

6. Spring FrameWork

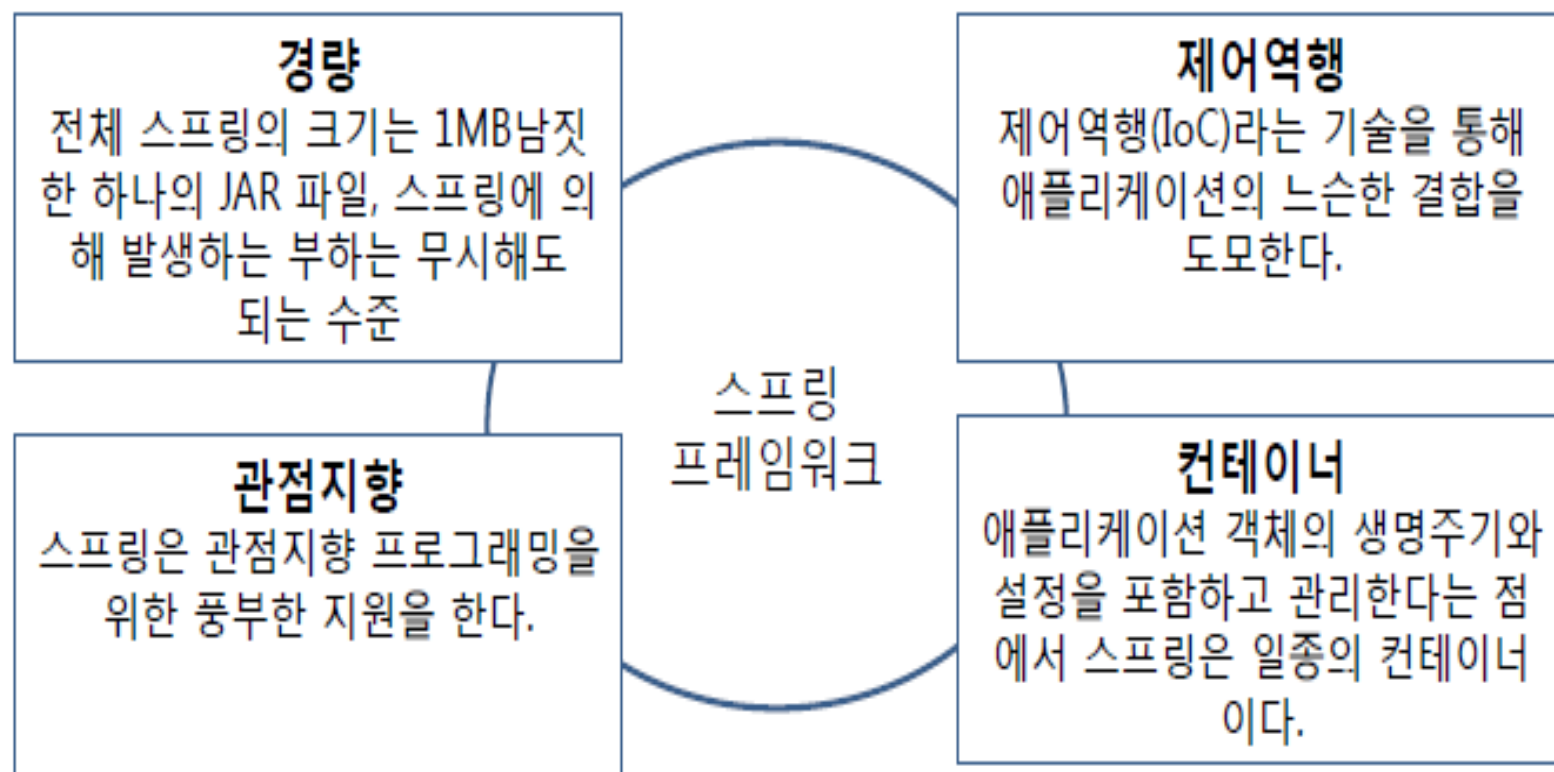
스프링 프레임워크란?

- Rod Johnson이 만든 오픈소스 프레임워크
 - 복잡한 엔터프라이즈 애플리케이션개발을 겨냥

Ejb기반으로 개발을 하지 않고 **POJO기반으로 개발을 하더라도 가볍고 제어가 가능한 상호 관련이 적은 AOP를 지원하고 컨테이너를 통해 라이프사이클을 관리하고 xml기반으로 컴포넌트를 개발** 할 수 있도록 지원하는 프레임워크 이다.

스프링 프레임워크란?

스프링은 경량의 제어 역행과 관점지향 컨테이너 프레임워크이다.



Spring 특징

- 경량의 프레임워크(가볍다) – 무거운 EJB의 대체 기술
- 설정 파일을 통해 의존관계를 주입하는 DI지원(느슨한 결합도)
- 공통관심사항을 분리하는 AOP지원(트랜잭션, 로깅, 보안)
- 특정 interface나 class를 상속받지 않아도되는 POJO지원
- 영속성(데이터를 지속적으로 유지)과 관련된 다양한 API지원 (JDBC, Ibatis, Hibernate 등등)

Spring 준비하기

1. 설치 항목

- a. Java JDK 1.5 이상
- b. Eclipse IDE J2EE
- c. Spring Framework 3.1.0 M2
- d. Spring Framework IDE (Eclipse Plug-in)
- e. Spring AOP Alliance 설치

2. Java설치

3. Eclipse IDE 설치

- a. 다운로드 주소
 - i. <http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/indigor>
- b. 다운받은후 압축을 푼다.
- c. Eclipse.exe 파일을 실행

Spring 준비하기

4. Spring Framework 설치

a. 다운로드 주소

i. <http://www.springsource.com/download>

b. 압축을 푼 후 dist 폴더로 이동.

c. 해당 폴더에 컴포넌트 별 jar파일이 위치함

d. 사용할 jar파일을 프로젝트의 lib폴더 또는 WEB-INF\lib 폴더로 복사

Spring 준비하기

<http://www.springframework.org/download>

- Spring Framework

- Latest GA release: 3.1.0.RELEASE
- More >>

- 3.1.0.RC2

[spring-framework-3.1.0.RC2-with-docs.zip \(sha1\) 51.4 MB](#)

[spring-framework-3.1.0.RC2.zip \(sha1\) 27.2 MB](#)

Spring 준비하기

다운로드 받은 파일 압축 풀 폴더 구성



각 모듈별 jar파일 포함



API 문서



모듈별 소스코드 및 빌드관련 파일

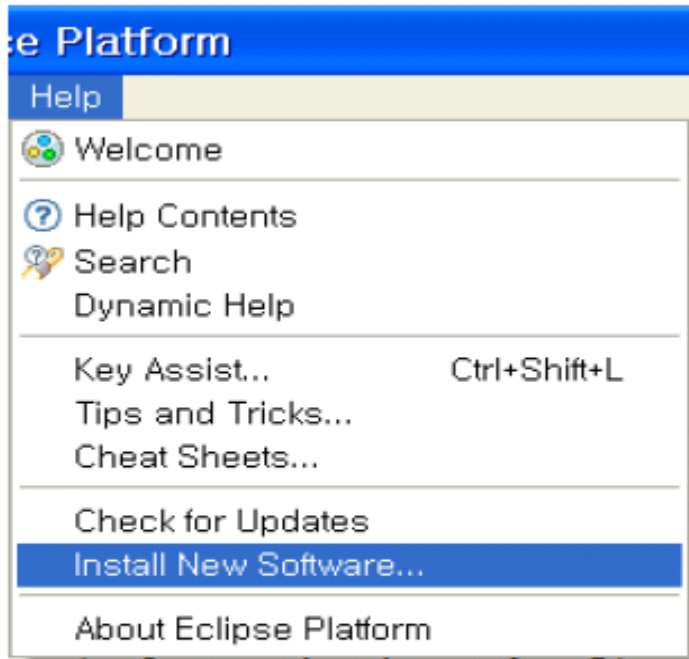


모듈별 소스jar

Spring 준비하기

5. Spring Framework IDE (Eclipse Plug-in)

a. 이클립스 를 통한 설치



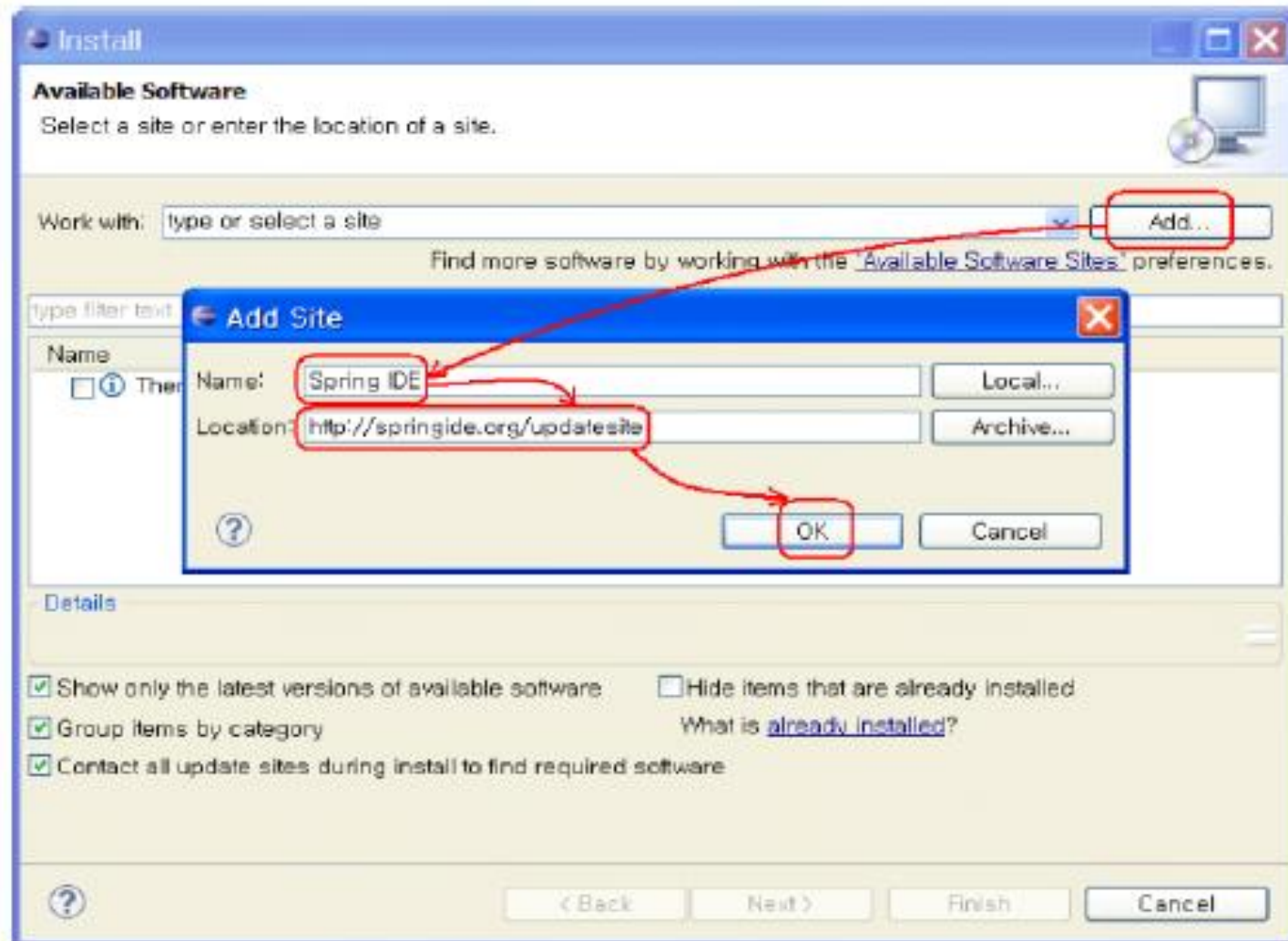
아래에 입력한 업데이트 사이트는 최근에 변경되었으므로 다음과 같은 사이트 주소를 사용해야 한다.

원래의 업데이트 사이트 주소: <http://springide.org/updatesite>

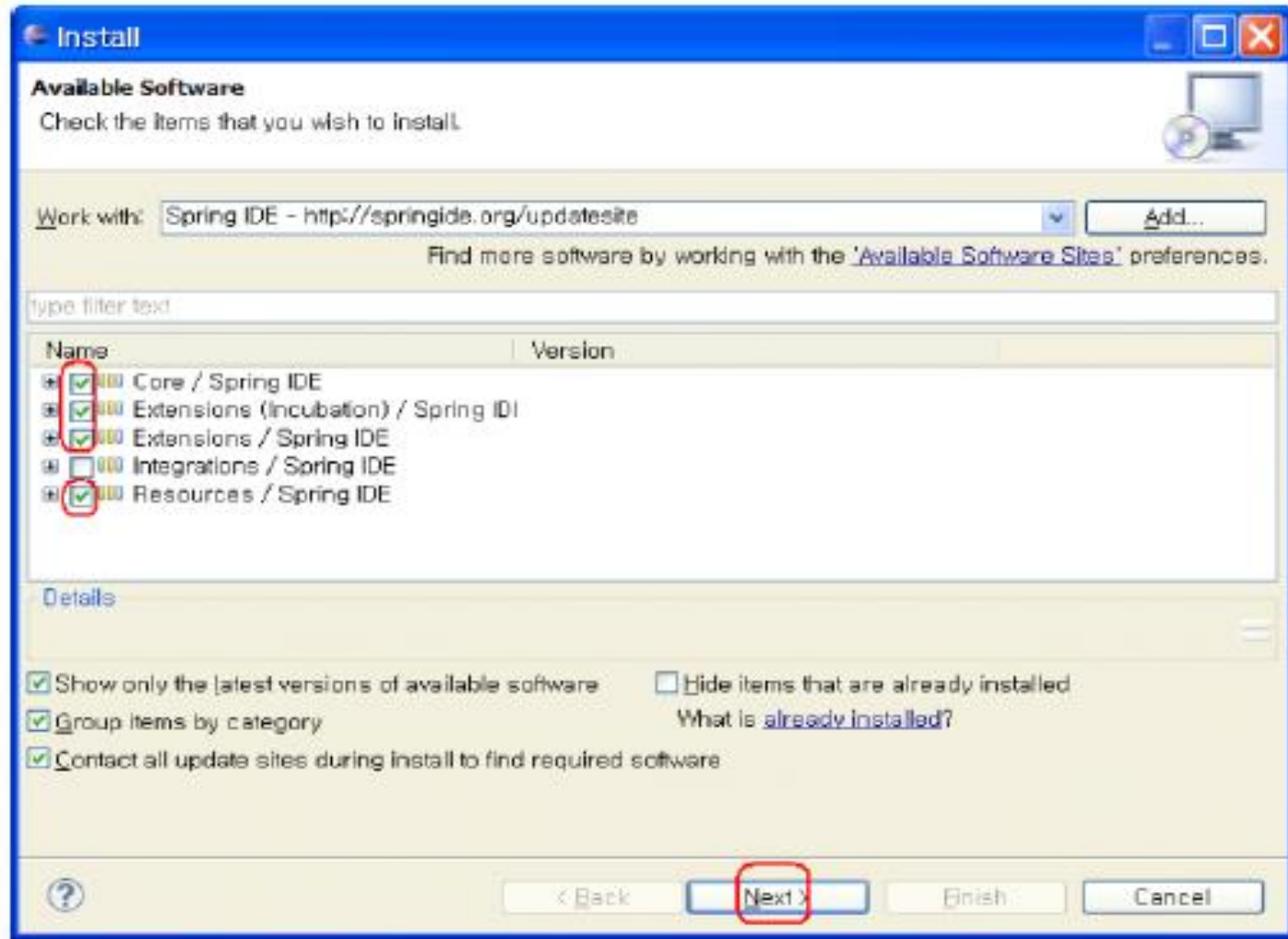
변경된 업데이트 사이트 주소: <http://dist.springframework.org/release/IDE>

