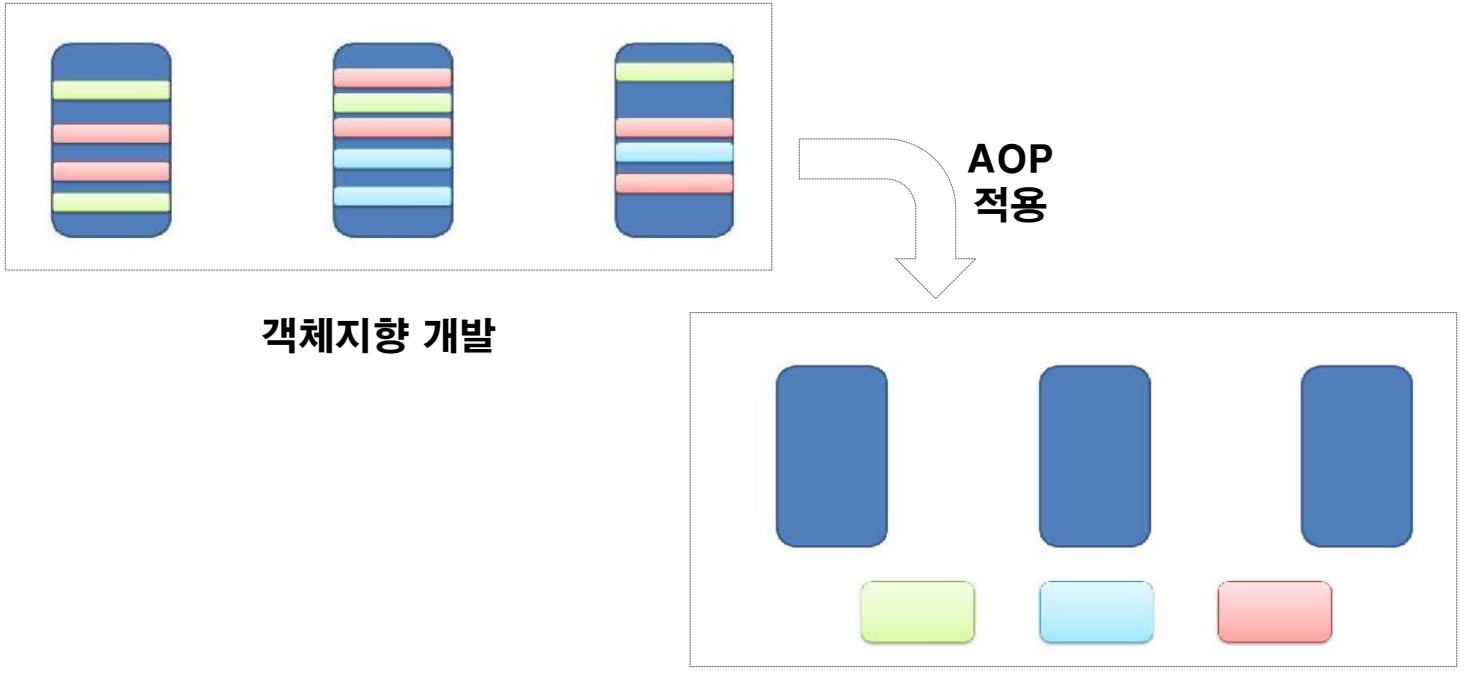
❑ 서비스 개요

– 객체지향 프로그래밍(Object Oriented Programming)을 보완하는 개념으로 어플리케이션을 객체지향적으로 모듈화 하여 작성하더라도 다수의 객체들에 분산되어 중복적으로 존재하는 공통 관심사가 여전히 존재한다. AOP는 이를 횡단관심으로 분리하여 핵심관심과 엮어서 처리할 수 있는 방법을 제공한다.

