06강 AOP 소개

예제 준비

- pom.xml
- 인터페이스 Calculator.java
- 구현클래스
  - ImpeCalculator.java
  - RecCalculator.java

예제 준비

- pom.xml

```
<dependencies>
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-context</artifactId>
          <version>4.1.0.RELEASE</version>
      </dependency>

      <dependency>
          <groupId>org.aspectj</groupId>
          <artifactId>aspectjweaver</artifactId>
          <version>1.8.2</version>
      </dependency>

  </dependencies>
```

예제 준비

- 인터페이스 Calculator.java

```
public interface Calculator {
    public long factorial(long num);
}
```

예제 준비

- 구현클래스
  - ImpeCalculator.java
  - RecCalculator.java

```java
public class ImpeCalculator implements Calculator {

    @Override
    public long factorial(long num) {// 재귀 없이 팩토리얼 구하는 법

        long result=1;
        for(long i=1;i<=num;i++) {
            result*=i;
        }

        return result;
    }
```

예제 준비

- 구현클래스
  - ImpeCalculator.java
  - RecCalculator.java

```
public class RecCalculator implements Calculator {

    @Override
    public long factorial(long num) {//재귀 호출 방식

            if(num==0) {
                return 1;
            }else {
                return num*factorial(num-1);
            }


    }

}
```

프록시와 AOP

- 앞서 준비한 예제의 두 구현 클래스의 실행 시간을 출력하고자 한다.
- 일반적인 방법은 메서드 시작과 끝에 시간을 구하고 이것을 출력하는 것이다.

예제 1

```java
public class ImpeCalculator implements Calculator {

    @Override
    public long factorial(long num) {// 재귀 없이 팩토리얼 구하는 법
        long start = System.currentTimeMillis();
        long result=1;
        for(long i=1;i<=num;i++) {
            result*=i;
        }
        long end = System.currentTimeMillis();

        System.out.printf("ImpeCalculator.factorial(%d)실행시간 : %d\n",num,(end-start));
        return result;
    }
```

```java
public class RecCalculator implements Calculator {

    @Override
    public long factorial(long num) {//재귀 호출 방식
        long start = System.currentTimeMillis();
        try {
            if(num==0) {
                return 1;
            }else {
                return num*factorial(num-1);
            }
        }finally {
            long end = System.currentTimeMillis();
            System.out.printf("RecCalculator.factorial(%d)실행시간 : %d\n",num,(end-start));
        }
    }
```

프록시와 AOP

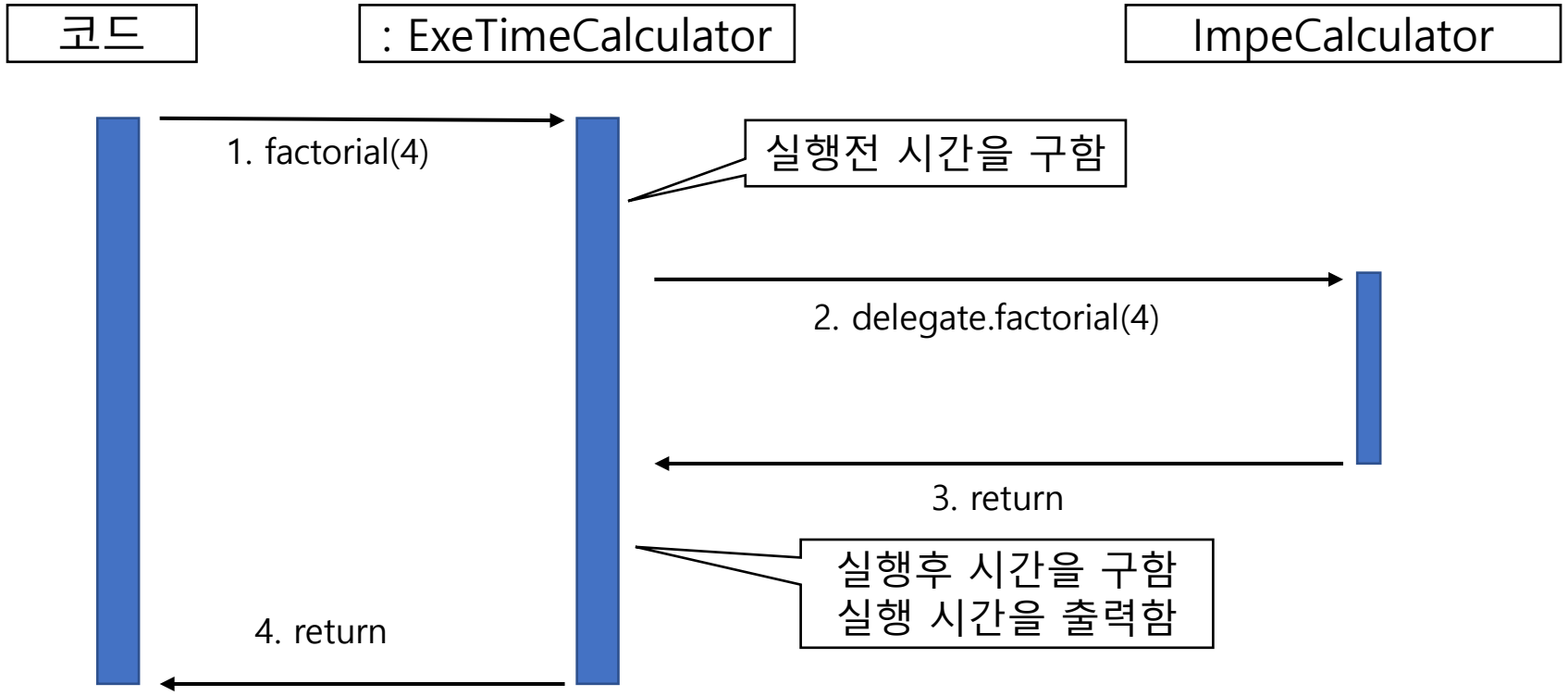
- 위 코드를 Main01.java에서 실행 해보자

```
public class Main01 {

    public static void main(String[] args) {
        ImpeCalculator impeCal = new ImpeCalculator();
        long fiveFactoral1 = impeCal.factorial(5);

        RecCalculator recCal = new RecCalculator();
        long fiveFactoral2 = recCal.factorial(5);

        System.out.println("fiveFactoral1의 결과값 : "+fiveFactoral1);
        System.out.println("fiveFactoral2의 결과값 : "+fiveFactoral2);
    }
```