원래의 업데이트 사이트 주소: <http://springide.org/updatesite>
변경된 업데이트 사이트 주소:
<http://dist.springframework.org/release/IDE>

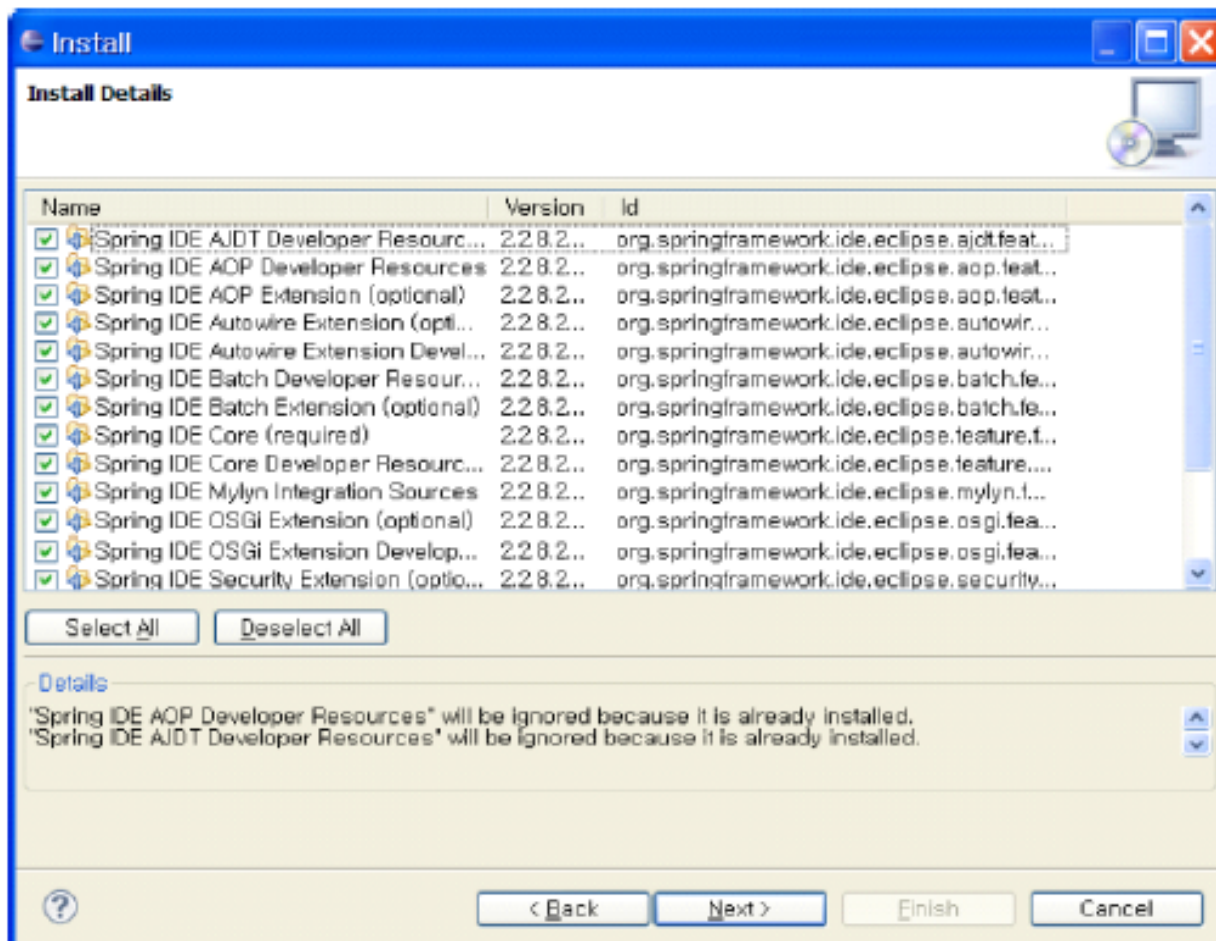
Spring 준비하기



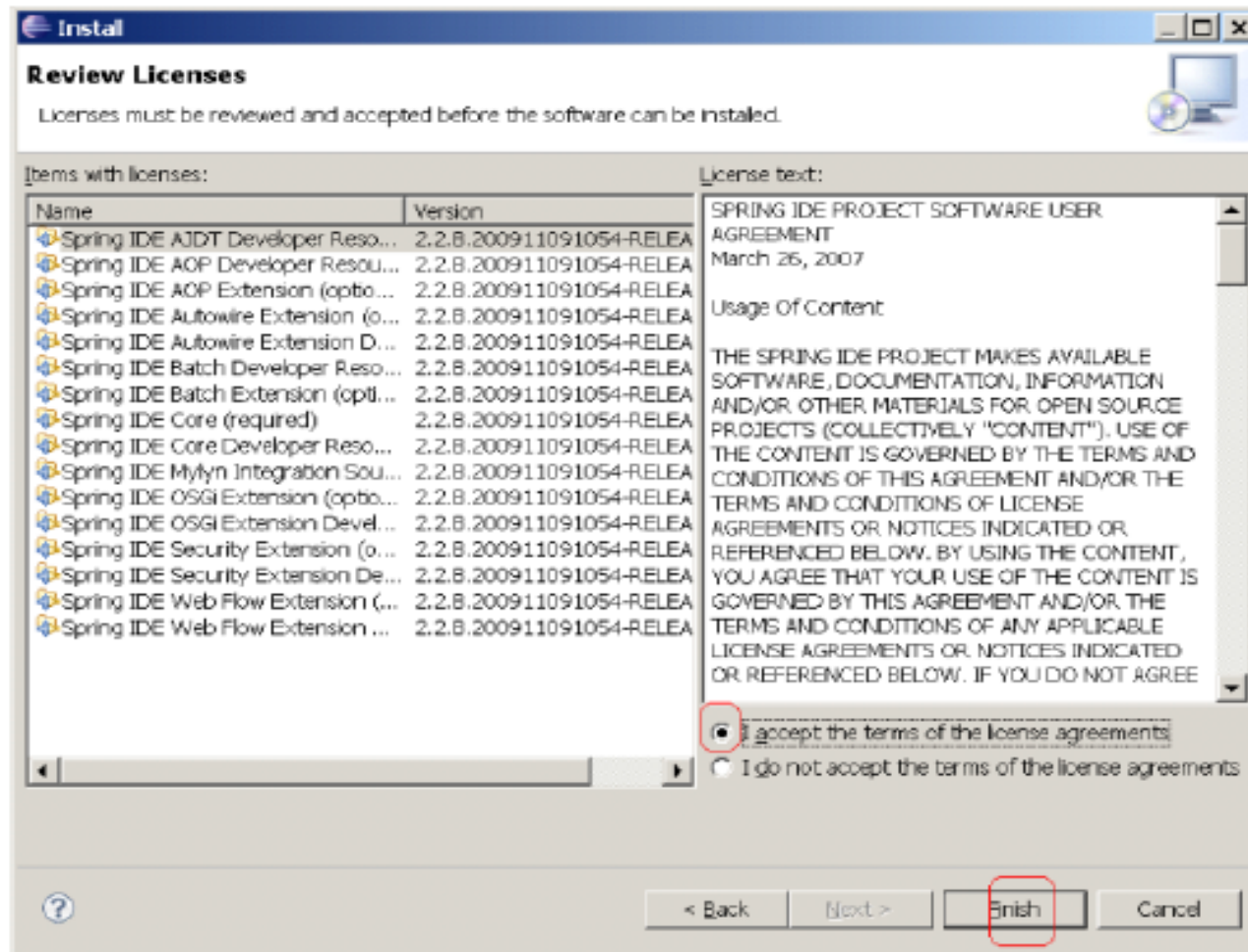
Spring 준비하기



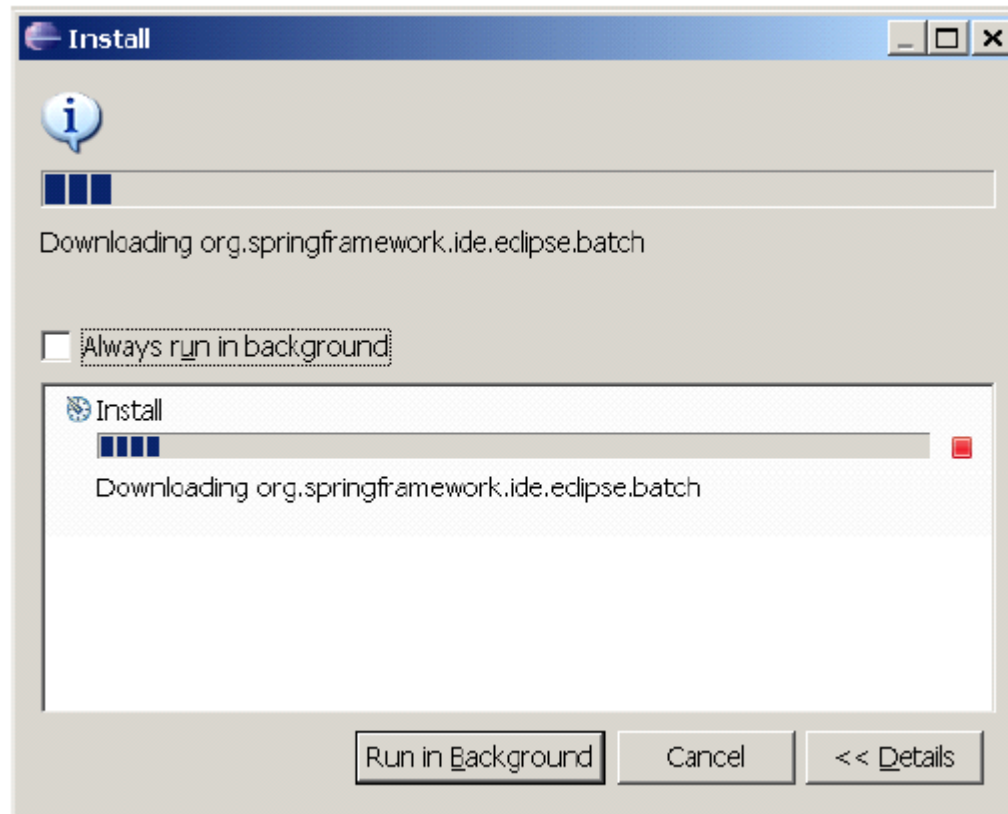
Spring 준비하기



Spring 준비하기



Spring 준비하기



Spring 준비하기

6. Spring AOP Alliance 설치

a. Jar파일 다운로드 (<http://aopalliance.sourceforge.net/>)

i. <http://sourceforge.net/projects/aopalliance/files/aopalliance/1.0/aopalliance.zip/download>

b. Jar 파일을 프로젝트의 lib폴더 또는 WEB-INF\lib 폴더로 복사

<http://sourceforge.net/projects/aopalliance/files/aopalliance/1.0/aopalliance.zip/download>

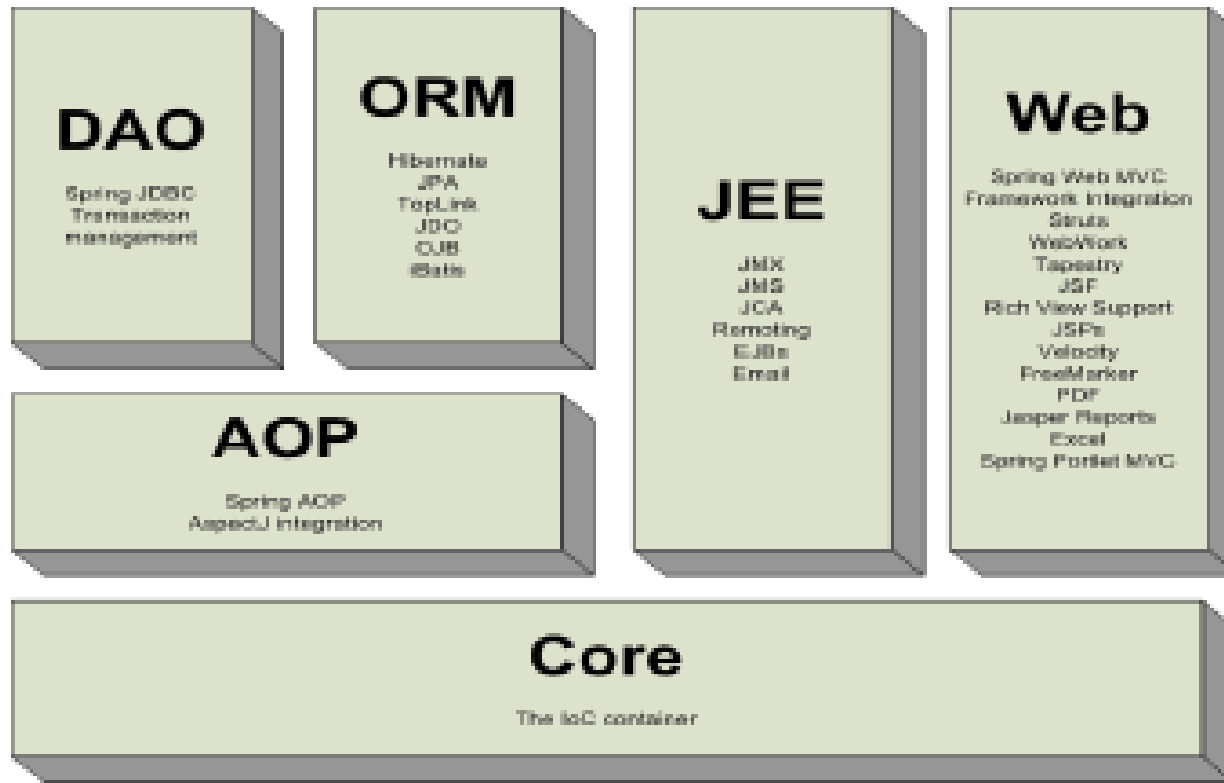
Spring 필수 라이브러리

`'spring-framework-3,1,0,RELEASE'##projects##spring-build##lib##ivy`



commons-logging
ALZip JAR File
60KB

Spring 모듈



Spring 모듈

- Spring Core

스프링의 근간이 되는 IoC(또는 DI)기능을 지원, BeanFactory를 기반으로 Bean 클래스들을 제어할 수 있는 기능을 지원한다.