– 로깅, 보안, 트랜잭션 등의 공통적인 기능의 활용을 기존의 비즈니스 로직에 영향을 주지 않고 모듈화 처리를 지원하는 프로그래밍 기법



❑ 주요 개념

– Join Point

- 횡단 관심(Crosscutting Concerns) 모듈이 삽입되어 동작할 수 있는 실행 가능한 특정 위치를 말함
- **메소드 호출, 메소드 실행 자체, 클래스 초기화, 객체 생성 시점 등**

– Pointcut

- Pointcut은 어떤 클래스의 어느 JoinPoint를 사용할 것인지를 결정하는 선택 기능을 말함
- **가장 일반적인 Pointcut은 '특정 클래스에 있는 모든 메소드 호출'로 구성된다.**

– 애스펙트(Aspect)

- **Advice와 Pointcut의 조합**
- 어플리케이션이 가지고 있어야 할 로직과 그것을 실행해야 하는 지점을 정의한 것

– Advice

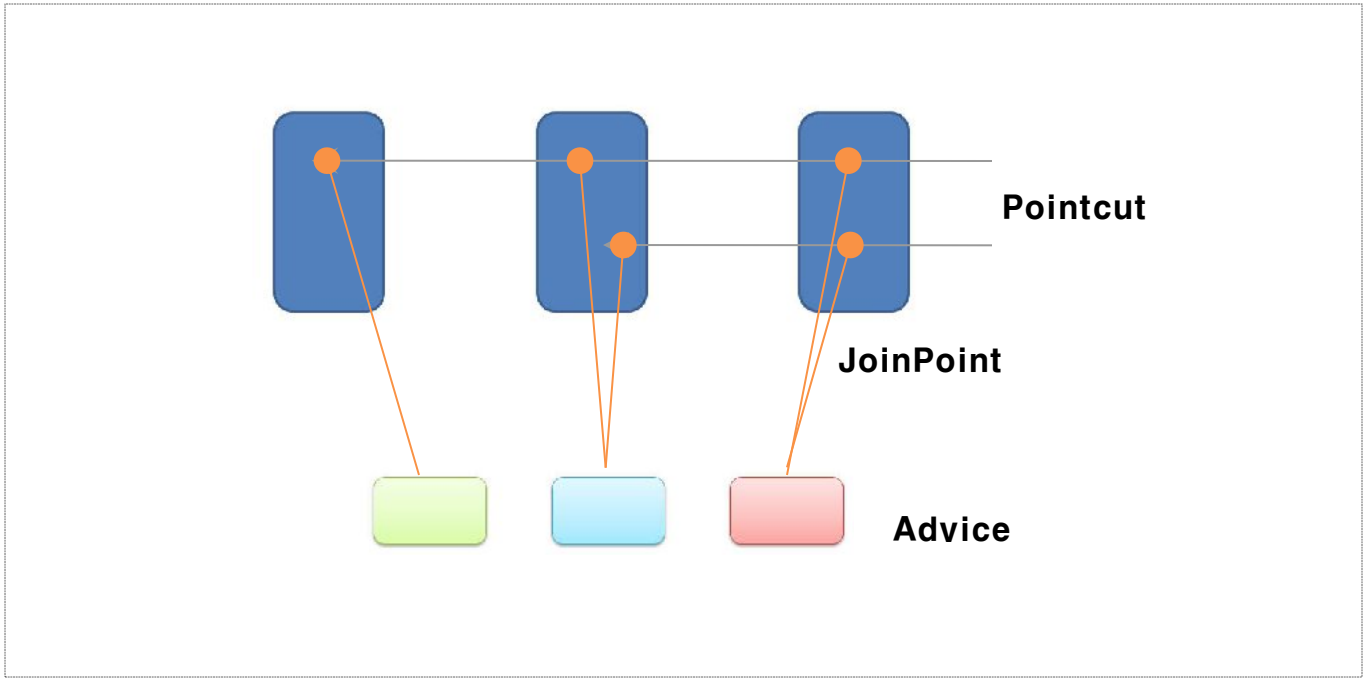
- Advice는 관점(Aspect)의 실제 구현체로 결합점에 삽입되어 동작할 수 있는 코드이다
- Advice 는 결합점(JoinPoint)과 결합하여 동작하는 시점에 따라 before advice, after advice, around advice 타입으로 구분된다
- **특정 Join point에 실행하는 코드**

– Weaving

- Pointcut에 의해서 결정된 JoinPoint에 지정된 Advice를 삽입하는 과정
- Weaving은 AOP가 기존의 Core Concerns 모듈의 코드에 전혀 영향을 주지 않으면서 필요한 Crosscutting Concerns 기능을 추가할 수 있게 해 주는 핵심적인 처리 과정임