프록시와 AOP

- 위 예제에서 실행시간을 밀리초가 아닌 나노초로 구하고자 한다면 어떻게 해야할까?
- 모든 코드에서 시간을 구하는 부분을 모두 바꿔주어야 할 것이다.
- 이때 기존의 코드는 수정하지 않고 코드 중복을 해결하는 방법이 프록시 객체이다.

 - 예제2



프록시와 AOP

- 프록시 객체를 만들어 본다. -> ExeTimeCalculator.java

```java
public class ExeTimeCalculator implements Calculator{
    private Calculator delegate;

    public ExeTimeCalculator(Calculator delegate) {
        this.delegate = delegate;
    }

    @Override
    public long factorial(long num) {
        long start = System.currentTimeMillis();
        long result = delegate.factorial(num);
        long end = System.currentTimeMillis();

        System.out.printf("%s.factorial(%d)의 실행 시간 : %d\n",
                          delegate.getClass().getSimpleName(),
                          num,(end-start));
        return result;
    }
}
```

프록시와 AOP

- 프록시를 사용하도록 기존의 코드를 수정한다.

```java
public class ImpeCalculator implements Calculator {

    @Override
    public long factorial(long num) {// 재귀 없이 팩토리얼 구하는 법
        //long start = System.currentTimeMillis();
        long result=1;
        for(long i=1;i<=num;i++) {
            result*=i;
        }
        //long end = System.currentTimeMillis();

        //System.out.printf("ImpeCalculator.
        return result;
    }
```

```java
public class RecCalculator implements Calculator {

    @Override
    public long factorial(long num) { //재귀 호출 방식
        //long start = System.currentTimeMillis();
        try {
            if(num==0) {
                return 1;
            }else {
                return num*factorial(num-1);
            }
        }finally {
        //  long end = System.currentTimeMillis();
        //  System.out.printf("RecCalculator.factorial(%d)실행시간 : %d\n",num,(end-start));
        }

    }
```

프록시와 AOP

- Main02.java를 통해서 실행 해보자.

```
public class Main02 {

    public static void main(String[] args) {
        ExeTimeCalculator impeCal=new ExeTimeCalculator(new ImpeCalculator());
        System.out.println("impeCal결과 : "+impeCal.factorial(10));

        ExeTimeCalculator recCal=new ExeTimeCalculator(new RecCalculator());
        System.out.println("recCal결과 : "+recCal.factorial(10));
    }
```

프록시와 AOP

- 프록시 객체를 수정해서 나노초 단위로 시간을 구해보자

```
@Override
public long factorial(long num) {
    //long start = System.currentTimeMillis();
    long start = System.nanoTime();
    long result = delegate.factorial(num);
    //long end = System.currentTimeMillis();
    long end = System.nanoTime();

    System.out.printf("%s.factorial(%d)의 실행 시간 : %d\n",
                      delegate.getClass().getSimpleName(),
                      num,(end-start));
    return result;
}
```

프록시와 AOP

- 위 결과로 알수있는 점
  - 기존 코드를 변경하지 않고 실행시간을 출력할 수 있다.
  - 실행 시간을 구하는 코드의 중복을 제거했다. (변경하고 싶다면 해당 코드만 바꿔주면 된다.

- 이와 같이 핵심 기능의 실행은 다른 객체에 위임하고 부가적인 기능을 제공하는 객체를 프록시 객체라고 한다.
- 실제 핵심 기능을 실행하는 객체를 대상 객체라고 한다.

- 프록시 객체는 핵심 기능을 구현하지 않는 대신 여러 객체에 공통으로 적용해야 할 기능을 구현한다.