- Spring AOP

스프링에 Aspect Oriented Programming을 지원하는 기능이다. 이 기능은 AOP Alliance 기반 하에서 개발되었다.

- Spring ORM

ORM(Object/ Relational Mapping)기능을 제공하는 모듈이다.

ORM 프레임워크(Hibernate, iBatis)와 JDO(Java Data Object)를 지원한다.

Spring 모듈

-Spring DAO

DAO(Data Access Object) 기능을 제공하는 모듈이다. JDBC에 의한 데이터베이스 액세스를 지원하고 트랜잭션 관리의 기반이 된다.

-Spring Web

웹 어플리케이션 개발에 필요한 Web Application Context 와 Multipart Request 등의 기능을 지원한다. 또한 스트럿츠, 웹워크와 같은 프레임워크의 통합을 지원하는 부분을 담당한다.

Spring 모듈

- Spring Context

Spring Core 바로 위에 있으면서 Spring Core에서 지원하는 기능 외에 추가적인 기능들과 좀 더 쉬운 개발이 가능하도록 지원하고 있다. 또한 JNDI(Java Naming and Directory Interface) 및 EJB의 지원, 메일 송.수신 기능 등을 제공한다.

- Spring Web MVC

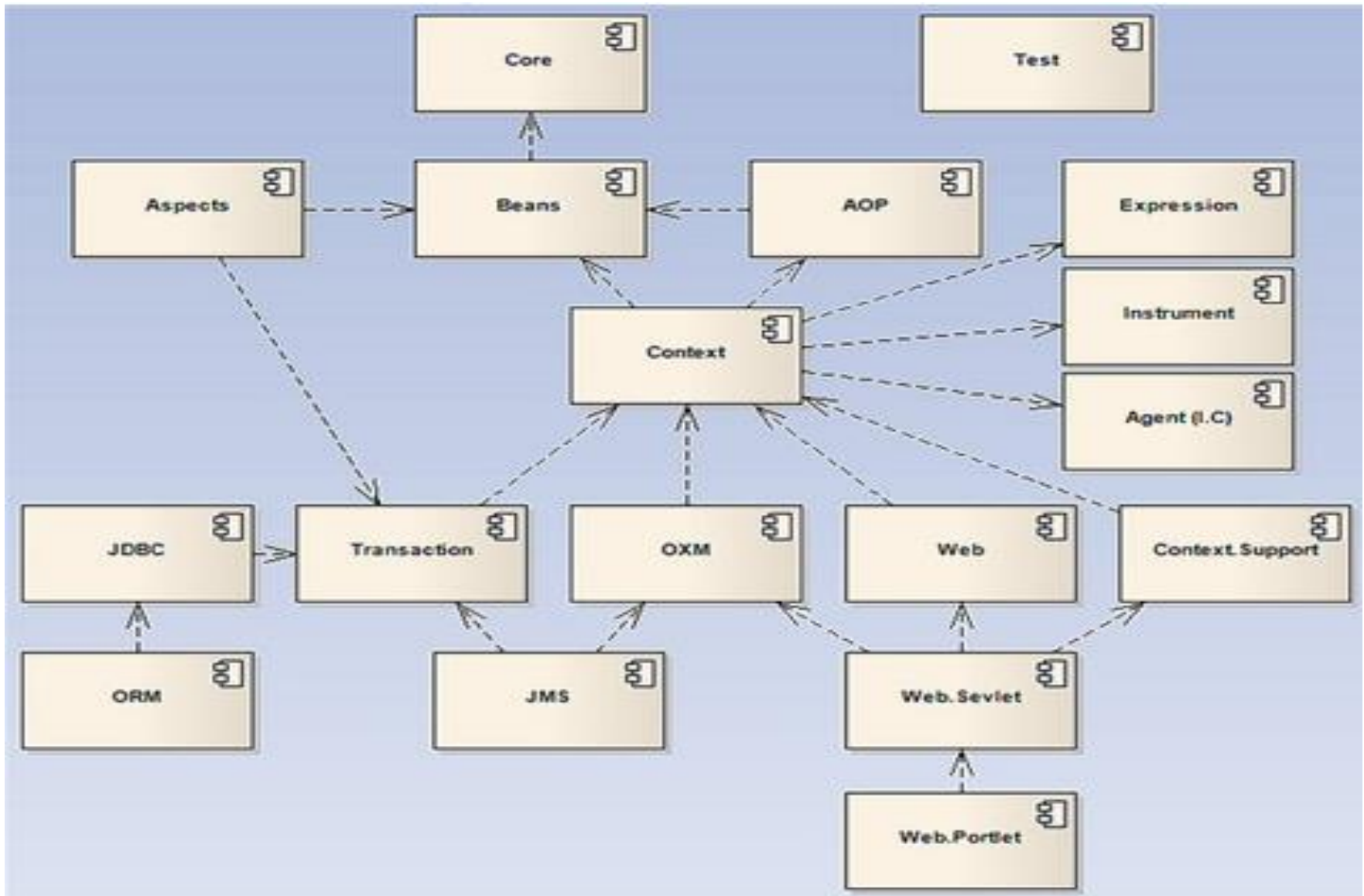
웹 어플리케이션의 MVC프레임워크 기능을 제공한다. JSP나 템플릿 엔진으로 유명한 벨로시티 지원 외에도 웹에서 PDF, 엑셀을 취급하기 위한 기능도 제공한다.

Spring 모듈

 org.springframework.aop-3.1.0.RELEASE
 org.springframework.asm-3.1.0.RELEASE
 org.springframework.aspects-3.1.0.RELEASE
 org.springframework.beans-3.1.0.RELEASE
 org.springframework.context-3.1.0.RELEASE
 org.springframework.context.support-3.1.0.RELEASE
 org.springframework.core-3.1.0.RELEASE
 org.springframework.expression-3.1.0.RELEASE
 org.springframework.instrument-3.1.0.RELEASE
 org.springframework.instrument.tomcat-3.1.0.RELEASE...
 org.springframework.jdbc-3.1.0.RELEASE
 org.springframework.jms-3.1.0.RELEASE
 org.springframework.orm-3.1.0.RELEASE
 org.springframework.oxm-3.1.0.RELEASE
 org.springframework.test-3.1.0.RELEASE
 org.springframework.transaction-3.1.0.RELEASE
 org.springframework.web-3.1.0.RELEASE
 org.springframework.web.portlet-3.1.0.RELEASE
 org.springframework.web.servlet-3.1.0.RELEASE
 org.springframework.web.struts-3.1.0.RELEASE

Spring 모듈의 의존관계

교재 33page참조



Spring 모듈의 의존관계

ASM : 엄밀히 말해 스프링 모듈이 아니다. ASM 모듈은 클래스 바이트 코드 조작 및 분석 프레임워크인 ASM을 재패키징한 모듈이다.

Core : 대부분의 스프링의 모듈에서 필요로 하는 공통 기능을 갖고 있는 핵심 모듈이다. 스프링이 사용하는 주요 타입, 애노테이션, 컨버터, 상수, 유틸리티 클래스 등을 제공한다. 모든 스프링 프로젝트에 반드시 포함시켜야 하는 필수 모듈이다.

Beans : 스프링 DI 기능의 핵심인 빈 팩토리와 DI 기능을 제공하는 모듈. 빈 메타정보, 빈 리더, 빈 팩토리의 구현과 프로퍼티 에디터가 포함되어 있다. Beans는 ASM, Core 두개의 필수 의존모듈을 갖는다.

Spring 모듈의 의존관계

AOP : 스프링의 프록시 AOP 기능을 제공하는 모듈이다. 프록시 기반 AOP를 만들때 필요한 어드바이스, 포인트컷, 프록시 팩토리빈, 자동 프록시 생성기 등을 제공한다. 스프링의 AOP는 DI에 기반을 두고 있다. 따라서 AOP 모듈은 DI 기능을 제공하는 Beans 모듈에 의존한다.

Expression : 스프링 표현식 언어 (spEL)을 지원, Core 모듈에 의존

Context : 애플리케이션 컨텍스트 기능을 제공하는 모듈. 애플리케이션 컨텍스트를 만드는데 필요한 대부분의 기능을 포함해서 빈 스캐너, 자바 코드에 대한 설정 기능, EJB 지원 기능, 포매터, 로드타임 위빙, 표현식, JMX, JNDI, 리모팅, 스케줄링, 스크립트 언어지원, 검증기 등의 애플리케이션 컨테이너로서의 주요한 기능을 담고 있다. 단순한 DI 프레임워크가 아니라 본격적인 엔터프라이즈 애플리케이션 프레임워크로 사용하기 위해 반드시 필요한 모듈이다.

Spring 모듈의 의존관계

Context.Support : Context 처럼 자주 사용되는 기능은 아니지만 경우에 따라 애플리케이션 컨텍스트에서 필요로 하는 부가기능을 담은 모듈

Transaction 모듈 : 스프링의 데이터 액세스 추상화의 공통 기능을 담고 있는 모듈. DataAccessException 예외 계층구조와 트랜잭션 추상화 기능, 트랜잭션 동기화 저장소 그리고 JCA(Java EE Connector Architecture-웹 애플리케이션 서버와 레가시 시스템과 연동할 수 있도록 하는 자바 기반 기술) 지원 기능을 포함하고 있다.
스프링의 데이터 액세스 기술을 사용하는데 반드시 필요한 모듈. 필수 의존모듈-Context

Spring 모듈의 의존관계

Context : 애플리케이션 컨텍스트 기능을 제공하는 모듈.
JDBC : JDBC 템플릿을 포함한 jdbc 지원 기능을 가진 모듈이다. 스프링이 직접 제공하는 DataSource 구현 클래스들이 제공된다. Transaction 모듈의 의존

ORM : 하이버네이트, Ibatis, JPA, JDO와 같은 ORM에 대한 스프링의 지원 기능을 갖고 있는 모듈.
JDBC에 의존, 일부 기능은 Web 모듈에 선택적 의존

Web 모듈 : 스프링 웹 기술의 공통적인 기능을 정의한 모듈. 스프링 MVC 외에도 스프링이 직접 지원하는 스트럿츠, JSF 등을 적용할 때 도 필요. Context 필수모듈, Xml 사용하는 메시징 컨버터 기능에는 OXM 모듈 필요.

Spring 모듈의 의존관계

Web.Servlet : 스프링 MVC 기능을 제공. 전통적인 스프링 MVC와 최신 @MVC기능이 포함.

Web, Context.Support – 필수모듈, XML을 이용한 뷰나 메시지 컨버트 사용시 선택적으로 OXM 필요

Web.Portlet : Portlet(재 사용이 가능한 웹 구성요소) 개발에 사용하는 스프링 모듈. Web.Servlet-필수 의존모듈.

Web.Struts : 스트럿츠 1.x를 지원하는 모듈, Web 모듈에 의존

Spring 모듈의 의존관계

JMS : 스프링의 JMS(Java Message Service) 지원 .
Transaction 모듈에 의존.

Aspect : AspectJ AOP 기능을 사용할 때 필요한 모듈. 스프링이 직접 제공하는 AspectJ로 만든 기능은 @Configurable을 이용한 도메인 오브젝트 DI기능, JPA 예외 변환기, AspectJ방식의 트랜잭션 기능 등이 있다.
JPA 지원 기능 사용시 ORM, 트랜잭션 사용시 Transaction 에 의존.

DI(Dependency Injection)

DI(Dependency Injection)

- 스프링 컨테이너가 지원하는 핵심 개념 중의 하나
- 객체간의 의존 관계를 객체 자신이 아닌 외부의 조립기 (스프링컨테이너)가 수행 해 준다는 개념.

DI(용어정리)

IoC(Inversion of Control) : 제어역행

어떠한 것을 하도록 만들어 프레임워크에 제어의 권한을 넘김으로써 클라이언트 코드가 신경 써야 할 것을 줄이는 전략. 이것을 제어가 역전 되었다 라고 한다. 일반적으로 라이브러리는 프로그래머가 작성하는 클라이언트 코드가 라이브러리의 메소드를 호출해서 사용하는 것을 의미 .

IoC는 프레임워크의 메소드가 사용자의 코드(프로그래머가작성)를 호출 한다는데 있다.