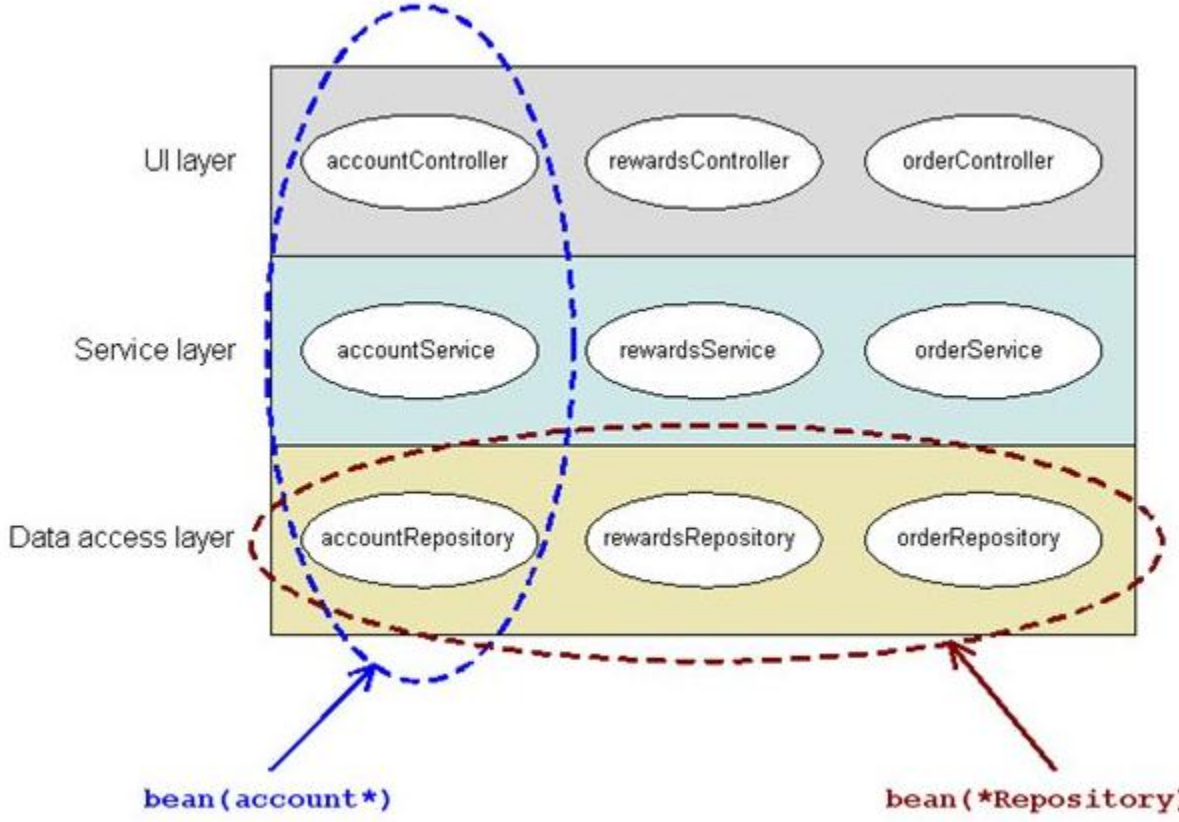
❑ 주요 개념

– Advice, JoinPoint, Pointcut



❑ 주요 개념

– Pointcut 예제 - bean() Pointcut을 이용하여 종적 및 횡적으로 빈을 선택



❑ 주요 개념

– Weaving 방식

- 컴파일 시 엮기: 별도 컴파일러를 통해 핵심 관심사 모듈의 사이 사이에 관점(Aspect) 형태로 만들어진 횡단 관심사 코드들이 삽입되어 관점(Aspect)이 적용된 최종 바이너리가 만들어지는 방식이다. (ex. AspectJ, …)
- 클래스 로딩 시 엮기: 별도의 Agent를 이용하여 JVM이 클래스를 로딩할 때 해당 클래스의 바이너리 정보를 변경한다. 즉, Agent가 횡단 관심사 코드가 삽입된 바이너리 코드를 제공함으로써 AOP를 지원하게 된다. (ex. AspectWerkz, …)
- 런타임 엮기: 소스 코드나 바이너리 파일의 변경없이 프록시를 이용하여 AOP를 지원하는 방식이다. 프록시를 통해 핵심 관심사를 구현한 객체에 접근하게 되는데, 프록시는 핵심 관심사 실행 전후에 횡단 관심사를 실행한다. 따라서 프록시 기반의 런타임 엮기의 경우 메소드 호출시에만 AOP를 적용할 수 있다는 제한점이 있다. (ex. Spring AOP, …)

