoinPoint jp, Exception exceptObj) {
```

```
        String method = jp.getSignature().getName();
```

```
        System.out.println(method + "() 메소드 수행 중 예외 발생!");
```

```
        if(exceptObj instanceof IllegalArgumentException) {
```

```
            System.out.println("부적합한 값이 입력되었습니다.");
```

```
        }
```

```
    }
```

```
}
```


- After Advice

@Service

@Aspect

```
public class AfterAdvice {
```

```
    @Pointcut("execution(* com.springbook.biz..*Impl.*(..))")
```

```
    public void allPointcut() {}
```

```
    @After("allPointcut()")
```

```
    public void finallyLog() {
```

```
        System.out.println("[사후 처리] 비즈니스 로직 수행 후 무조건 동작");
```

```
    }
```

```
}
```

- Around Advice

@Service

@Aspect

```
public class AroundAdvice {
```

```
    @Pointcut("execution(* com.springbook.biz..*Impl.*(..))")
```

```
    public void allPointcut(){}  
  
    @Around("allPointcut()")
```

```
    public Object aroundLog(ProceedingJoinPoint pjp) throws Throwable {
```

```
        String method = pjp.getSignature().getName();
```

```
        System.out.println(method + "() 메소드 수행에 걸린 시간 : "
```

```
        + stopWatch.getTotalTimeMillis() + "(ms)초");
```

```
        return obj;
```

```
    }
```

```
}
```

- 외부 Pointcut 참조

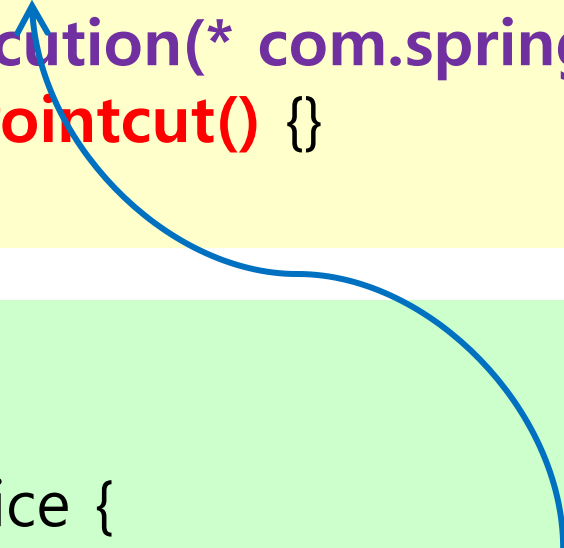
@Aspect

```
public class PointcutCommon {  
    @Pointcut("execution(* com.springbook.biz.*Impl.*(..))")  
    public void allPointcut() {}  
    @Pointcut("execution(* com.springbook.biz.*Impl.get*(..))")  
    public void getPointcut() {}  
}
```

@Service

@Aspect

```
public class BeforeAdvice {  
    @Before("PointcutCommon.allPointcut()")  
    public void beforeLog(JoinPoint jp) {  
  