DI(용어정리)

DI(Dependency Injection) : 의존성주입

- DI는 스프링 컨테이너가 지원하는 핵심 개념 중 하나
- DI는 객체 사이의 의존관계를 객체 자신이 아닌 외부의 조립기(스프링 컨테이너)가 수행한다는 개념
- 스프링은 설정 파일이나 어노테이션을 이용하여 객체 간의 의존관계를 설정할 수 있다.

DI(용어정리)

IoC(또는 DI) Container

- POJO(Plain Old Java Object)의 생성, 초기화, 서비스 소멸에 관한 모든 권한을 가지면서 POJO의 생명주기를 관리한다.
- 개발자들이 직접 POJO를 생성 할 수 도 있지만, 모든 권한을 Container에게 맡긴다.
- Transaction, Security 추가적인 기능을 제공한다. AOP 기능을 이용하여 새로운 Container 기능을 추가하는 것이 가능하다.

DI(용어정리)

마틴 파울러는 2004년의 글에서 제어의 어떤 측면이 역행되는 것인지에 대한 의문을 제기했다. 그는 의존하는 객체를 역행적으로 취득하는 것이라는 결론을 내렸다. 그는 그와 같은 정의에 기초하여 제어 역행이라는 용어에 좀더 참신한 ‘의존성 주입(dependency injection)’이라는 이름을 지어줬다.

모든 어플리케이션은 비즈니스 로직을 수행하기 위해 서로 협업하는 둘 또는 그 이상의 클래스들로 이뤄진다. 전통적으로 각 객체는 협업할 객체의 참조를 취득해야 하는 책임이 있다. 이것이 의존성이다. 이는 결합도가 높으며 테스트하기 어려운 코드를 만들어 낸다.

IoC를 적용함으로써 객체들은 시스템 내의 각 객체를 조정하는 어떤 외부의 존재에 의해 생성 시점에서 의존성을 부여 받는다. 즉 의존성이 객체로 주입(inject)된다는 말이다. 따라서 IoC는 한 객체가 협업해야 하는 다른 객체의 참조를 취득하는 방법에 대한 책의 역행이라는 의미를 갖는다.

결론 : IoC로 부터 DI개념이 생겼으며 두 개념은 같은 것임.

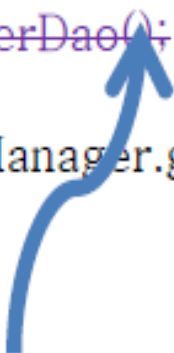
DI

기존의 방식

```
CustomerManager customerManager = new CustomerManager();  
CustomerDao dao = new CustomerDao();  
customerManagere.setDao(dao);  
Customer customer = customerManager.getCustomer(suninet@naver.com);
```

Spring의 방식

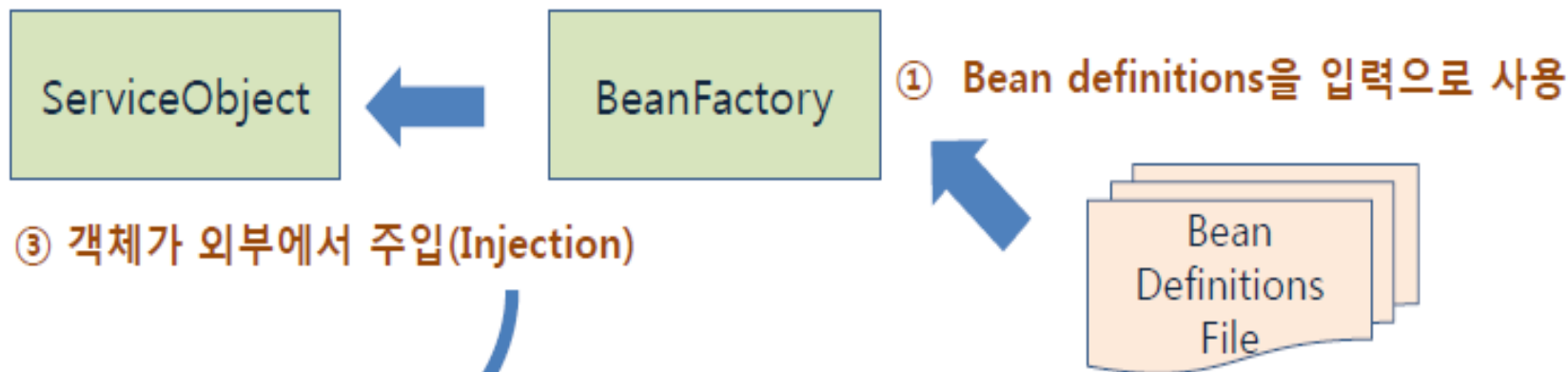
```
CustomerManager customerManager = new CustomerManager();  
CustomerDao dao = new CustomerDao();  
customerManagere.setDao(dao);  
Customer customer = customerManager.getCustomer(suninet@naver.com);
```



Spring 컨테이너가 처리, 컨테이너가 처리할 수 있는 메커니즘이 필요, XML 파일에 정의함

DI

② 내부적으로 객체를 생성



③ 객체가 외부에서 주입(Injection)

```
public class CustomerManager {  
    private CustomerDao customerDao;  
    public void setCustomerDao(CustomerDao dao) {  
        this.customerDao = dao;  
    }  
}
```

Bean 설정방법(applicationContext.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<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-2.5.xsd">

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions go here -->

</beans>
```

DI

Spring IoC container를 직접 인스턴스화하기

인스턴스화 하기

```
ApplicationContext context = new ClassPathXmlApplicationContext(  
    new String[] {"services.xml", "daos.xml"});
```

```
// an ApplicationContext is also a BeanFactory (via inheritance)  
BeanFactory factory = context;
```

빈을 얻어오기

```
MyBean myBean = (MyBean)factory.getBean("myBean");
```

DI

설정 파일 읽어오는 방법

FileSystemResource

절대경로 /상대 경로 둘다 가능 함, WWW, / 둘다 가능함

```
BeanFactory factory =  
new XmlBeanFactory(new FileSystemResource("WebContent/WEB-  
INF/spring-config/hello.xml"));
```

ClassPathResource

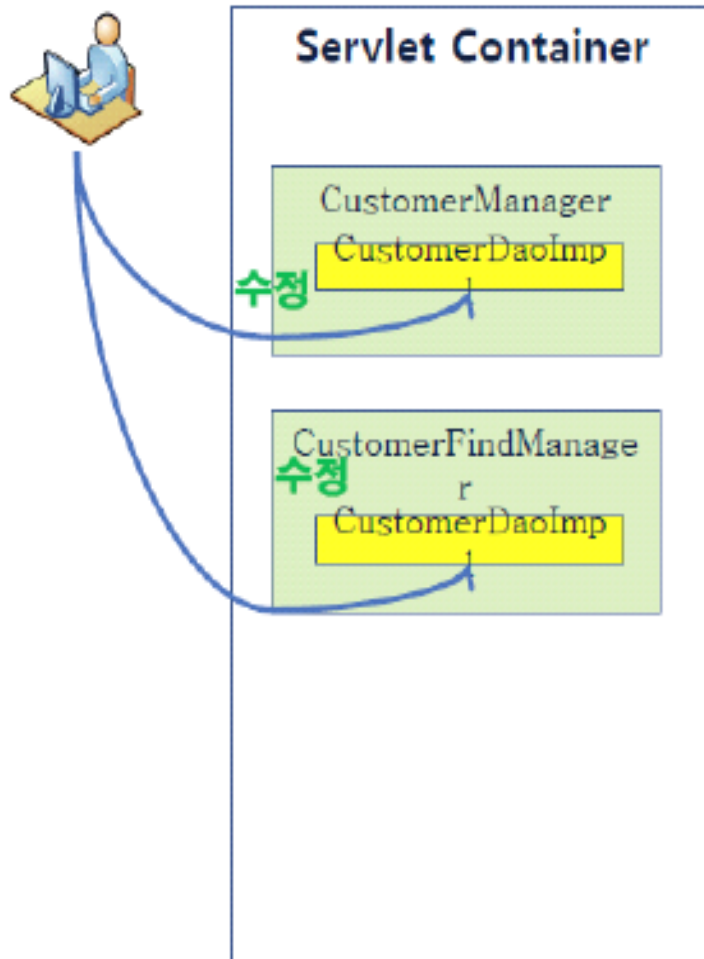
classpath 내에 있는 리소스 검색시 사용

```
BeanFactory factory =  
new XmlBeanFactory(new  
ClassPathResource("edu/biz/helloworld/hello.xml"));
```

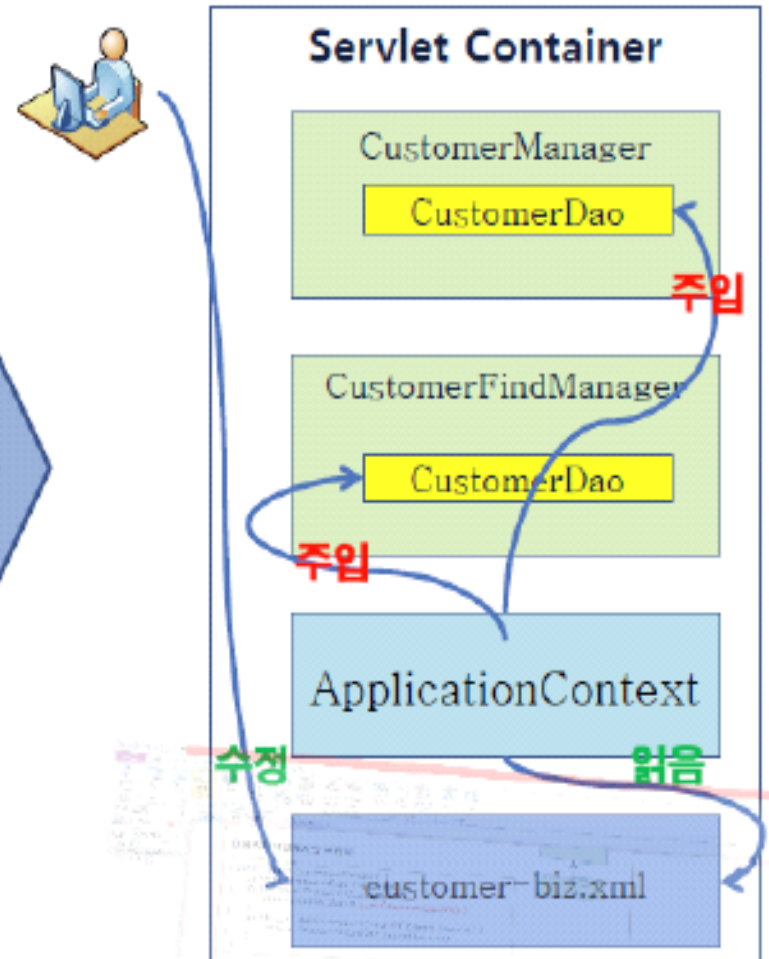
DI

IoC를 사용한 경우와 그렇지 않은 경우 비교

IoC를 사용하지 않는 경우



IoC를 사용한 경우



유연한 객체 지향적인 프로그램이 되기 위해서

설계원칙

- 자주 변경되는 구상클래스(Concrete class)에 의존하지 마라
- 어떤 클래스를 상속받아야 한다면, 기반 클래스를 추상 클래스로 만들어라
- 인터페이스를 만들어서 이 인터페이스에 의존하라
→ DIP(Dependency Inversion Principle)

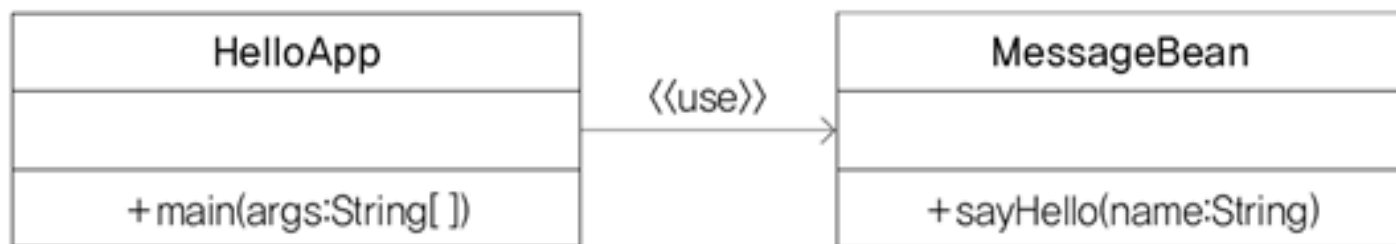
나쁜 품질의 코드

- 경직성: 무엇이든 하나를 바꿀 때 마다 다른 것도 바꿔야 한다.
- 부서지기쉬움: 시스템에서 한 부분을 변경하면 그것과 전혀 상관없는 다른 부분이 오류 발생
- 부동성: 시스템을 여러 컴포넌트로 분해해서 재사용하기 힘들다
- 끈끈함: 개발환경이 편집-컴파일-테스트 순환을 한번도는데 시간이 엄청나게 길다.
- 쓸데없이 복잡함: 괜히 머리를 굴려서 짠 코드 구조가 굉장히 많다.
- 필요없는 반복: copy & paste
- 불투명함: 코드를 만든 의도에 대한 설명을 볼때 그 설명에 '표현이 고인다'라는 말이 줄어들림

DI

예_1)

첫 번째 예제는 단순한 샘플 어플리케이션으로 sample1 패키지에 MessageBean과 HelloApp 라는 2개의 클래스가 있다.



▲ sample1 클래스 그림

MessageBean 클래스는 멤버로 sayHello() 메소드를 한 개 가지며, 이 메소드는 '이름' 을 String형 인수로 받아서 화면에 'Hello, ○○!' 라고 출력한다.

HelloApp 클래스는 main() 메소드 안에서 MessageBean의 인스턴스를 생성하여 sayHello() 메소드를 호출할 뿐이다.

◆ MessageBean.java

```
package sample1;

public class MessageBean {

    public void sayHello(String name) {
        System.out.println("Hello," + name + "!");
    }

}
```

◆ HelloApp.java

```
package sample1;

public class HelloApp {

    public static void main(String[] args) {
        MessageBean bean = new MessageBean();
        bean.sayHello("Spring");
    }