– **Advice 결합점 결합 타입**

- Before advice: joinpoint 전에 수행되는 advice
- After returning advice: joinpoint가 성공적으로 리턴된 후에 동작하는 advice
- After throwing advice: 예외가 발생하여 joinpoint가 빠져나갈때 수행되는 advice
- After (finally) advice: join point를 빠져나가는(정상적이거나 예외적인 반환) 방법에 상관없이 수행되는 advice
- Around advice: joinpoint 전, 후에 수행되는 advice

❑ 주요 기능

- 횡단 관심(CrossCutting Concern) 모듈이 삽입되어 동작할 수 있도록 지정하는 JointPoint 기능
- 횡단 관심 모듈을 특정 JointPoint에 사용할 수 있도록 지정하는 Pointcut 기능
- Pointcut 지정을 위한 패턴 매칭 표현식
- Pointcut에서 수행해야하는 동작을 지정하는 Advice 기능
- Pointcut에 의해서 결정된 JoinPoint에 지정된 Advice를 삽입하여 실제 AOP 방식대로 동작

❑ 장점

- 중복 코드의 제거
  - 횡단 관심(CrossCutting Concerns)을 여러 모듈에 반복적으로 기술되는 현상을 방지
- 비즈니스 로직의 가독성 향상
  - 핵심기능 코드로부터 횡단 관심 코드를 분리함으로써 비즈니스 로직의 가독성 향상
- 생산성 향상
  - 비즈니스 로직의 독립으로 인한 개발의 집중력을 높임
- 재사용성 향상
  - 횡단 관심 코드는 여러 모듈에서 재사용될 수 있음
- 변경 용이성 증대
  - 횡단 관심 코드가 하나의 모듈로 관리되기 때문에 이에 대한 변경 발생시 용이하게 수행할 수 있음

❑ Spring의 AOP 지원

- 스프링은 프록시 기반의 런타임 Weaving 방식을 지원한다.
- 스프링은 AOP 구현을 위해 다음 세가지 방식을 제공한다.
  - @AspectJ **어노테이션을 이용한 AOP 구현**
  - **XML Schema를 이용한 AOP 구현**
  - 스프링 API를 이용한 AOP 구현
- 실행환경은 XML Schema를 이용한 AOP 구현 방법을 사용한다.

❑ XML 스키마를 이용한 AOP 지원 (1/11)

– 개요

- Java 5 버전을 사용할 수 없거나, XML 기반 설정을 선호하는 경우 사용한다.
- AOP 선언을 한 눈에 파악할 수 있다.
- 실행환경은 XML 스키마를 이용한 AOP를 사용한다.

❑ XML 스키마를 이용한 AOP 지원 (2/11)

– Aspect 정의하기

Advice 정의

Pointcut 정의

JoinPoint 정의

Aspect 정의

```
<bean id="adviceUsingXML" class="egovframework.rte.fdl.aop.sample.AdviceUsingXML" />

<aop:config>
    <aop:pointcut id="targetMethod"
     expression="execution(* egovframework.rte.fdl.aop.sample.*Sample.*(..))" />
    <aop:aspect ref="adviceUsingXML">
        <aop:before pointcut-ref="targetMethod" method="beforeTargetMethod" />
        <aop:after-returning pointcut-ref="targetMethod"
         method="afterReturningTargetMethod" returning="retVal" />
        <aop:after-throwing pointcut-ref="targetMethod"
         method="afterThrowingTargetMethod" throwing="exception" />
        <aop:after pointcut-ref="targetMethod" method="afterTargetMethod" />
        <aop:around pointcut-ref="targetMethod" method="aroundTargetMethod" />
    </aop:aspect>
</aop:config>

<bean id="adviceSample" class="egovframework.rte.fdl.aop.sample.AdviceSample" />
```

❑ XML 스키마를 이용한 AOP 지원(3/11)

– Advice 정의하기 – before advice

```
public class AdviceUsingXML {

    public void beforeTargetMethod(JoinPoint thisJoinPoint) {
        Class clazz = thisJoinPoint.getTarget().getClass();
        String className = thisJoinPoint.getTarget().getClass().getSimpleName();
        String methodName = thisJoinPoint.getSignature().getName();

        // 대상 메서드에 대한 로거를 얻어 해당 로거로 현재 class, method 정보 로깅
        Log logger = LogFactory.getLog(clazz);
        logger.debug(className + "." + methodName + " executed.");

    }
    …
}
```

❑ XML 스키마를 이용한 AOP 지원(4/11)