프록시와 AOP

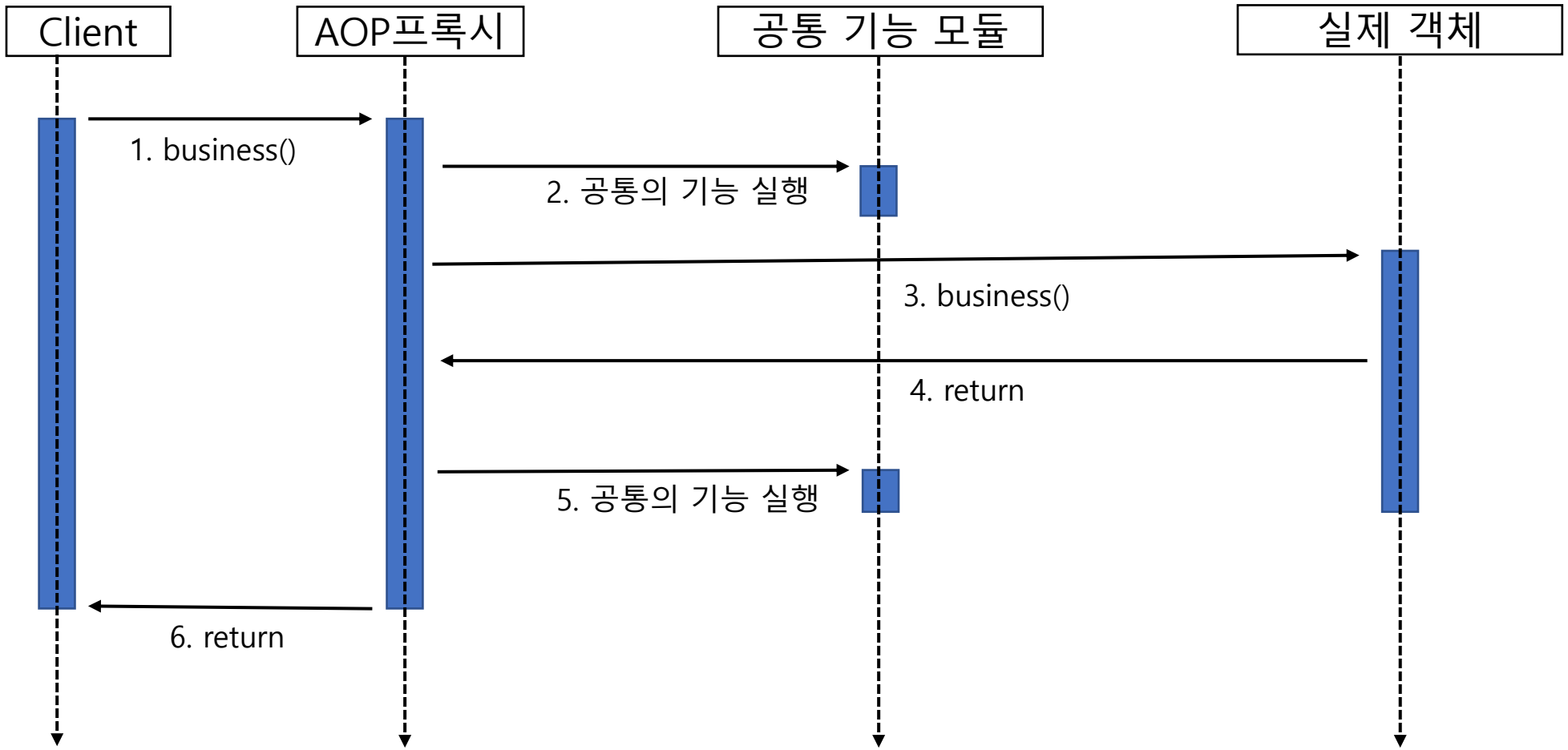
- 핵심 코드와 여러 객체에 공통으로 적용가능한 객체를 구분함으로써 재사용성을 높이는 프로그래밍 기법을 AOP(Aspect Oriented Programming)이라고 한다.

- AOP의 기본 개념은 핵심 기능에 공통의 기능을 삽입하는 것이다.
  - 컴파일 시점에 코드에 공통의 기능을 추가하는 방법
  - 클래스 로딩 시점에 바이트 코드에 공통의 기능을 추가하는 방법
  - 런타임에 프록시 객체를 생성해서 공통의 기능을 추가하는 방법

- 스프링에서는 프록시 객체를 활용한 세번째 방식을 채택한다.

프록시와 AOP

- 실제 스프링 AOP는 프록시 객체를 자동으로 생성해 준다. 그러므로 개발자는 공통의 기능을 구현하기 위한 클래스만 알맞게 구현해 주면 된다.



프록시와 AOP

- AOP의 주요 용어

용어	설명
Aspect	여러 객체에 공통으로 적용되는 기능을 Aspect라고 한다. 트랜젝션이나 보안등이 Aspect의 좋은 예시이다.
Advice	언제 공통 관심 기능을 핵심 로직에 적용할 지를 정의하고 있다. 예를 들어 메서드 호출전에(언제) 트랜젝션 시작 기능(공통기능)을 적용 같은 경우이다.
Weaving	Advice를 핵심 로직 코드에 적용하는 것을 Weaving라고 한다.
Joinpoint	Advice를 적용 가능한 지점을 의미한다. 메서드 호출, 필드 값 변경 등이 Joinpoint에 해당한다. 스프링은 프록시를 이용해서 AOP를 구현하기 때문에 메서드 호출에 대한 Joinpoint만 지원한다.
Pointcut	Joinpoint의 부분집합으로 실제로 Advice가 적용되는 Joinpoint를 나타낸다. 스프링에서는 정규 표현식이나 AspectJ의 문법을 이용하여 Pointcut을 정의할 수 있다.