}
```

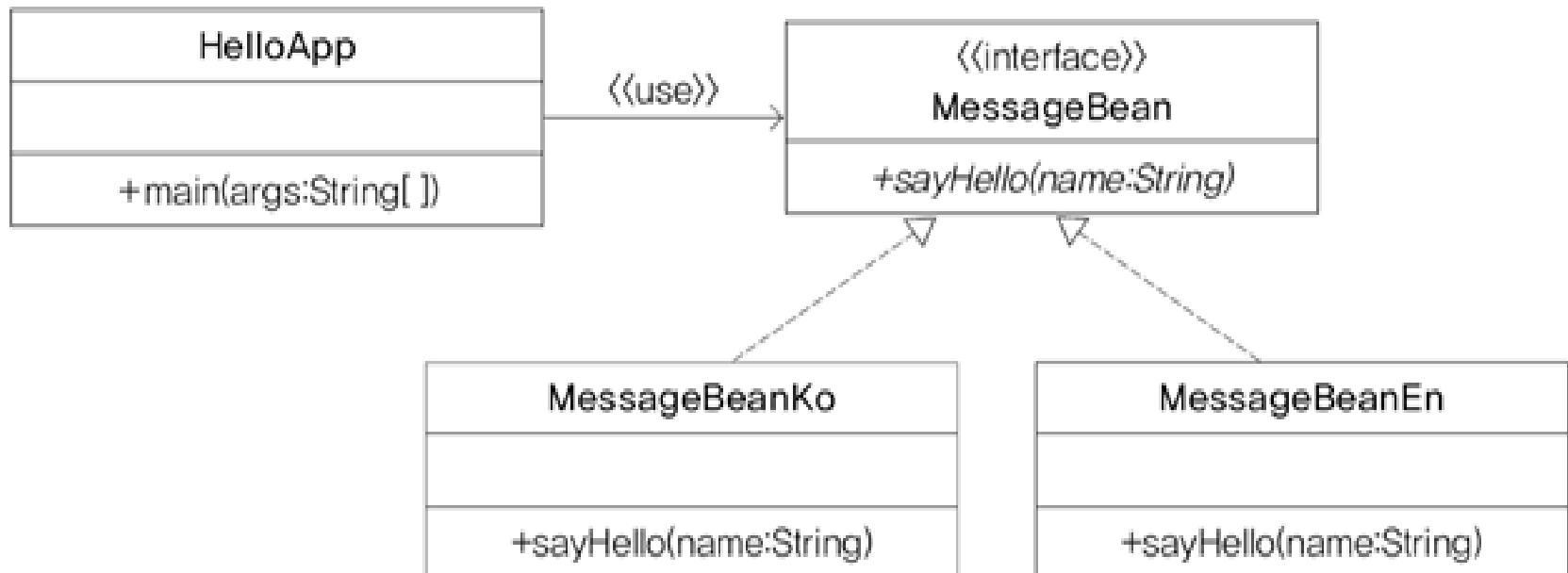
HelloApp는 MessageBean 클래스의 객체를 직접 생성해서 sayHello 메소드를 호출하고 있는데 이런 구현에서 HelloApp가 MessageBean에 강하게 의존하게 된다.

클래스간 결합이 강해지고 의존하고 있는 클래스에 변경이 생기면, 코드를 수정해야 하는 범위가 넓어질 가능성이 높아짐.

DI

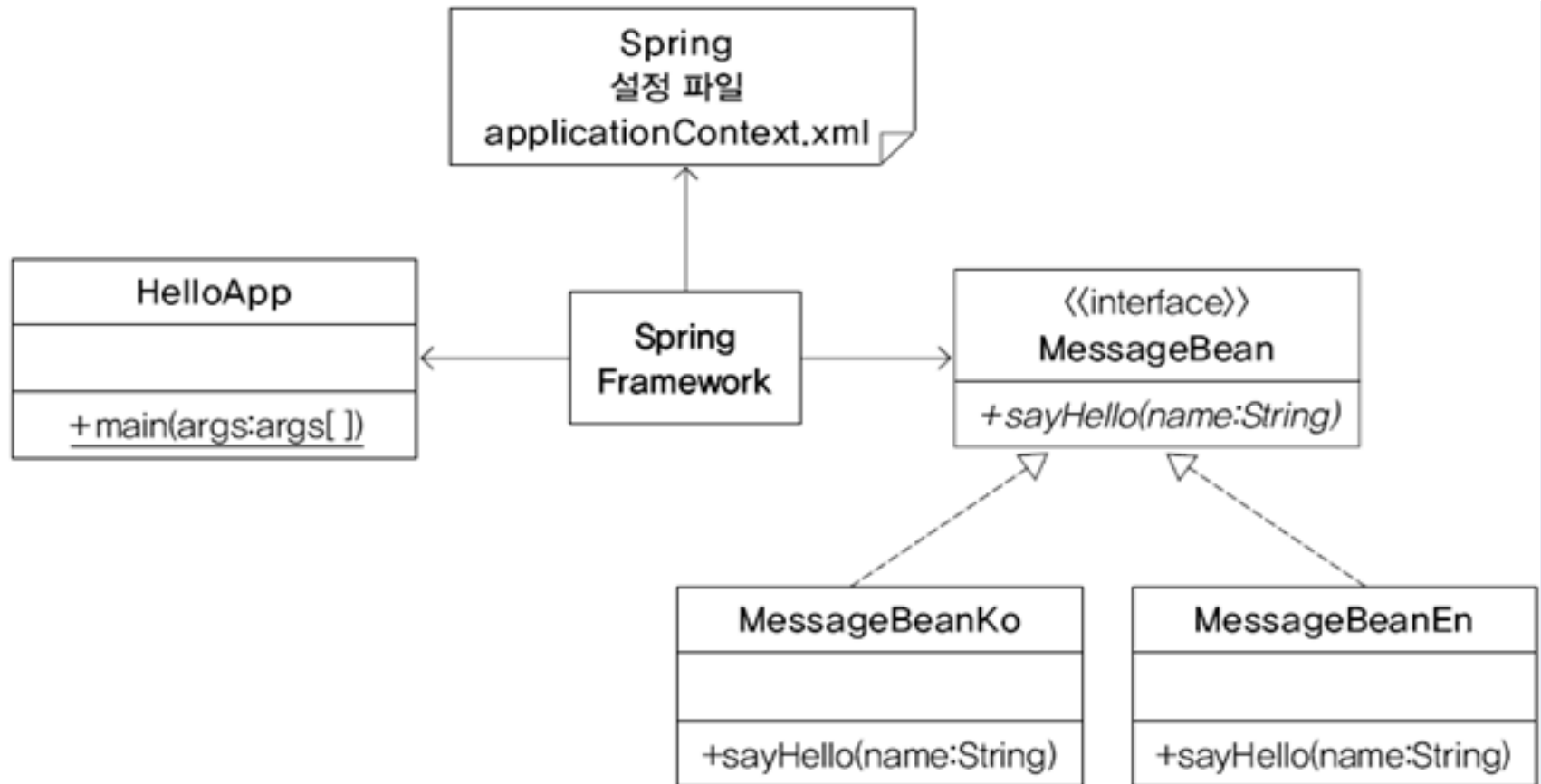
예_2)

두 번째 샘플 어플리케이션은 인터페이스를 이용하여 클래스 사이의 의존 관계를 약하게 설정하고 있다. sample2 패키지에 MessageBean 인터페이스를 마련하고, 이 인터페이스를 구현한 두 개의 클래스(MessageBeanEn과 MessageBeanKo)를 작성하였다.



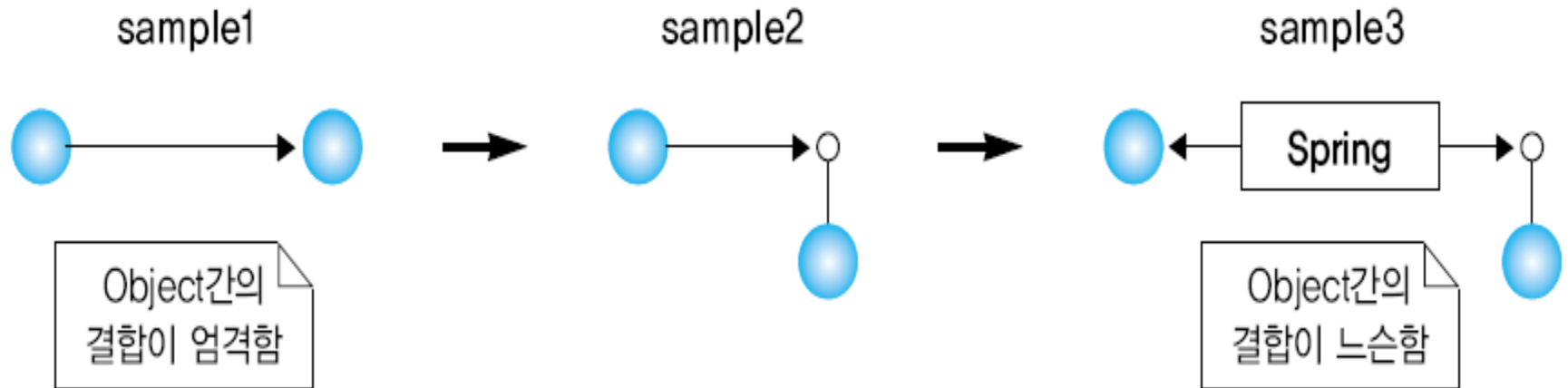
DI

예_3)

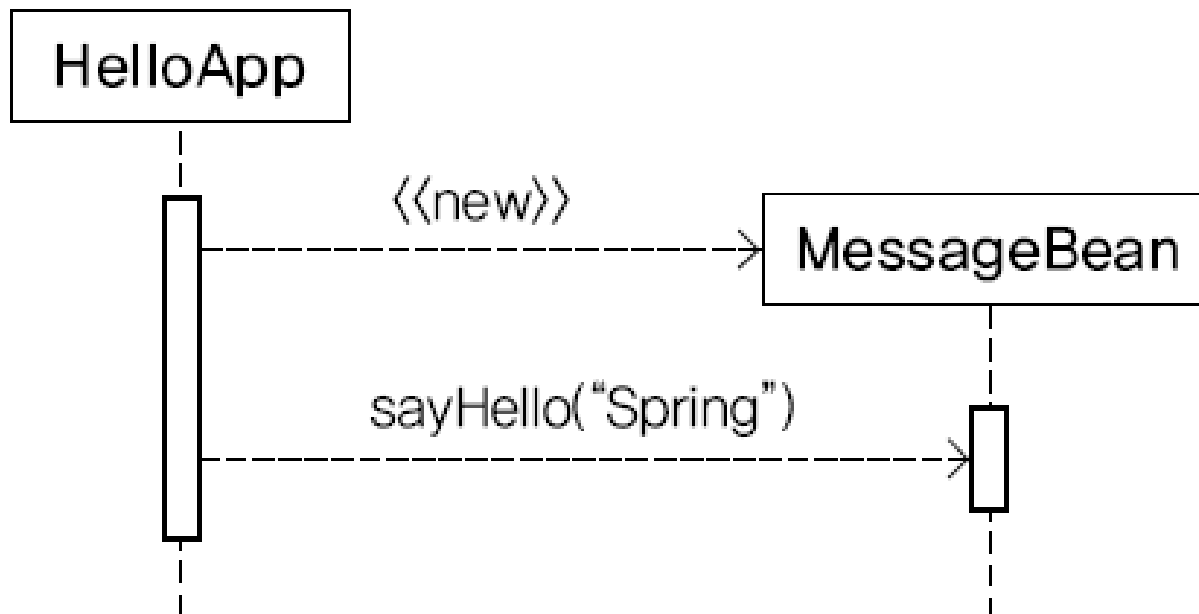


▲ sample3 클래스 그림(스프링 사용)

DI



DI



▲ Sample 샘플 어플리케이션 시퀀스 그림

DI

예제_2 코드

◆ MessageBean.java

```
package sample2;  
  
public interface MessageBean {  
    void sayHello(String name);  
}
```

◆ MessageBeanEn.java

```
package sample2;  
  
public class MessageBeanEn implements MessageBean {  
    @Override  
    public void sayHello(String name) {  
        System.out.println("Hello," + name + "!");  
    }  
}
```


DI

예제_2 코드

```
MessageBean bean = new MessageBeanEn( );  
bean.sayHello("Spring");
```

수정하지 않아도 되는 코드

```
MessageBean bean = new MessageBeanKo( );  
bean.sayHello("Spring");
```

아직은 수정해야 하는 코드가 존재

수정하지 않아도 되는 코드

예제 3 코드

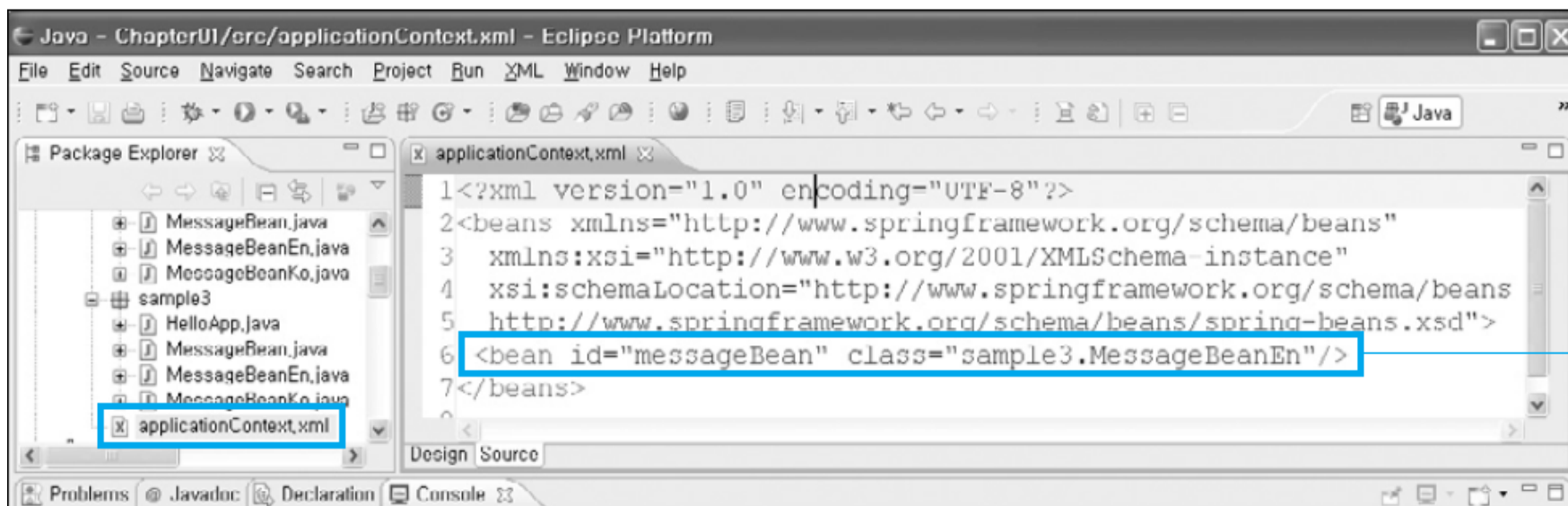
◆ HelloApp.java