– Advice 정의하기 – After returning advice

- After returing advice는 정상적으로 메소드가 실행될 때 수행된다.

```
public class AdviceUsingXML {

    public void afterReturningTargetMethod(JoinPoint thisJoinPoint,
            Object retVal) {
        System.out.println("AspectUsingAnnotation.afterReturningTargetMethod executed.” +
                " return value is [" + retVal + "]");
    }    …
}
```

❑ XML 스키마를 이용한 AOP 지원(5/11)

– Advice 정의하기 – After throwing advice

- After throwing advice 는 메소드가 수행 중 예외사항을 반환하고 종료하는 경우 수행된다.

```
public class AdviceUsingXML {
  ...
  public void afterThrowingTargetMethod(JoinPoint thisJoinPoint,
              Exception exception) throws Exception{
    System.out.println("AdviceUsingXML.afterThrowingTargetMethod executed.");
    System.err.println("에러가 발생했습니다.", exception);
    throw new BizException("에러가 발생했습니다.", exception);
  }
     ...
}
```

❑ XML 스키마를 이용한 AOP 지원(6/11)

– Advice 정의하기 – After (finally) advice

- After (finally) advice 는 메소드 수행 후 무조건 수행된다.
- After advice 는 정상 종료와 예외 발생 경우를 모두 처리해야 하는 경우에 사용된다 (예:리소스 해제 작업)

```
public class AdviceUsingXML {

    public void afterTargetMethod(JoinPoint thisJoinPoint) {
        System.out.println("AspectUsingAnnotation.afterTargetMethod executed.");
    }
    …
}
```

❑ XML 스키마를 이용한 AOP 지원(7/11)

– Advice 정의하기 – Around advice

- Around advice 는 메소드 수행 전후에 수행된다.
- Return값을 가공하기 위해서는 Around를 사용해야한다.

```
public class AdviceUsingXML {

    public Object aroundTargetMethod(ProceedingJoinPoint thisJoinPoint)
            throws Throwable {
        System.out.println("AspectUsingAnnotation.aroundTargetMethod start.");
        long time1 = System.currentTimeMillis();
        Object retVal = thisJoinPoint.proceed();

        System.out.println("ProceedingJoinPoint executed. return value is [" + retVal + "]");

        retVal = retVal + "(modified)";
        System.out.println("return value modified to [" + retVal + "]");

        long time2 = System.currentTimeMillis();
        System.out.println("AspectUsingAnnotation.aroundTargetMethod end. Time("
                + (time2 - time1) + ")");
        return retVal;
    }
    …
}
```

❑ XML 스키마를 이용한 AOP 지원(8/11)

– Aspect 실행하기 – 정상 실행의 경우

```
public class AnnotationAspectTest {

    @Resource(name = "adviceSample")
    AdviceSample adviceSample;

    @Test
    public void testAdvice () throws Exception {

        SampleVO vo = new SampleVO();
        ..
        String resultStr = annotationAdviceSample.someMethod(vo);

        assertEquals("someMethod executed.(modified)", resultStr);

    }
}
```

❑ XML 스키마를 이용한 AOP 지원(9/11)

– Aspect 실행하기 – 정상 실행인 경우

- 콘솔 로그 출력 Advice 적용 순서

1. before
2. around (대상 메소드 수행 전)
3. 대상 메소드
4. after-returning
5. after(finally)
6. around (대상 메소드 수행 후)

❑ XML 스키마를 이용한 AOP 지원(10/11)

– Aspect 실행하기 – 예외 발생의 경우

```
public class AnnotationAspectTest {

    @Resource(name = "adviceSample")
    AdviceSample adviceSample;

    @Test
    public void testAdviceWithException() throws Exception {

        SampleVO vo = new SampleVO();
        // exception 을 발생하도록 플래그 설정
        vo.setForceException(true);
        ..
        try {
            String resultStr = annotationAdviceSample.someMethod(vo);

            fail("exception을 강제로 발생시켜 이 라인이 수행될 수 없습니다.");
        } catch(Exception e) {
            ...
        }
    }
}
```

❑ XML 스키마를 이용한 AOP 지원(11/11)

– Aspect 실행하기 – 예외 발생의 경우

- 콘솔 로그 출력 Advice 적용 순서

1. before
2. around (대상 메소드 수행 전)
3. 대상 메소드 (ArithmeticException 예외가 발생한다)
4. afterThrowing
5. after(finally)