프록시와 AOP

- Advice의 종류

종류	설명
Before Advice	대상 객체의 메서드 호출 전에 공통의 기능을 실행한다.
After Return Advice	대상 객체의 메서드가 익셉션이 없이 실행된 이후에 공통의 기능을 실행한다.
After Throwing Advice	대상 객체의 메서드를 실행하는 도중에 익셉션이 발생한 경우에 공통의 기능을 실행한다.
After Advice	대상 객체의 메서드를 실행하는 도중에 익셉션이 발생 했는지 여부에 상관없이 메서드 실행 후 공통의 기능을 실행한다.
Around Advice	대상 객체의 메서드 실행 전, 후 또는 익셉션 발생 시점에 공통의 기능을 실행하는데 사용된다.

- 이중 가장 많이 사용되는 것은 Around Advice 이다

스프링 AOP 구현

- 스프링에서 AOP를 이용해서 공통의 기능을 구현하는 방법은 다음과 같은 절차를 따르면 된다.

- 공통의 기능을 제공하는 Aspect를 구현한다.
- Aspect를 어디(Pointcut)에 적용할 지를 설정한다. 즉 (Advice)를 설정한다.

- Aspect를 구현하는 방법에는 두가지가 있다
  - XML 스키마 기반의 자바 POJO클래스를 이용하는 방법
  - @Aspect 애노테이션을 이용하는 방법

스프링 AOP 구현 – XML 스키마 기반

- XML 스키마를 이용하는 경우 – XML 설정을 이용해서 Aspect를 어디에 적용할 지를 설정한다.
- 우선 적용할 Aspect객체를 만든다.
  - 예제 3 (이 객체는 공통의 기능을 담고 있으며 Around Advice에 사용할 객체이다.)

- getSignature() : 메서드 시그니처
- getTarget() : 대상 객체
- getArgs() : 인자 목록

```
public class ExeTimeAspect {

    public Object measure(ProceedingJoinPoint joinPoint) throws Throwable {
        long start = System.nanoTime();
        try {
            Object result = joinPoint.proceed();//대상 객체의 적용 메소드로 연결
            return result;
        }finally {
            long end = System.nanoTime();
            Signature sig = joinPoint.getSignature();
            System.out.printf("%s.%s(%s) 실행 시간 : %d\n",
                    joinPoint.getTarget().getClass().getSimpleName(),
                    sig.getName(), Arrays.toString(joinPoint.getArgs()),
                    (end-start));
        }
    }
}
```

스프링 AOP 구현 – XML 스키마 기반

- XML 스키마를 이용하는 경우 – XML 설정을 이용해서 Aspect를 어디에 적용할 지를 설정한다.
- 위에서 구한 Aspect 객체를 적용시키기 위해 XML 스키마 설정을 한다.
  - 예제 4 (설정파일 생성시 aop를 체크한다.)

```
<!-- 공통의 기능을 제공할 클래스를 빈으로 등록 -->
    <bean id="exeTimeAspect" class="ex06.ExeTimeAspect" />

    <!-- Aspect 설정 : advice를 어떤 PointCut에 적용할 지를 설정 -->
    <aop:config>
        <aop:aspect id="measureAspect" ref="exeTimeAspect">
            <aop:pointcut
            expression="execution(public * spring.calc..*(..))"
            id="publicMethod"/>
            <aop:around method="measure" pointcut-ref="publicMethod"/>
        </aop:aspect>
    </aop:config>

    <bean id="impeCal" class="ex06.ImpeCalculator">
    </bean>

    <bean id="recCal" class="ex06.RecCalculator">
    </bean>
```