    }  
}
```

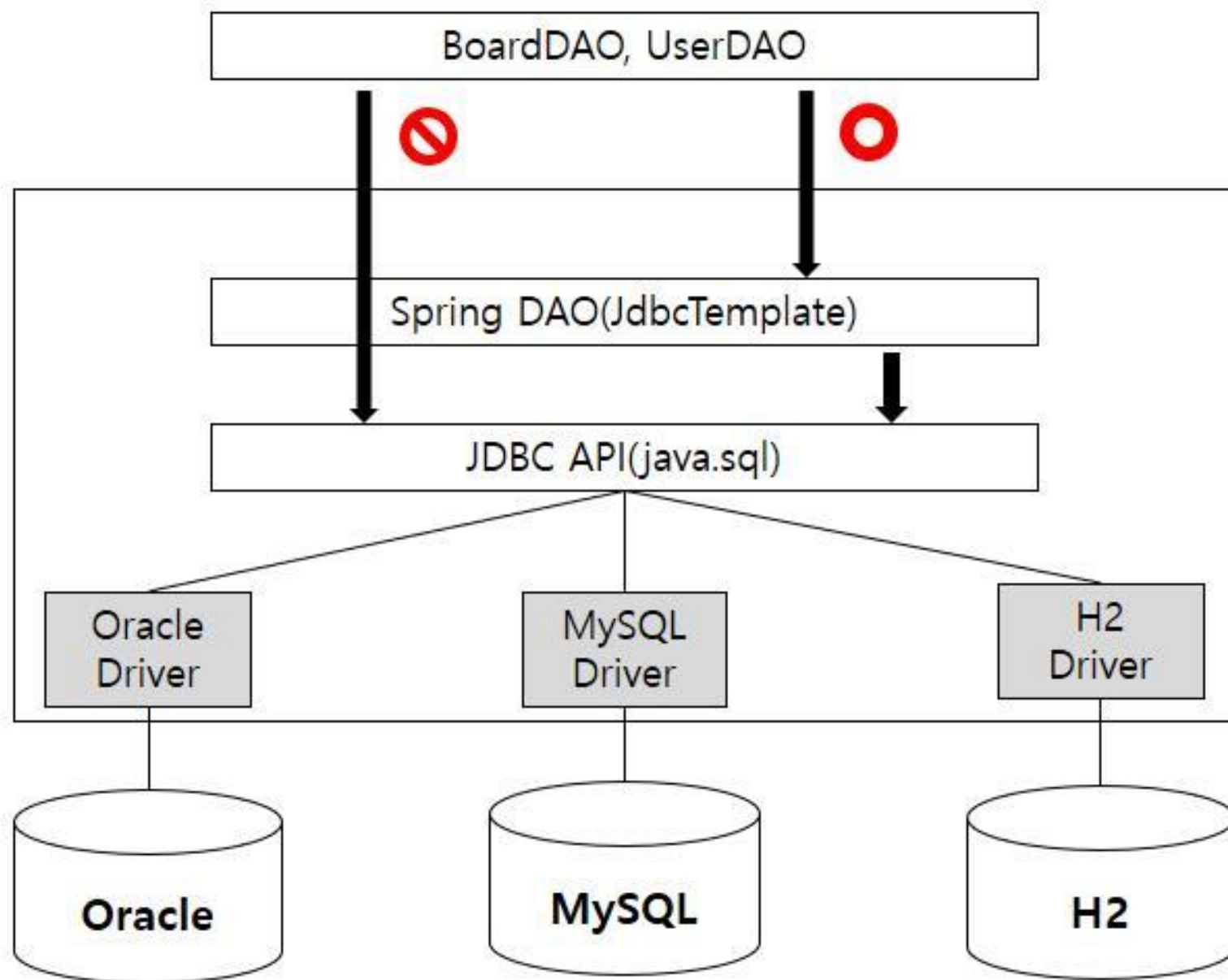


Spring DAO

- JDBC 프로그램의 문제

- 1. 정해진 순서대로 프로그램을 구현해야 한다.
- 2. 모든 DB 연동 메소드에서 동일한 로직이 반복적으로 등장한다.
- 3. 많은 코드로 인해 실수할 가능성이 높다.
- 4. 결과적으로 유지보수가 어렵다.

- Spring JDBC 구조



- DataSource 설정 (1)