❑ Pointcut 지정자

- execution: 메소드 실행 결합점(join points)과 일치시키는데 사용된다.
- within: 특정 타입에 속하는 결합점을 정의한다.
- this: 빈 참조가 주어진 타입의 인스턴스를 갖는 결합점을 정의한다.
- target: 대상 객체가 주어진 타입을 갖는 결합점을 정의한다.
- args: 인자가 주어진 타입의 인스턴스인 결합점을 정의한다.
- @target: 수행중인 객체의 클래스가 주어진 타입의 어노테이션을 갖는 결합점을 정의한다.
- @args: 전달된 인자의 런타입 타입이 주어진 타입의 어노테이션을 갖는 결합점을 정의한다.
- @within: 주어진 어노테이션을 갖는 타입 내 결합점을 정의한다.
- @annotation: 결합점의 대상 객체가 주어진 어노테이션을 갖는 결합점을 정의한다.

❑ Pointcut 표현식 조합

- '&&' : anyPublicOperation() && inTrading()
- '||' : bean(*dataSource) || bean(*DataSource)
- '!' : !bean(accountRepository)

❑ Pointcut 정의 예제

Pointcut	선택된 Joinpoints
execution(public * *(..))	public 메소드 실행
execution(* set*(..))	이름이 set으로 시작하는 모든 메소드명 실행
execution(* com.xyz.service.AccountService.*(..))	AccountService 인터페이스의 모든 메소드 실행
execution(* com.xyz.service.*.*(..))	service 패키지의 모든 메소드 실행
execution(* com.xyz.service..*.*(..))	service 패키지와 하위 패키지의 모든 메소드 실행
within(com.xyz.service.*)	service 패키지 내의 모든 결합점
within(com.xyz.service..*)	service 패키지 및 하위 패키지의 모든 결합점
this(com.xyz.service.AccountService)	AccountService 인터페이스를 구현하는 프록시 개체의 모든 결합점
target(com.xyz.service.AccountService)	AccountService 인터페이스를 구현하는 대상 객체의 모든 결합점
args(java.io.Serializable)	하나의 파라미터를 갖고 전달된 인자가 Serializable인 모든 결합점
@target(org.springframework.transaction.annotation.Transactional)	대상 객체가 @Transactional 어노테이션을 갖는 모든 결합점
@within(org.springframework.transaction.annotation.Transactional)	대상 객체의 선언 타입이 @Transactional 어노테이션을 갖는 모든 결합점
@annotation(org.springframework.transaction.annotation.Transactional)	실행 메소드가 @Transactional 어노테이션을 갖는 모든 결합점
@args(com.xyz.security.Classified)	단일 파라미터를 받고, 전달된 인자 타입이 @Classified 어노테이션을 갖는 모든 결합점
bean(accountRepository)	“accountRepository” 빈
!bean(accountRepository)	“accountRepository” 빈을 제외한 모든 빈
bean(*)	모든 빈
bean(account*)	이름이 'account'로 시작되는 모든 빈
bean(*Repository)	이름이 “Repository”로 끝나는 모든 빈
bean(accounting/*)	이름이 “accounting/“로 시작하는 모든 빈
bean(*dataSource) || bean(*DataSource)	이름이 “dataSource” 나 “DataSource” 으로 끝나는 모든 빈

❑ 실행환경 AOP 가이드라인

– 실행환경은 **예외 처리와 트랜잭션 처리**에 AOP를 적용함

❑ 실행환경 AOP 가이드라인– 예외 처리(1/2)

– 관점(Aspect) 정의: resources/egovframework.spring/context-aspect.xml

```
<bean id="exceptionTransfer" class="egovframework.rte.fdl.cmmn.aspect.ExceptionTransfer">
...
</bean>

<aop:config>
  <aop:pointcut id="serviceMethod"
  expression="execution(* egovframework.rte.sample..impl.*Impl.*(..))" />

  <aop:aspect ref="exceptionTransfer">
      <aop:after-throwing throwing="exception"
       pointcut-ref="serviceMethod" method="transfer" />
  </aop:aspect>
</aop:config>
```

Advice 정의

Pointcut 정의

JoinPoint 정의

Aspect 정의

❑ 실행환경 AOP 가이드라인– 예외 처리(2/2)