```
package sample3;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class HelloApp {
    public static void main(String[] args) {
        Resource resource = new FileSystemResource("applicationContext.xml"); ..... ❶
        BeanFactory factory = new XmlBeanFactory(resource);
        MessageBean bean = (MessageBean)factory.getBean("messageBean"); ..... ❷
        bean.sayHello("Spring");
    }
}
```

springbean문서 만들기

```
<bean id = "messageBean" class = "sample3.MessageBeanEn"/>
```



▲ applicationContext.xml 파일에 <bean> 요소 추가

DI

```
<bean id="ob" class="sample3.MessageBeanEn">  
</bean>
```

```
<bean id="ob" class="sample3.MessageBeanEn"/>
```

```
<bean name="ob" class="sample3.MessageBeanEn"/>
```

DI

```
ApplicationContext context =
```

```
    new ClassPathXmlApplicationContext("applicationContext.xml");
```

```
MessageBean bean = (MessageBean) context.getBean("ob");
```

```
bean.sayHello("길동");
```

* 스프링 3.0

```
MessageBean bean = context.getBean("ob", MessageBean.class);
```

DI

Bean객체 얻어오는 방법

//1.

```
Resource resource = new FileSystemResource("src/applicationContext.xml");  
BeanFactory factory = new XmlBeanFactory(resource);
```

//2.

```
Resource resource = new ClassPathResource("applicationContext.xml");  
BeanFactory factory = new XmlBeanFactory(resource);
```

//3.

```
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  
BeanFactory factory = context;
```

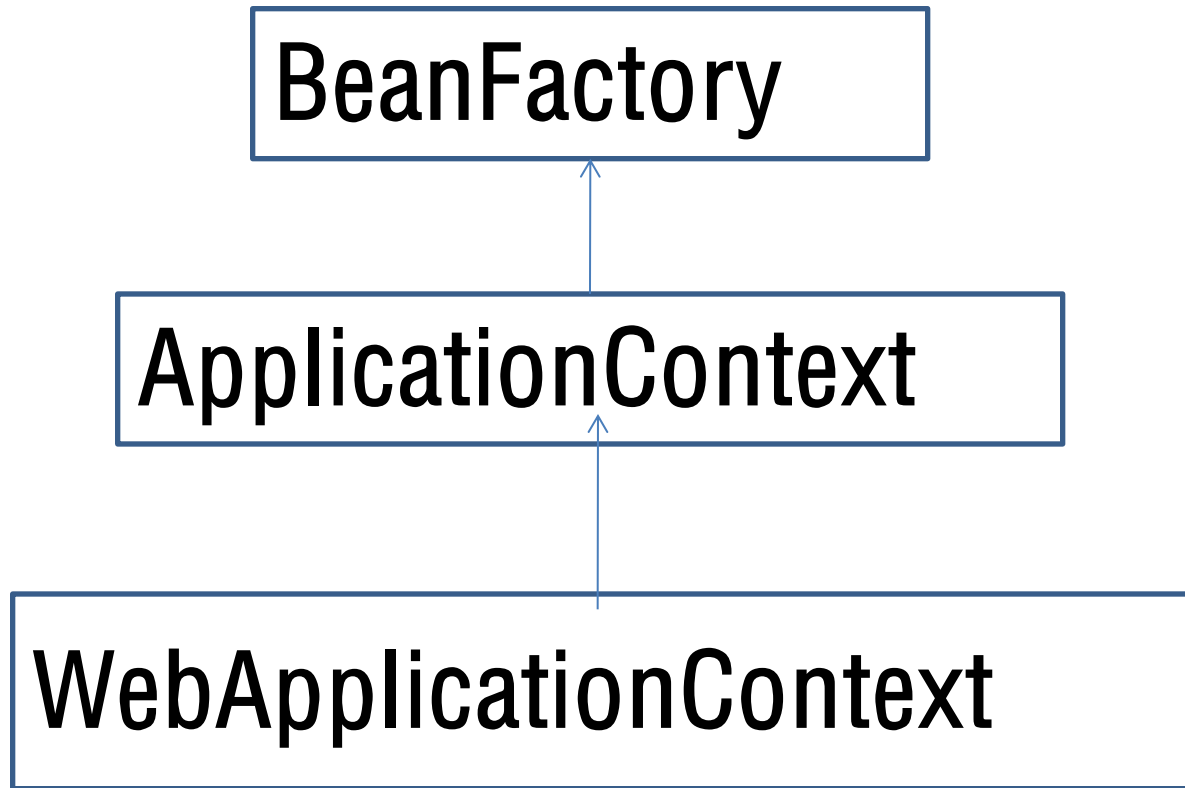
```
MessageBean bean = (MessageBean)factory.getBean("en");  
bean.sayHello("heejung");
```

//4.

```
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  
MessageBean m = (MessageBean)context.getBean("en");  
m.sayHello("희정");
```

DI

Spring 컨테이너



DI

BeanFactory

- 빈 객체를 관리하고 객체간의 의존 관계를 설정해 주는 가장 단순한 컨테이너
- 구현 클래스
 - ➡ XmlBeanFactory
 - 외부 설정정보를 읽어와 빈 객체를 생성한다.
 - Resource를 이용하여 설정정보를 전달한다.

Resource resource = **new**

ClassPathResource("applicationContext.xml");

BeanFactory factory = **new XmlBeanFactory(resource);**

MessageBean bean = (MessageBean)factory.getBean("ob");

DI

ApplicationContext

- BeanFactory 인터페이스를 상속
- 빈 객체 라이프 사이클
- 파일과 같은 자원 처리 추상화
- 메시지 지원 및 국제화 지원
- 이벤트 지원
- Xml 스키마 확장

BeanFactory, 보다는 ApplicationContext를 주로 사용

DI

WebApplicationContext

- 웹 어플리케이션을 위한 ApplicationContext
- 하나의 웹 어플리케이션 마다 한 개 이상의 WebApplicationContext를 가질 수 있다.

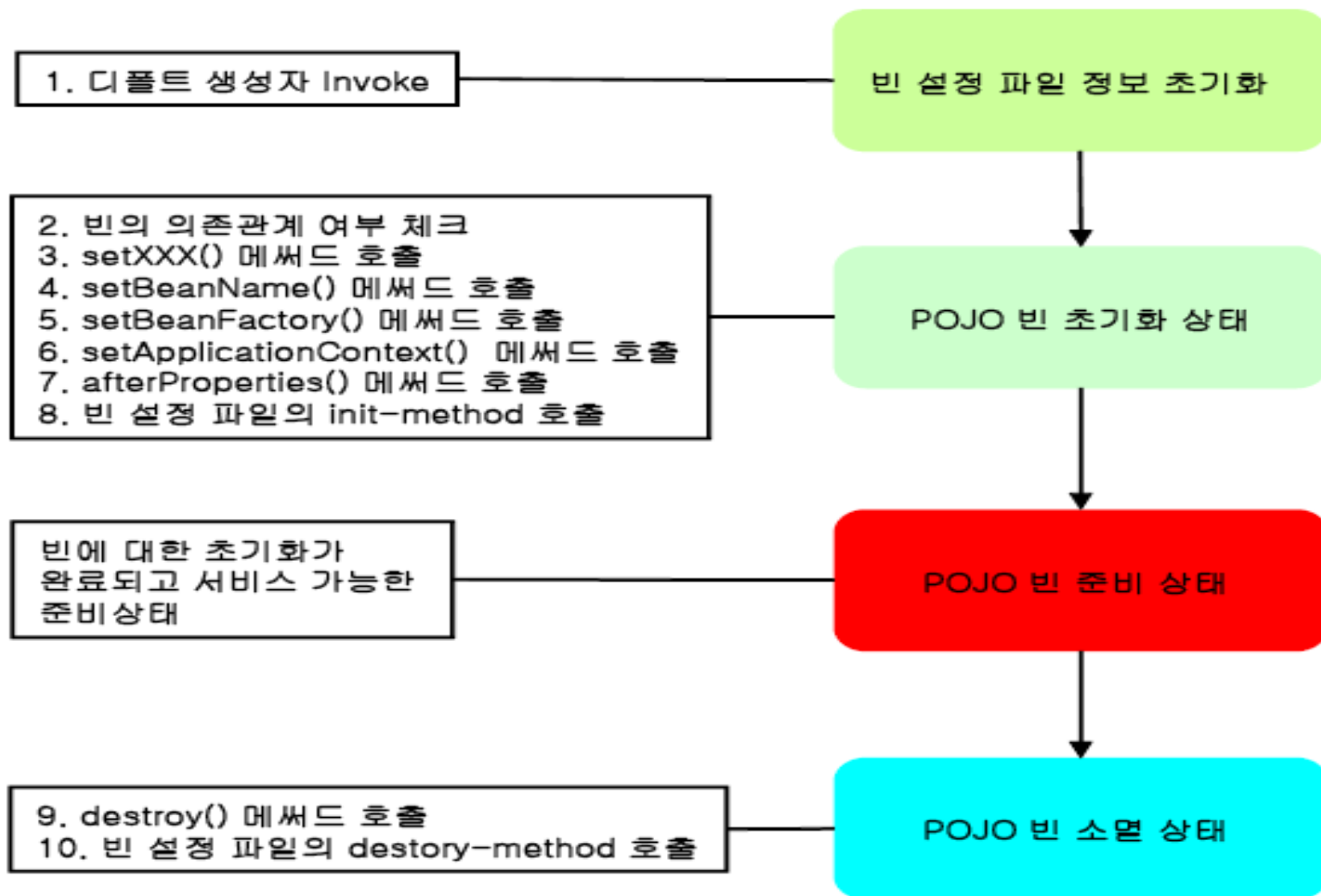
DI

ApplicationContext,
WebApplicationContext를 구현한 클래스

- **ClassPathXmlApplicationContext**
- **FileSystemXmlApplicationContext**
- **XmlWebApplicationContext**

DI

IoC컨테이너가 해주는 일- 생성주기 관리



DI

Bean 생성과 의존 관계 설정

- **생성자 방식**
- **프로퍼티 설정 방식**

생성자 방식

```
public class WriteArticleServiceImpl
    implements WriteArticleService {

    private ArticleDao articleDao;

    public WriteArticleServiceImpl(ArticleDao articleDao) {
        this.articleDao = articleDao;
    }
}
```

생성자가 전달 받은 값이 기본자료형

```
<bean id="systemMonitor"  
class="madvirus.spring.chap02.SystemMonitor">  
  <constructor-arg>  
    <value>10</value>  
  </constructor-arg>  
</bean>
```

```
<bean id="systemMonitor"  
class="madvirus.spring.chap02.SystemMonitor">  
  <constructor-arg value="10"/>  
</bean>
```

생성자의 매개변수가 2개 이상

```
public class SystemMonitor {  
    private long periodTime;  
    private SmsSender sender;  
    public SystemMonitor(long periodTime, SmsSender sender) {  
        super();  
        this.periodTime = periodTime;  
        this.sender = sender;  
    }  
}
```

```
<bean id="systemMonitor"  
class="madvirus.spring.chap02.SystemMonitor">  
    <constructor-arg value="10"/>  
    <constructor-arg ref="smsSender"/>  
</bean>
```


생성자가 두 개 이상일때

```
public class Executor {  
    private Worker worker;  
    public Executor(Worker worker) {  
        this.worker = worker;  
        System.out.println("생성자 1");  
    }  
    public Executor(String run) {  
        System.out.println("생성자 2");  
    }  
}
```

생성자가 두 개 이상일때

```
<bean id="worker"
      class="madvirus.spring.chap02.Worker"/>

<bean id="executor1" class="madvirus.spring.chap02.Executor">
  <constructor-arg>
    <ref local="worker"/>
  </constructor-arg>
</bean>

<bean id="executor2" class="madvirus.spring.chap02.Executor">
  <constructor-arg>
    <value>홍길동</value>
  </constructor-arg>
</bean>
```

객체 생성시 전달되는 파라미터의 값에 따라 적합한 생성자가 결정됨.

```
<bean id="writeArticleService"  
class="madvirus.spring.chap02.WriteArticleServiceImpl">  
    <constructor-arg>  
        <ref local="articleDao"/>  
    </constructor-arg>  
</bean>
```

```
<bean id="articleDao"  
class="madvirus.spring.chap02.MysqlArticleDao"/>
```

```
<bean id="writeArticleService"  
class="madvirus.spring.chap02.WriteArticleServiceImpl">  
    <constructor-arg ref="articleDao"/>  
</bean>
```

생성자 방식

```
public class WriteArticleServiceImpl
    implements WriteArticleService {

    private ArticleDao articleDao;

    public WriteArticleServiceImpl(ArticleDao articleDao) {
        this.articleDao = articleDao;
    }
}
```

<value>에 전달된 값은 기본적으로 String으로 처리

```
public class JobExecutor {  
    public JobExecutor(String name, int seconds) {  
        System.out.println("생성자 호출1");  
    }  
  
    public JobExecutor(String name, long milliseconds) {  
        System.out.println("생성자 호출2");  
    }  
  
    public JobExecutor(String name, String seconds) {  
        System.out.println("생성자 호출3");  
    }  
}
```

```
<constructor-arg>  
    <value>홍길동</value>  
</constructor-arg>  
<constructor-arg>  
    <value>1000</value>  
</constructor-arg>
```

숫자형태의 값이 전달되더라도 기본적으로 String으로 처리함.