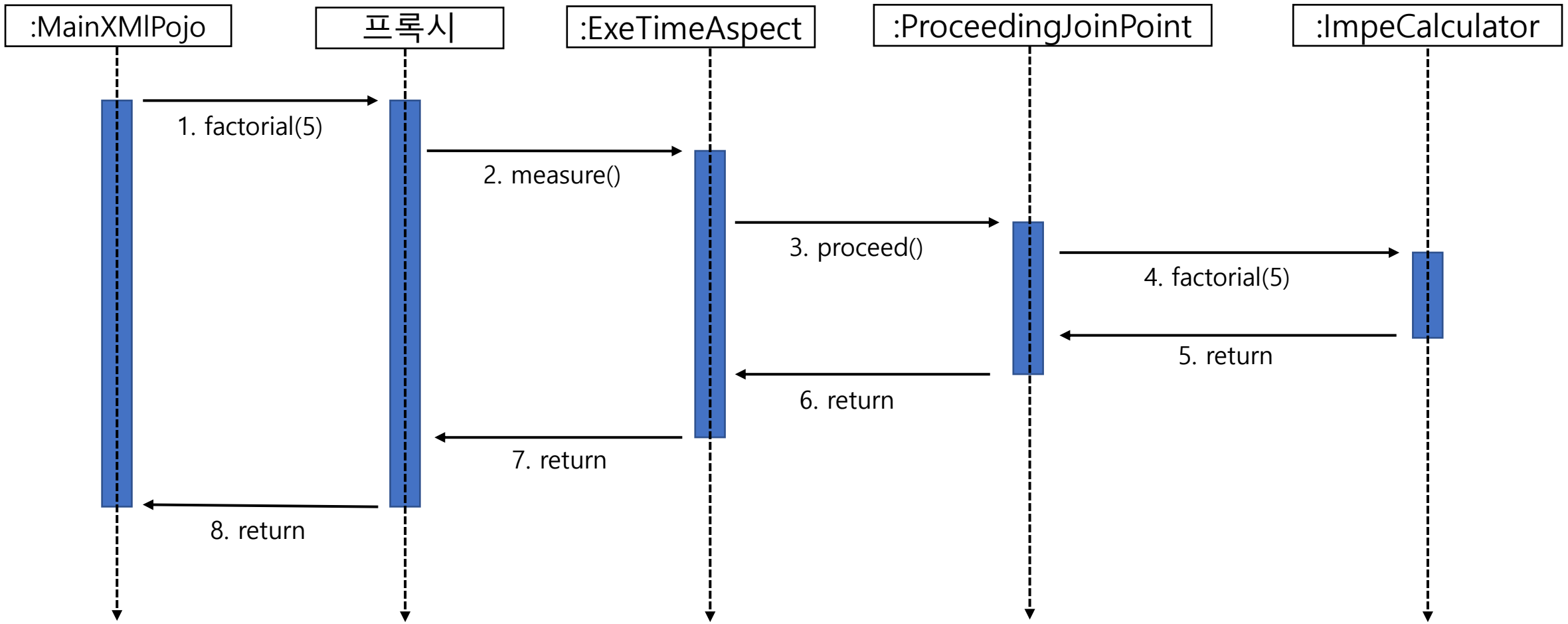
스프링 AOP 구현 – XML 스키마 기반

- XML 스키마를 이용하는 경우 – XML 설정을 이용해서 Aspect를 어디에 적용할 지를 설정한다.
- 실제 Aspect가 적용되는 지 확인한다. => MainXMLPojo.java

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

        GenericXmlApplicationContext ctx =
                new GenericXmlApplicationContext("classpath:aopPojo.xml");

        Calculator impeCal = ctx.getBean("impeCal",Calculator.class);
        long fiveFact1 = impeCal.factorial(5);
        System.out.println("impeCal.factorial(5) = "+fiveFact1);

        Calculator recCal = ctx.getBean("recCal",Calculator.class);
        long fiveFact2 = recCal.factorial(5);
        System.out.println("recCal.factorial(5) = "+fiveFact2);
    }
}
```

스프링 AOP 구현 – XML 스키마 기반

- 실제 실행 흐름



스프링 AOP 구현 – @Aspect 애노테이션 이용

- @Aspect 애노테이션을 이용하는 방법도 POJO방식과 크게 다르지 않다.
- 차이점은 @Aspect 애노테이션을 적용할 클래스에 공통 기능과 PointCut 을 설정한다는 점이다.

예제 6

```
@Aspect
public class ExeTimeAspect2 {

    @Pointcut("execution(public * spring.calc..*(..))")
    private void publicTarget() {
    }

    @Around("publicTarget()")
    public Object measure(ProceedingJoinPoint joinPoint) throws Throwable {
        long start = System.nanoTime();
        try {
            Object result = joinPoint.proceed();
            return result;
        }finally {
            long end = System.nanoTime();
            Signature sig = joinPoint.getSignature();
            System.out.printf("%s.%s(%s) 실행 시간 : %d\n",
                    joinPoint.getTarget().getClass().getSimpleName(),
                    sig.getName(), Arrays.toString(joinPoint.getArgs()),
                    (end-start));
        }
    }
}
```

스프링 AOP 구현 – @Aspect 애노테이션 이용

- @Aspect 애노테이션을 이용하는 방법도 POJO방식과 크게 다르지 않다.
- 차이점은 @Aspect 애노테이션을 적용할 클래스에 공통 기능과 PointCut 을 설정한다는 점이다.

- 앞서 본 POJO 객체와 유사하지만 다음 내용이 추가 되어있다
  - 클래스에 @Aspect 애노테이션 추가
  - @Pointcut 애노테이션을 이용해서 Pointcut 설정
  - @Around 애노테이션을 사용해서 메서드가 Around Advice로 사용됨을 설정

스프링 AOP 구현 – @Aspect 애노테이션 이용

- @Aspect 애노테이션을 적용해 공통의 기능을 추가한 경우 XML에서 이것을 인식할 수 있도록 설정해 주어야 한다.

  <aop:aspectj-autoproxy /> 태그를 사용하면 @Aspect애노테이션이 적용된 빈 객체를 Advice로 사용한다.