– Advice 클래스 : egovframework.rte.fdl.cmmn.aspect.ExceptionTransfer

```
public class ExceptionTransfer {

  public void transfer(JoinPoint thisJoinPoint, Exception exception) throws Exception {
    ...
    if (exception instanceof EgovBizException) {
      log.debug("Exception case :: EgovBizException ");
      EgovBizException be = (EgovBizException) exception;
      getLog(clazz).error(be.getMessage(), be.getCause());
      processHandling(clazz, exception, pm, exceptionHandlerServices, false);
      throw be;
    } else if (exception instanceof RuntimeException) {
      log.debug("RuntimeException case :: RuntimeException ");
      RuntimeException be = (RuntimeException) exception;
      getLog(clazz).error(be.getMessage(), be.getCause());
      processHandling(clazz, exception, pm, exceptionHandlerServices, true);
      ...
      throw be;
    } else if (exception instanceof FdlException) {
      ...
      throw fe;
    } else {
      ...
      throw processException(clazz, "fail.common.msg", new String[] {}, exception, locale);
    }
  }
}
```

❑ 실행환경 AOP 가이드라인– 트랜잭션 처리

– 관점(Aspect) 정의: resources/egovframework.spring/context-transaction.xml

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

<!-- 트랜잭션 Advice를 설정한다. -->
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="*" rollback-for="Exception"/>
  </tx:attributes>
</tx:advice>

<!-- 트랜잭션 Pointcut를 설정한다.--->
<aop:config>
  <aop:pointcut id="requiredTx"
   expression="execution(* egovframework.rte.sample..impl.*Impl.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="requiredTx" />
</aop:config>
```

- **Spring 2.5/3.0 Reference Documentation**
  - http://static.springframework.org/spring/docs/2.5.x/reference/aop.html
  - http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/aop.html
- **The new() Pointcut**
  - http://blog.springsource.com/2007/09/24/the-new-bean-pointcut/

LAB 202-AOP 실습

❑ @AspectJ 어노테이션을 이용한 AOP 지원 (1/11)

– 개요

- @AspectJ는 Java 5 어노테이션을 사용하여 일반 Java 클래스로 관점(Aspect)을 정의하는 방식이다.
- @AspectJ 방식은 AspectJ 5 버전에서 소개되었으며, Spring은 2.0 버전부터 AspectJ 5 어노테이션을 지원한다.
- Spring AOP 실행환경은 AspectJ 컴파일러나 Weaver에 대한 의존성이 없이 @AspectJ 어노테이션을 지원한다.

❑ @AspectJ 어노테이션을 이용한 AOP 지원 (2/11)

– @AspectJ 설정하기

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

– Aspect 정의하기

```
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class AspectUsingAnnotation {
    ..
}
```

– Pointcut 정의하기

```
@Aspect
public class AspectUsingAnnotation {
    ...
    @Pointcut("execution(public * egovframework.rte.fdl.aop.sample.*Sample.*(..))")
    public void targetMethod() {
        // pointcut annotation 값을 참조하기 위한 dummy method
    }
    ...
}
```

❑ @AspectJ 어노테이션을 이용한 AOP 지원 (3/11)

– Advice 정의하기 – before advice

- Before advice는 @Before 어노테이션을 사용한다.

```
@Aspect
public class AspectUsingAnnotation {

    @Before("targetMethod()")
    public void beforeTargetMethod(JoinPoint thisJoinPoint) {
        Class clazz = thisJoinPoint.getTarget().getClass();
        String className = thisJoinPoint.getTarget().getClass().getSimpleName();
        String methodName = thisJoinPoint.getSignature().getName();

        // 대상 메서드에 대한 로거를 얻어 해당 로거로 현재 class, method 정보 로깅
        Log logger = LogFactory.getLog(clazz);
        logger.debug(className + "." + methodName + " executed.");

    }
    …
}
```

❑ @AspectJ 어노테이션을 이용한 AOP 지원 (4/11)

– Advice 정의하기 – After returning advice

- After returing advice 는 정상적으로 메소드가 실행될 때 수행된다.
- After returning advice 는 @AfterReturing 어노테이션을 사용한다.

```
@Aspect
public class AspectUsingAnnotation {

    @AfterReturning(pointcut = "targetMethod()", returning = "retVal")
    public void afterReturningTargetMethod(JoinPoint thisJoinPoint,
            Object retVal) {
        System.out.println("AspectUsingAnnotation.afterReturningTargetMethod executed.” +
                " return value is [“ + retVal + "]");
    }    …
}
```

❑ @AspectJ 어노테이션을 이용한 AOP 지원 (5/11)

– Advice 정의하기 – After throwingadvice

- After throwing advice 는 메소드가 수행 중 예외사항을 반환하고 종료하는 경우 수행된다.
- After throwing advice 는 @AfterThrowing 어노테이션을 사용한다.