```
<!-- DataSource 설정 -->
```

```
<bean id="dataSource"
```

```
    class="org.apache.commons.dbcp.BasicDataSource"
```

```
    destroy-method="close">
```

```
    <property name="driverClassName" value="org.h2.Driver" />
```

```
    <property name="url" value="jdbc:h2:tcp://localhost/~ /test" />
```

```
    <property name="username" value="sa" />
```

```
    <property name="password" value="" />
```

```
</bean>
```

- DataSource 설정 (2)

<!-- DataSource 설정 -->

<context:property-placeholder location="classpath:database.properties" />

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">

<property name="driverClassName" value="**\${jdbc.driver}**" />

<property name="url" value="**\${jdbc.url}**" />

<property name="username" value="**\${jdbc.username}**" />

<property name="password" value="**\${jdbc.password}**" />

</bean>

database.properties

jdbc.driver=org.h2.Driver

jdbc.url=jdbc:h2:tcp://localhost/~ /test

jdbc.username=sa

jdbc.password=

- JdbcTemplate 메소드

메소드	설 명
update()	INSERT, UPDATE, DELETE 명령어를 처리한다.
queryForInt()	정수 하나를 검색하여 리턴한다.
queryForObject()	검색 결과를 객체(Value Object) 하나에 매핑하여 리턴한다.
query()	검색 결과 여러 개를 java.util.List에 저장하여 리턴한다.

DAO 클래스 구현

- DAO 클래스 구현

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

```
@Repository  
public class BoardDAOSpring {  
    @Autowired  
    private JdbcTemplate spring;  
  
    // 글 상세 조회  
    public BoardVO getBoard(BoardVO vo){  
        Object[] args = {vo.getSeq()};  
        return spring.queryForObject(  
            "select * from board where seq=?", args, new BoardRowMapper());  
    }  
}
```

- Transaction 설정 (1) : 네임스페이스 추가

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.2.xsd">

</beans>
```

- Transaction 설정 (2) : TransactionManager 등록

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

```
<!-- Transaction 설정 -->
<bean id="txManager"
```

```
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"> </property>
</bean>
```

- Transaction 설정 (3) : Transaction Advice 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="get*" read-only="true"/>  
    <tx:method name="*"/>  
  </tx:attributes>  
</tx:advice>
```

- Transaction 설정 (4)

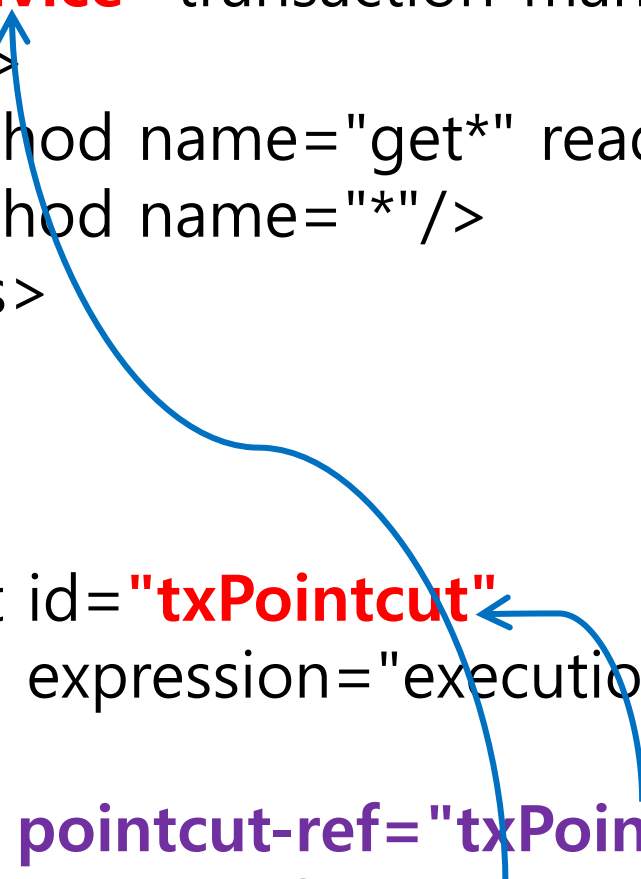
속 성	의 미
name	트랜잭션이 적용될 메소드 이름 지정
read-only	읽기 전용 여부 지정(기본값 false)
no-rollback-for	트랜잭션을 롤백하지 않을 예외 지정
rollback-for	트랜잭션을 롤백할 예외 지정

- Transaction 설정 (5) : AOP 설정

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>

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

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



- Transaction 설정