```
<aop:aspectj-autoproxy />

<bean id="exeTimeAspect" class="spring.aspect.ExeTimeAspect2"/>

<bean id="impeCal" class="spring.calc.ImpeCalculator">
</bean>

<bean id="recCal" class="spring.calc.RecCalculator">
    </bean>
```

스프링 AOP 구현 – @Aspect 애노테이션 이용

- @Aspect 애노테이션을 적용해 공통의 기능을 추가한 경우 XML에서 이것을 인식할 수 있도록 설정해 주어야 한다.

- 실제 공통의 기능이 사용되었는 지 확인하기 위해 다음 예제를 실행한다. – 예제 8

```
public class MainXmlAspect {

    public static void main(String[] args) {

        GenericXmlApplicationContext ctx =
                new GenericXmlApplicationContext("classpath:aopAspect.xml");

        Calculator impeCal = ctx.getBean("impeCal",Calculator.class);
        long fiveFact1 = impeCal.factorial(5);
        System.out.println("impeCal.factorial(5) = "+fiveFact1);

        Calculator recCal = ctx.getBean("recCal",Calculator.class);
        long fiveFact2 = recCal.factorial(5);
        System.out.println("recCal.factorial(5) = "+fiveFact2);

    }
```

스프링 AOP 구현 – @Aspect 애노테이션 이용

- XML 설정이 아닌 Java 설정을 사용한다면 Java 설정 클래스에 @EnableAspectJAutoProxy 애노테이션을 적용하면 XML의 <aop:aspectj-autoproxy /> 태그와 같은 효과를 볼수 있다.

예제 9

```
@Configuration
@EnableAspectJAutoProxy
public class JavaConfig {

    @Bean
    public ExeTimeAspect2 exeTimeAspect() {//빈 객체가 아닌 Aspect 객체
        return new ExeTimeAspect2();
    }

    @Bean
    public Calculator impeCal() {
        return new ImpeCalculator();
    }
    @Bean
    public Calculator recCal() {
        return new RecCalculator();
    }
```

스프링 AOP 구현 - ProceedingJoinPoint객체

- 보통의 경우 Around Advice에서 사용할 공통의 기능 메서드는 대부분 파라미터로 전달 받은 ProceedingJoinPoint의 proceed()메서드만 호출하면 된다.

```
public Object measure(ProceedingJoinPoint joinPoint) throws Throwable {
        long start = System.nanoTime();
        try {
                Object result = joinPoint.proceed();
                return result;
……
```

스프링 AOP 구현 - ProceedingJoinPoin객체

- 이 경우에도 호출되는 대상 객체의 정보, 실행되는 메서드에 대한 정보, 메서드를 호출할 때 전달되는 매개값에 대한 정보가 필요할 때가 있다.
- 이때 ProceedingJoinPoin객체에는 다음과 같은 메서드를 제공한다.
  - Signature getSignature() - 호출되는 메서드에 대한 정보를 구한다.
  - Object getTarget() - 대상 객체를 구한다.
  - Object[] getArgs() – 파라미터 목록을 구한다.

- 또한 Signature 인터페이스는 다음과 같은 메서드를 제공한다.
  - String getName() - 메서드의 이름을 구한다.
  - String toLongString() - 메서드를 완전하게 표현한 문장을 구한다(리턴타입, 파라미터 타입이 모두 표시)
  - String toShortString() – 메서드를 축약해서 표현한 문자을 구한다.

스프링 AOP 구현 – 프록시 생성 방식

- 기존의 MainXMLPOJO 객체에는 다음과 같은 내용이 있다.

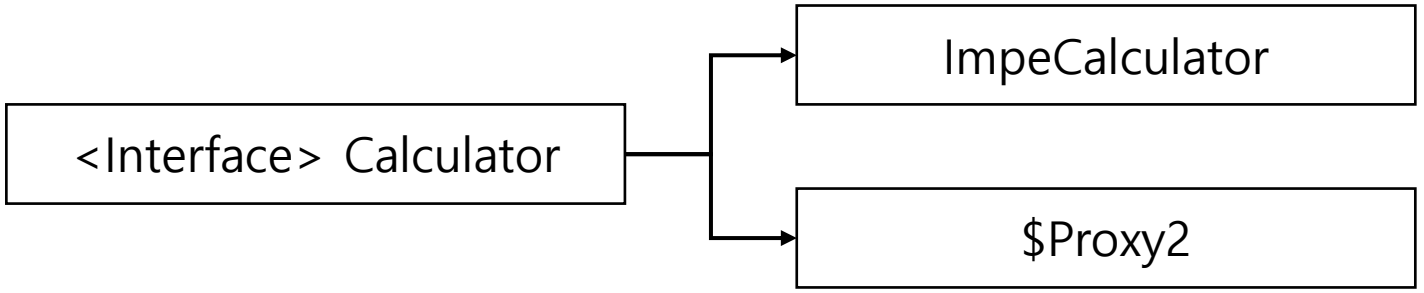
```
Calculator impeCal = ctx.getBean("impeCal", Calculator.class);
```

- 위 내용을 다음과 수정을 해도 사용이 가능할 것인가?

```
ImpeCalculator impeCal = ctx.getBean("impeCal", ImpeCalculator.class);
```

- 일반적인 자바 클래스는 사용이 가능해야 하지만 실행해 보면 에러가 발생한다.

- 그 이유는 getBean으로 생성되는 객체가 실제 객체가 아닌 프록시 객체인데 이 프록시 객체는 ImpeCalculator클래스가 구현받은 Calculator인터페이스를 구현받아 생성되기 때문이다.



스프링 AOP 구현 – 프록시 생성 방식

- 빈 객체가 인터페이스를 구현 받고 있을 때 인터페이스가 아닌 클래스를 이용해서 프록시를 생성하고 싶다면 다음과 같은 설정을 해주면 된다.