```
@Aspect
public class AspectUsingAnnotation {

    @AfterThrowing(pointcut = "targetMethod()", throwing = "exception")
    public void afterThrowingTargetMethod(JoinPoint thisJoinPoint,
            Exception exception) throws Exception {
        System.out.println("AspectUsingAnnotation.afterThrowingTargetMethod executed.");
        System.out.println("에러가 발생했습니다.", exception);

        throw new BizException("에러가 발생했습니다.", exception);
    }
    …
}
```

❑ @AspectJ 어노테이션을 이용한 AOP 지원 (6/11)

– Advice 정의하기 – After (finally) advice

- After (finally) advice 는 메소드 수행 후 무조건 수행된다.
- After advice 는 정상 종료와 예외 발생 경우를 모두 처리해야 하는 경우에 사용된다 (예:리소스 해제 작업)
- After (finally) advice 는 @After 어노테이션을 사용한다.

```
@Aspect
public class AspectUsingAnnotation {

    @After("targetMethod()")
    public void afterTargetMethod(JoinPoint thisJoinPoint) {
        System.out.println("AspectUsingAnnotation.afterTargetMethod executed.");
    }    …
}
```

❑ @AspectJ 어노테이션을 이용한 AOP 지원 (7/11)

– Advice 정의하기 – Around advice

- Around advice 는 메소드 수행 전후에 수행된다.
- Around advice 는 @Around 어노테이션을 사용한다.

```
@Aspect
public class AspectUsingAnnotation {

 @Around("targetMethod()")
    public Object aroundTargetMethod(ProceedingJoinPoint thisJoinPoint)
            throws Throwable {
        System.out.println("AspectUsingAnnotation.aroundTargetMethod start.");
        long time1 = System.currentTimeMillis();
        Object retVal = thisJoinPoint.proceed();

        System.out.println("ProceedingJoinPoint executed. return value is [" + retVal + "]");

        retVal = retVal + "(modified)";
        System.out.println("return value modified to [" + retVal + "]");

        long time2 = System.currentTimeMillis();
        System.out.println("AspectUsingAnnotation.aroundTargetMethod end. Time("
                + (time2 - time1) + ")");
        return retVal;
    }…
}
```

❑ @AspectJ 어노테이션을 이용한 AOP 지원 (8/11)

– Aspect 실행하기 – 정상 실행인 경우

```
public class AnnotationAspectTest {

    @Resource(name = "annotationAdviceSample")
    AnnotationAdviceSample annotationAdviceSample;

    @Test
    public void testAnnotationAspect() throws Exception {

        SampleVO vo = new SampleVO();
        ..
        String resultStr = annotationAdviceSample.someMethod(vo);

        assertEquals("someMethod executed.(modified)", resultStr);
    }
}
```

❑ @AspectJ 어노테이션을 이용한 AOP 지원 (9/11)

– Aspect 실행하기 – 정상 실행인 경우

- 콘솔 로그 출력 Advice 적용 순서

1. @Before
2. @Around (대상 메소드 수행 전)
3. 대상 메소드
4. @Around (대상 메소드 수행 후)
5. @After(finally)
6. @AfterReturning

❑ @AspectJ 어노테이션을 이용한 AOP 지원 (10/11)

– Aspect 실행하기 – 예외 발생의 경우

```
public class AnnotationAspectTest {

    @Resource(name = "annotationAdviceSample")
    AnnotationAdviceSample annotationAdviceSample;

    @Test
    public void testAnnotationAspect() throws Exception {

        SampleVO vo = new SampleVO();
        // exception 을 발생하도록 플래그 설정
        vo.setForceException(true);
        ..
        try {
            String resultStr = annotationAdviceSample.someMethod(vo);

            fail("exception을 강제로 발생시켜 이 라인이 수행될 수 없습니다.");
        } catch(Exception e) {
            ...
        }
    }
}
```

❑ @AspectJ 어노테이션을 이용한 AOP 지원 (11/11)

– Aspect 실행하기 – 예외 발생의 경우

- 콘솔 로그 출력 Advice 적용 순서

1. @Before
2. @Around (대상 메소드 수행 전)
3. 대상 메소드 (ArithmeticException 예외가 발생한다)
4. @After(finally)
5. @AfterThrowing