```

<bean id="jobExecutor"
class="madvirus.spring.chap02.JobExecutor">
    <constructor-arg>
        <value>홍길동</value>
    </constructor-arg>
    <constructor-arg>
        <value>1000</value>
    </constructor-arg>
</bean>

```

- ** String type 생성자가 없다면, 전달되는값 1000을 평가하여 int로 적용됨
- ** 만약 long으로 적용하려면 다음과 같이 설정한다.

<value type="long">1000</value>

<constructor-arg value="1000" type="long"/>

프로퍼티 설정 방식

```
public class WriteArticleServiceImpl
    implements WriteArticleService {

    private ArticleDao articleDao;

    public void setArticleDao(ArticleDao articleDao) {
        this.articleDao = articleDao;
    }
}
```

```
<bean id="writeArticleService"
class="madvirus.spring.chap02.WriteArticleServiceImpl">
  <property name="articleDao">
    <ref local="mysqlArticleDao"/>
  </property>
</bean>

<bean id="mysqlArticleDao"
class="madvirus.spring.chap02.MysqlArticleDao"/>
```



```
<bean id="lockingFailManager"
class="madvirus.spring.chap02.PessimisticLockingFailManager">
  <property name="retryCount">
    <value>3</value>
  </property>
</bean>
```

```
<bean id="monitor"
class="madvirus.spring.chap02.SystemMonitor">
  <property name="periodTime" value="10"/>
  <property name="sender" ref="smsSender"/>
</bean>
```

Bean 태그들의 속성

<bean> 태그

<bean> 태그는 컨텍스트 의 기본저장단위인 bean 을 생성하는 태그이다.

bean 태그의 속성

- id
- class
- singleton
- init-method
- destroy-method
- factory-method
- lazy-init

Bean 태그들의 속성

1. **id** : bean 태그의 이름이며, 해당 bean 에 접근하기 위한 key.
2. **class** : bean 의 실제 클래스이며, 항상 full name 으로 선언되어야 한다.
3. **singleton** : true/false 값을 가지며, 해당 bean 을 singleton 으로 유지할 것인지 여부를 결정. 이 속성 를 지정하지 않았을 경우의 default 값은 true 이며, false 로 지정하였을 경우, `getBean()` 메소드 호출시마다 해당 bean 객체가 생성.
4. **init-method** : 해당 bean 이 초기화된 이후 context 에 저장되기 직전에 호출되는 초기화 메소드 이름. `init-method="init"` 식으로 기술.

Bean 태그들의 속성

5. **destroy-method** : 해당 bean 이 더 이상 사용되지 않을 때 호출되는 메소드. 때문에, `destroy-method="destroy"` 식으로 기술.
6. **factory-method** : bean 의 초기화시 생성자 호출이 아니라 특정한 메소드 호출을 통해 인스턴스를 받고자 할 때 사용되는 메소드. `factory-method="getInstance"` 식으로 기술.
7. **lazy-init** : true/false 값을 가지며, 해당 bean 이 호출되기 이전까지는 초기화될 것인지 여부를 지정. 기술되지 않았을 경우의 default 값은 false

Bean 태그들의 속성

```
<bean id="memberBeanId" class="bean.MemberBean"  
scope="prototype"  
    init-method="init"  
    destroy-method="destroy"  
    lazy-init="default">
```

list 타입과 배열

```
public class ProtocolHandler {  
    private List<Filter> filters;  
  
    public void setFilters(List<Filter> filters) {  
        this.filters = filters;  
    }  
}
```

```
<bean id="handler" class="madvirus.spring.chap02.ProtocolHandler">
  <property name="filters">
    <list>
      <ref bean="encryptionFilter"/>
      <ref bean="zipFilter"/>
      <bean class="madvirus.spring.chap02.HeaderFilter"/>
    </list>
  </property>
</bean>
```

```
<bean id="zipFilter" class="madvirus.spring.chap02.ZipFilter"/>
<bean id="encryptionFilter" class="madvirus.spring.chap02.EncryptionFilter"/>
```

List에 저장될 객체가 String이 아닌 경우

```
public class PerformanceMonitor {  
    private List deviations;  
    |  
    public void setDeviations(List deviations) {  
        this.deviations = deviations;  
    }  
}
```



```
<bean id="perperformanceMonitor"  
class="madvirus.spring.chap02.PerformanceMonitor">  
  <property name="deviations">  
    <list value-type="java.lang.Double">  
      <value>0.2</value>  
      <value>0.3</value>  
    </list>  
  </property>  
</bean>
```

```
<bean id="perperformanceMonitor2"  
class="madvirus.spring.chap02.PerformanceMonitor">  
  <property name="deviations">  
    <list>  
      <value type="java.lang.Double">0.2</value>  
      <value type="java.lang.Double">0.3</value>  
    </list>  
  </property>  
</bean>
```

```

public class PerpformanceMonitor {
    private List<Double> deviations;

    public void setDeviations(List deviations) {
        this.deviations = deviations;
    }
}

```

```

<bean id="perpformanceMonitor"
class="madvirus.spring.chap02.PerpformanceMonitor">
    <property name="deviations">
        <list>
            <value>0.2</value>
            <value>0.3</value>
        </list>
    </property>
</bean>

```

**** 자바 5 부터 generic을 이용할 경우 타입을 명시 하지않아도 알맞게 변환됨.**

실습하기

1. bean 속성 과 bean 생성시점
2. 생성자
3. property
4. properties
5. List

실습하기(bean생성)

```
<!-- 1. bean 생성
      scope속성 = prototype는 getBean할때마다 새로운객체생성
                  singleton는 getBean여러번해도 같은객체
-->
<bean id="bean" class="soa.spring.di.MemberBean"
scope="prototype"/>
```

실습하기

```
/**
 * 1. BeanFactory를 이용한 설정파일 가져오기
 * (getBean을 통해 객체를 가져올때 객체가 생성됨.)
 * */
/*Resource resource =
    new ClassPathResource("soa/spring/di/applicationContext.xml");
BeanFactory f = new XmlBeanFactory(resource);

MemberBean m = f.getBean("bean", MemberBean.class);
MemberBean m2 = f.getBean("bean", MemberBean.class);

System.out.println("m = " + m);
System.out.println("m2 = " + m2);*/
```

```
/**
 * 2.ApplicationContext를 이용한 설정파일 가져오기
 * (application이 생성될때 모든 bean 이 함께 생성됨
 * => scope가 singleton일때만가능함.(기본값)
 * )
 * */
```

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("soa/spring/di/applicationContext.xml");

MemberBean m = context.getBean("bean", MemberBean.class);
MemberBean m2 = context.getBean("bean", MemberBean.class);

System.out.println("m = " + m);
System.out.println("m2 = " + m2);
```

실습하기(생성자)

<!-- 2. 생성자를 통한 DI -->

```
<bean id="bean2" class="soa.spring.di.MemberBean">
  <constructor-arg>
    <value>즐거운 금요일!</value>
  </constructor-arg>
</bean>
```

<!-- 3. 생성자를 통한 DI(인수 2개)
type="int"는 인수의 타입을 설정
-->

```
<bean id="bean3" class="soa.spring.di.MemberBean">
  <constructor-arg value="주말은 spring과 함께~"/>
  <constructor-arg value="20" type="int"/>
</bean>
```

실습하기(생성자)

```
public class MemberBean {  
    private int age;  
    private String name;  
    private String message;  
  
    public MemberBean(){  
        System.out.println("MemberBean() 생성자 호출됨!");  
    }  
  
    public MemberBean(String message){  
        System.out.println("MemberBean(String message) 호출됨.");  
        this.message = message;  
    }  
  
    public MemberBean(String message , int age){  
        System.out.println("MemberBean(String message , int age) 호출됨.");  
        this.message= message;  
        this.age =age;  
    }  
  
    public MemberBean(String message , String message2){  
        System.out.println("MemberBean(String message , String message2) 호출됨");  
        this.message = message+" , " + message2;  
    }  
}
```

실습하기(생성자)

//1. 생성자를 통해 인자 전달

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("soa/spring/di/applicationContext.xml");
```

```
MemberBean m = context.getBean("bean2", MemberBean.class);
```

```
System.out.println("내용 : " + m.getMessage());
```

```
System.out.println("이름 : " + m.getName());
```

```
System.out.println("나이 : " + m.getAge());
```

```
System.out.println("=====");
```

```
MemberBean m2 = context.getBean("bean3", MemberBean.class);
```

```
System.out.println("내용 : " + m2.getMessage());
```

```
System.out.println("이름 : " + m2.getName());
```

```
System.out.println("나이 : " + m2.getAge());
```


실습하기(property)

```
<!-- 4.setXxx()를 이용한 DI(property) -->
```

```
<bean id="bean4" class="soa.spring.di.MemberBean">  
    <property name="age" value="26"/> <!-- setAge()호출됨 -->  
    <property name="name" value="장희정"/><!-- setName()호출됨 -->  
    <constructor-arg value="배고프다~~"/>  
</bean>
```

```
<!-- 5. ref를 이용한 DI(객체주입) -->
```

```
<bean id="dao" class="soa.spring.di.MemberDAO">  
    <constructor-arg ref="bean4"/>  
</bean>
```

```
<bean id="dao2" class="soa.spring.di.MemberDAO">  
    <property name="bean" ref="bean4"/><!-- setBean()호출됨. -->  
</bean>
```

실습하기 (property)

```
public int getAge() {  
    return age;  
}  
  
public void setAge(int age) {  
    this.age = age;  
}  
  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
public String getMessage() {  
    return message;  
}  
  
public void setMessage(String message) {  
    this.message = message;  
}
```

실습하기 (property)

//2. property를 이용한 인수 전달

```
MemberBean m4 = context.getBean("bean4", MemberBean.class);  
System.out.println("내용 : " + m4.getMessage());  
System.out.println("이름 : " + m4.getName());  
System.out.println("나이 : " + m4.getAge());
```

```
System.out.println("*****8888");
```

//3. ref(참조변수)를 이용한 인수전달

```
MemberDAO dao = context.getBean("dao", MemberDAO.class);  
dao.insert();
```

```
System.out.println("*****8888");
```

```
MemberDAO dao2 = context.getBean("dao2", MemberDAO.class);  
dao2.setInsert();
```

```
System.out.println("*****8888");
```

```
MemberBean m6 = context.getBean("bean6", MemberBean.class);  
System.out.println("내용 : " + m6.getMessage());  
System.out.println("이름 : " + m6.getName());  
System.out.println("나이 : " + m6.getAge());
```

실습하기(properties)

```
<!-- properties파일 선언하기 -->
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
    <array>
      <value>classpath:properties/member.properties</value>
    </array>
  </property>
</bean>
```

```
<!-- 6. properties설정 값을 이용한 DI -->
> <bean id="bean6" class="soa.spring.di.MemberBean">
  <constructor-arg value="${user.message}"/>
  <property name="age" value="${age}"/>
  <property name="name" value="${name}"/>
</bean>
```

실습하기(List)

```
<!-- 7. List - 기본형 -->
<bean id="listBean" class="soa.spring.di.ListBean">
    <property name="list">
        <list>
            <value>10</value>
            <value>20</value>
            <value>30</value>
        </list>
    </property>
</bean>
```

```
<!-- 8. List -객체주입 -->
<bean id="listBean2" class="soa.spring.di.ListBean">
    <property name="memberList">
        <list>
            <ref bean="bean4"/>
            <ref bean="bean6"/>
            <bean class="soa.spring.di.MemberBean"/>
        </list>
    </property>
</bean>
```

실습하기_2

* spring 프로젝트를 생성하여 작성함.

UserBean.java 클래스
id, name, addr, phone

ProductBean.java
p_no, p_price, p_name

OrderMessage interface
public void getOrder_Message();

OrderBeanImpl 클래스 ----> OrderMessage 구현
private int orderId
UserBean userBean,
ProductBean productBean,
String message;

*재정의된 getOrder_Message();에서
주문번호, 이름,전화번호, 상품, 가격, 메시지를 출력한다.

- * 위의 사항을 구현 한후
xml 문서에서 DI를 이용하여 값을 넣는다.
 - UserBean의 addr과 phone은 properties를 이용하여 주입함.
 - ProductBean의 p_no, p_price 는 properties를 이용하여 주입함.
- * 메인 클래스에서
OrderBeanImpl객체의 bean을 생성한 후
getOrder_Message()를 호출한다.
결과화면은
주문번호, 이름,전화번호, 상품, 가격, 메시지가 출력되면 됨.

실습하기(List)

```
public class ListBean {  
    private List<Integer> list;  
    private List<MemberBean> memberList;  
    public void setList(List<Integer> list) {  
        this.list = list;  
    }  
  
    public List<Integer> getList() {  
        return list;  
    }  
    public void setMemberList(List<MemberBean> memberList) {  
        this.memberList = memberList;  
    }  
  
    public List<MemberBean> getMemberList() {  
        return memberList;  
    }  
}
```

실습하기(List)

```
public static void main(String[] args) {
    ApplicationContext context =
        new ClassPathXmlApplicationContext("soa/spring/di/applicationContext.xml");

    /*ListBean bean = context.getBean("listBean", ListBean.class);
    List<Integer> list = bean.getList();
    for (Integer it : list) {
        System.out.println(it);
    }*/