```
---POJO 방식의 XML설정
<aop:config proxy-target-class="true">
.....
</aop:config >
---@Aspect 방식의 XML 설정
<aop:aspectj-autoproxy proxy-target-class="true"/>

--- @Aspect 방식의 자바 설정
@Configuration
@EnableAspectJAutoProxy(proxyTargetClass=true)
public class JavaConfig { .....
```

스프링 AOP 구현 – execution 명시자 표현식

- Aspect 를 적용할 위치를 지정할 때 사용한 Pointcut 설정을 보면 다음과 같이 execution 명시자를 사용했다.

```
@Pointcut("execution(public * chap07..*(..))")
private void publicTarget() {}
```

- execution 명시자는 Advice를 적용할 메서드를 지정할 때 사용되며 형식은 다음과 같다.

```
execution(수식어패턴? 리턴타입패턴 클래스이름패턴? 메서드이름패턴(파라미터패턴))
```

- '수식어패턴'은 생략이 가능한 부분으로 public,protected등이 온다. 스프링 AOP의 경우 public만 적용이 가능하므로 무의미하다.
- '리턴타입패턴'은 리턴타입을 명시한다.
- '클래스이름패턴'과 '메서드이름패턴'은 클래스 이름 및 메서드 이름을 패턴으로 명시한다.
- '파라미터 패턴'부분은 매칭될 파라미터에 대해서 명시한다.

- *로 표기하는 경우 모든 값을 표기할 수 있다
- ..을 이용해서 0개 이상이라는 의미도 표기할 수 있다.

스프링 AOP 구현 – execution 명시자 표현식

- 명시자 표현식 => 메서드를 찾는 방법
  메서드의 정보가 필요
  1. 대상 객체
  2. public void method(int a, intb){....}
  2-1. public
  2-2. void, int, long, String, double[], List<m>..... *
  2-3. 메서드의 이름
  2-4. 매개변수의 정보 => 타입, 개수, 순서

스프링 AOP 구현 – execution 명시자 표현식

- publuc * spring..*(..)

=>AOP 프록시에서 잡아내야할 메서드...
- public => public이 아닌 메서드는 통과
- * 반환타입 => 어떤 반환 타입을 가지든 프록시에 잡아냄
- spring.. => spring 패키지의 모든 클래스를 프록시에 잡아냄
- * 메서드 이름 => 어떤 이름의 메서드이든 프록시에 잡아냄
- (..) 매개변수 => 매개변수의 개수가 0개 이상이면 프록시에 잡아냄

스프링 AOP 구현 – execution 명시자 표현식

- execution명시자의 몇가지 예

execution(public void set*(..))

->반환타입이 void이고 메서드 이름은 set으로 시작하며 파라미터가 0개 이상인 메서드 호출

execution(* chap07.*.*())

-> chap07 패키지의 타입에 속한 파라미터가 없는 모든 메서드 호출

execution (* chap07..*.*(..))

-> chap07 패키지및 하위 패키지에 있는 파라미터가 0개 이상인 메서드 호출

execution(Long chap07.Calculator.factorial(..))

->반환 타입이 Long인 Calculator타입의 factorial메서드를 호출

스프링 AOP 구현 – execution 명시자 표현식

- execution명시자의 몇가지 예

execution(* get*(*))

->이름이 get으로 시작하고 1개의 파라미터를 가지는 메서드 호출

execution (* get*(*,*))

-> 이름이 get으로 시작하고 2개의 파라미터를 가지는 메서드 호출

execution (* read*(Integer,..))

-> 메서드 이름이 read로 시작하고, 첫번째 파라미터 타입이 Integer이며 , 1개 이상의 파라미터를 갖는 메서드 호출

스프링 AOP 구현 – Advice 적용 순서

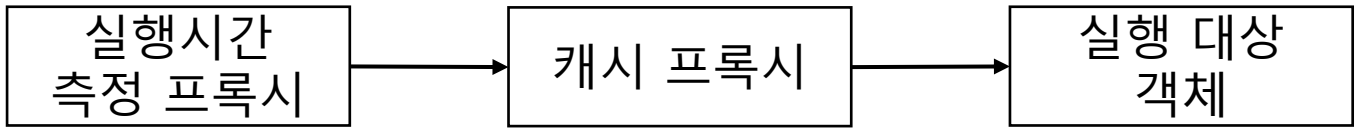
- 다음 CacheAspect 클래스를 만들자 - 예제10

```
public class CacheAspect {
    private Map<Long,Object> cache = new HashMap<>();//임시로 저장할 데이터

    public Object execute(ProceedingJoinPoint joinPoint) throws Throwable {
        Long num = (Long)joinPoint.getArgs()[0];

        if(cache.containsKey(num)) {
            System.out.printf("CacheAspect: Cache에서 구함 [%d]\n",num);
            return cache.get(num);
        }
        Object result = joinPoint.proceed();
        cache.put(num, result);
        System.out.printf("CacheAspect: Cache에 추가 [%d]\n",num);
        return result;
    }
```

스프링 AOP 구현 – Advice 적용 순서

- CacheAspect는 일종의 캐시를 구현한 Aspect로 실행 결과를 Map에 보관했다가 다음 동일한 요청이 들어오면 Map에 저장된 결과를 반환한다.
- 동일한 작업을 수행하지 않으므로 결과적으로 시간을 줄여준다.

- 캐시의 효과를 보려면 다음과 같은 순서로 실행되어야 한다.



- 이렇듯 Aspect의 적용 순서가 중요한 경우 순서를 직접 정해 주어야 한다.

스프링 AOP 구현 – Advice 적용 순서

- XML 스키마 기반 AOP설정을 사용하는 경우 <aop:aspect>태그에 order속성을 이용한다.

```
<aop:aspect id="calculatorCache" ref="cacheAspect" order="1">
</aop:aspect>
<aop:aspect id="measureAspect" ref="exeTimeAspect" order="0">
</aop:aspect>
```

```
<!-- 공통의 기능을 제공할 클래스를 빈으로 등록 -->
    <bean id="exeTimeAspect" class="spring.aspect.ExeTimeAspect"/>
    <bean id="cacheAspect" class="spring.aspect.CacheAspect"/>

    <!-- Aspect 설정 -->
    <aop:config>
        <aop:aspect id="measureAspect" ref="exeTimeAspect" order="0">
            <aop:pointcut id="publicMethod"
                expression="execution(public * spring.calc..*(..))" />
            <aop:around method="measure" pointcut-ref="publicMethod"/>
        </aop:aspect>

        <aop:aspect id="calcCache" ref="cacheAspect"  order="1">
            <aop:pointcut id="caleCacheMehod"
                expression="execution(public * spring.calc.Calculator.*(..))" />
            <aop:around method="execute" pointcut-ref="caleCacheMehod"/>
        </aop:aspect>
    </aop:config>

    <!-- 스프링 빈 객체 -->
    <bean id="impeCalc" class="spring.calc.ImpeCalculator">
    </bean>
```

스프링 AOP 구현 – Advice 적용 순서

- XML 스키마 기반 AOP설정을 사용하는 경우 <aop:aspect>태그에 order속성을 이용한다.
- MainXmlOrder클래스로 확인해 본다

```
public class MainXmlOrder {

    public static void main(String[] args) {
        GenericXmlApplicationContext ctx =
                new GenericXmlApplicationContext("classpath:aopOrder.xml");

        Calculator impeCalc = ctx.getBean("impeCalc",Calculator.class);
        impeCalc.factorial(5);
        impeCalc.factorial(5);


    }
}
```

스프링 AOP 구현 – Advice 적용 순서

- @Aspect 애노테이션을 사용하는 경우

```
@Aspect
@Order(1)
public class ExeTimeAspect3 {....}
```

```java
@Aspect
@Order(2)  //숫자가 작을 수록 우선순위가 높아진다.
public class CacheAspect2 {

    private Map<Long,Object> cache = new HashMap<>();//임시로 저장할 데이터

    @Pointcut("execution(public * spring.calc..*(..))")
    private void publicCacheTarget() {

    }

    @Around("publicCacheTarget()")
    public Object execute(ProceedingJoinPoint joinPoint) throws Throwable {
        Long num = (Long)joinPoint.getArgs()[0];

        if(cache.containsKey(num)) {
            System.out.printf("CacheAspect: Cache에서 구함 [%d]\n",num);
            return cache.get(num);
        }
        Object result = joinPoint.proceed();
        cache.put(num, result);
        System.out.printf("CacheAspect: Cache에 추가 [%d]\n",num);
        return result;
    }
}
```

```java
@Aspect
@Order(1)  //숫자가 작을 수록 우선순위가 높아진다.

public class ExeTimeAspect3 {
    @Pointcut("execution(public * spring.calc..*(..))")
    private void publicTarget() {
    }

    @Around("publicTarget()")
    public Object measure(ProceedingJoinPoint joinPoint) throws Throwable {
        long start = System.nanoTime();
        try {
            Object result = joinPoint.proceed();
            return result;
        }finally {
            long end = System.nanoTime();
            Signature sig = joinPoint.getSignature();
            System.out.printf("%s.%s(%s) 실행 시간 : %d\n",
                    joinPoint.getTarget().getClass().getSimpleName(),
                    sig.getName(), Arrays.toString(joinPoint.getArgs()),
                    (end-start));
        }
    }
```

스프링 AOP 구현 – Advice 적용 순서

- aopOrder2.xml 과 MainXmlOrder2클래스를 각각 만들어서 테스트 해본다

```
<aop:aspectj-autoproxy/>

<bean id="cacheAspect" class="spring.aspect.CacheAspect2" />
<bean id="exeTimeAspect" class="spring.aspect.ExeTimeAspect3" />
<bean id="impeCalc" class="spring.calc.ImpeCalculator" />
```

```
public class MainXmlOrder2 {

    public static void main(String[] args) {
        GenericXmlApplicationContext ctx =
                new GenericXmlApplicationContext("classpath:aopOrder2.xml");

        Calculator impeCalc = ctx.getBean("impeCalc",Calculator.class);
        impeCalc.factorial(5);
        impeCalc.factorial(5);
    }
```