    ListBean bean = context.getBean("listBean2", ListBean.class);
    List<MemberBean> list = bean.getMemberList();
    for (MemberBean it : list) {
        System.out.printf("age : %d , name : %s , message : %s\n",
            it.getAge(), it.getName(), it.getMessage());
    }
}
```


AOP

AOP

- 스프링은 DI 컨테이너로써 뿐만 아니라 AOP 프레임워크로서의 기능도 제공하고 있다. AOP(Aspect Oriented Programming : 관점지향 프로그래밍)는 최근 각광받는 새로운 프로그래밍 기법에 대한 개념이다

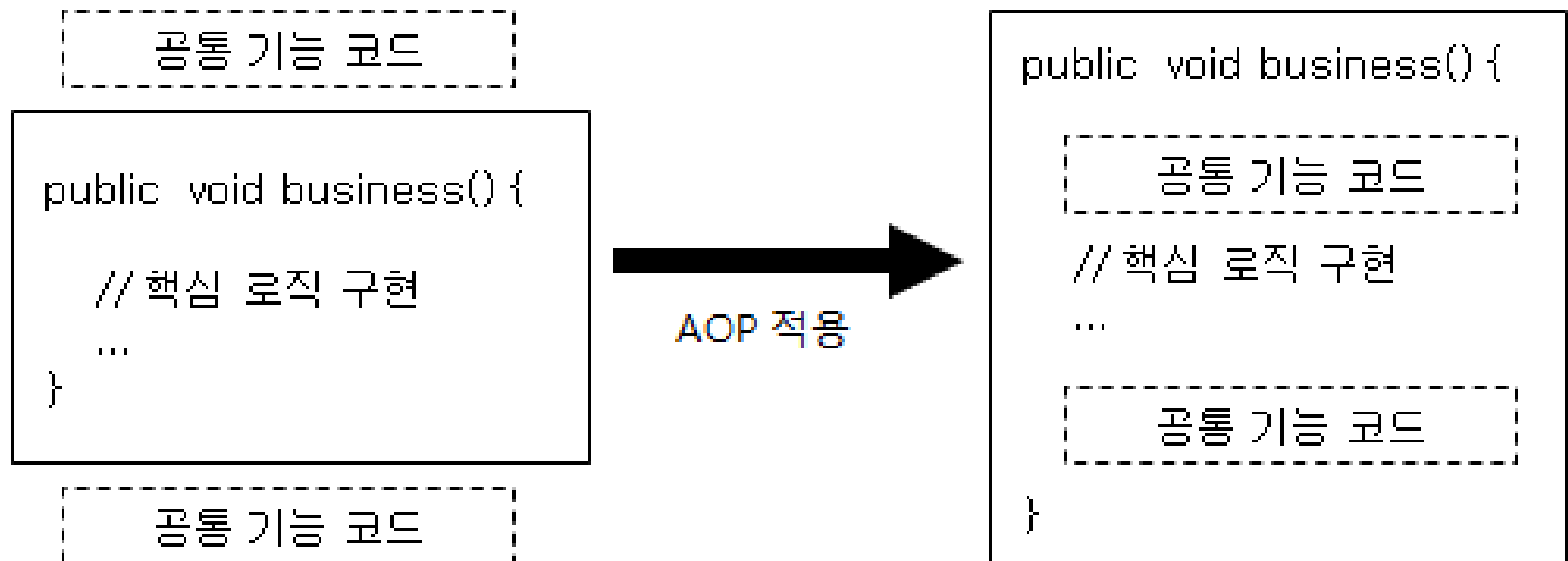
AOP

- 관점 지향 프로그래밍(Aspect Oriented Programming, 이하 AOP)은 결국 객체 지향 프로그래밍(Object Oriented Programming)의 뒤를 잇는 또 하나의 프로그래밍 언어 구조라고 생각될 수 있다.
- OOP를 더욱 OOP답게 만들어 준다.

AOP

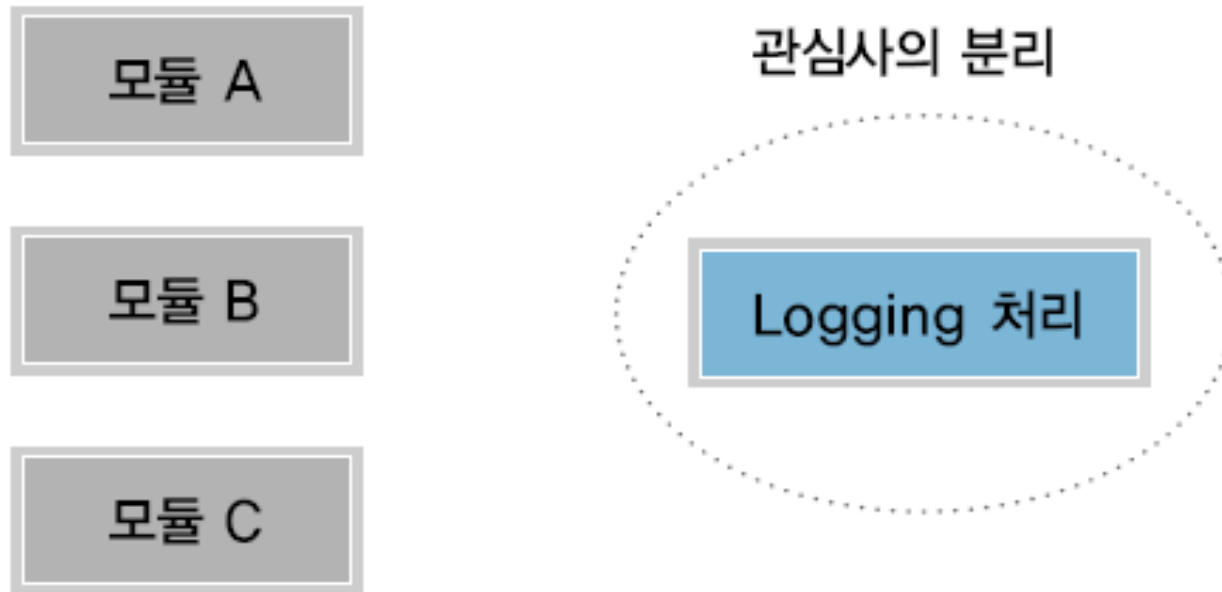
- Aspect 지향에서 중요한 개념은 「**횡단 관점의 분리(Separation of Cross-Cutting Concern)**」이다. 이에 대한 이해를 쉽게 하기 위해서 은행 업무를 처리하는 시스템을 예를 들어 보겠다.
- 은행 업무 중에서 계좌이체, 이자계산, 대출처리 등은 주된 업무(핵심 관점, 핵심 비즈니스 기능)로 볼 수 있다. 이러한 업무(핵심 관점)들을 처리하는데 있어서 「로깅」, 「보안」, 「트랜잭션」등의 처리는 **어플리케이션 전반에 걸쳐 필요한 기능**으로 핵심 비즈니스 기능과는 구분하기 위해서 **공통 관심 사항(Cross-Cutting Concern)**이라고 표현한다.

공통 관심 사항(cross-cutting concern)과 핵심 관심 사항(core concern)



AOP

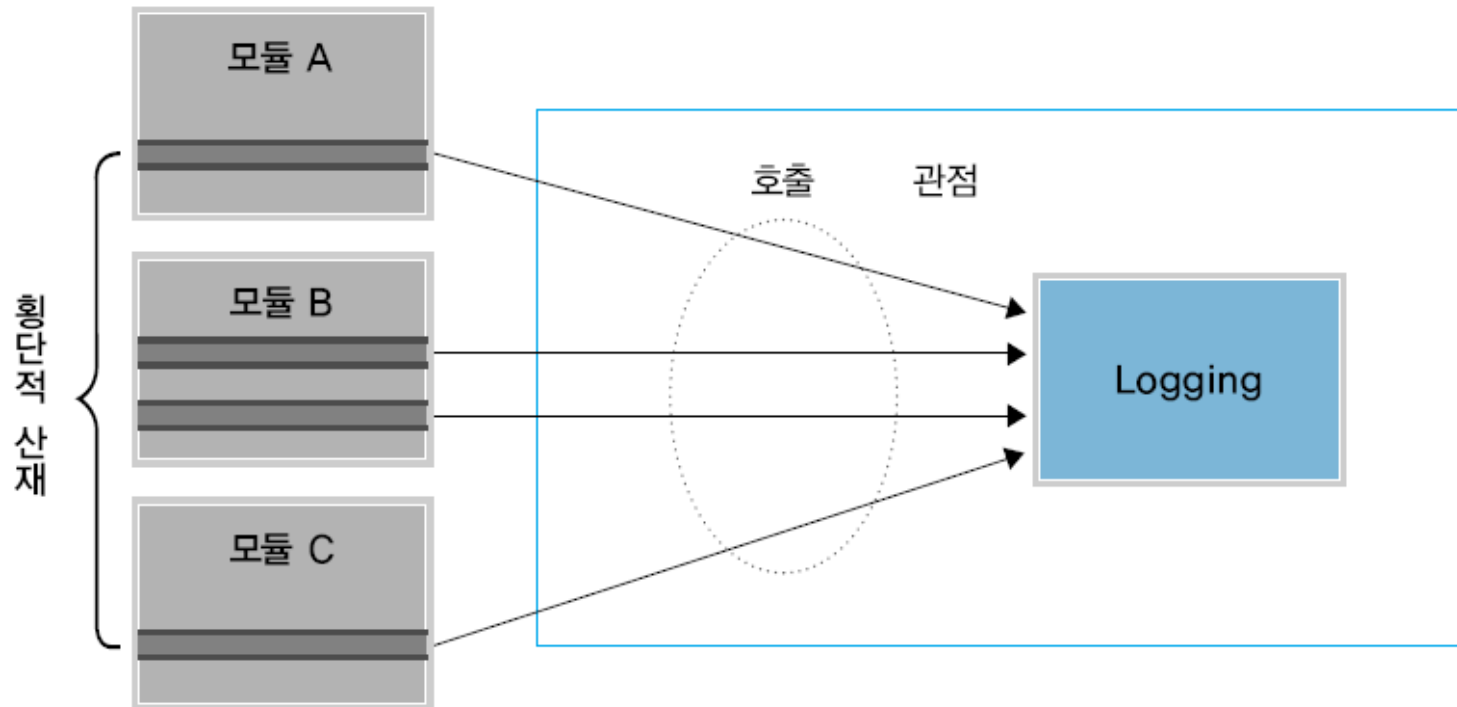
- 오브젝트 지향에서는 이들 업무들을 하나의 클래스라는 단위로 모으고 그것들을 모듈로부터 분리함으로써 재사용성과 보수성을 높이고 있다.



▲ 기존의 객체 지향에서 관점의 분리

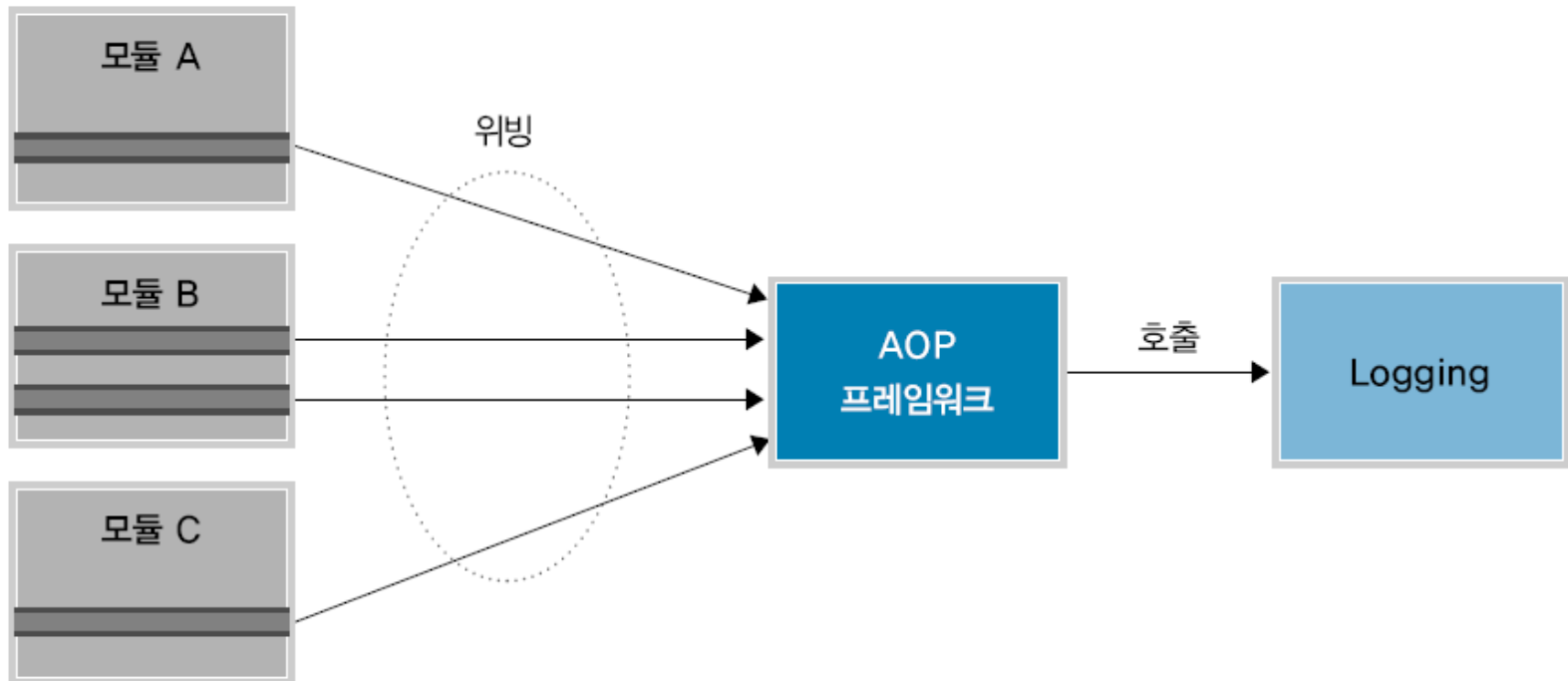
AOP

- 오브젝트 지향에서는 로깅이라는 기능 및 관련하는 데이터 자체는 각 모듈로부터 분리하는 것으로 성공했지만 **그 기능을 사용하기 위해서 코드까지는 각 모듈로부터 분리할 수 없다.** 그렇기 때문에 분리한 기능을 이용하기 위해서 코드가 각 모듈에 횡단으로 산재하게 된다.



AOP

- AOP에서는 분리한 기능의 호출도 포함하여 「관점」으로 다룬다.
그리고 이러한 각 모듈로 산재한 관점을 「횡단 관점」라 부르고 있다.
- AOP에서는 이러한 「횡단 관점」까지 분리함으로써 **각 모듈로부터 관점에 관한 코드를 완전히 제거하는 것을 목표로 한다.**



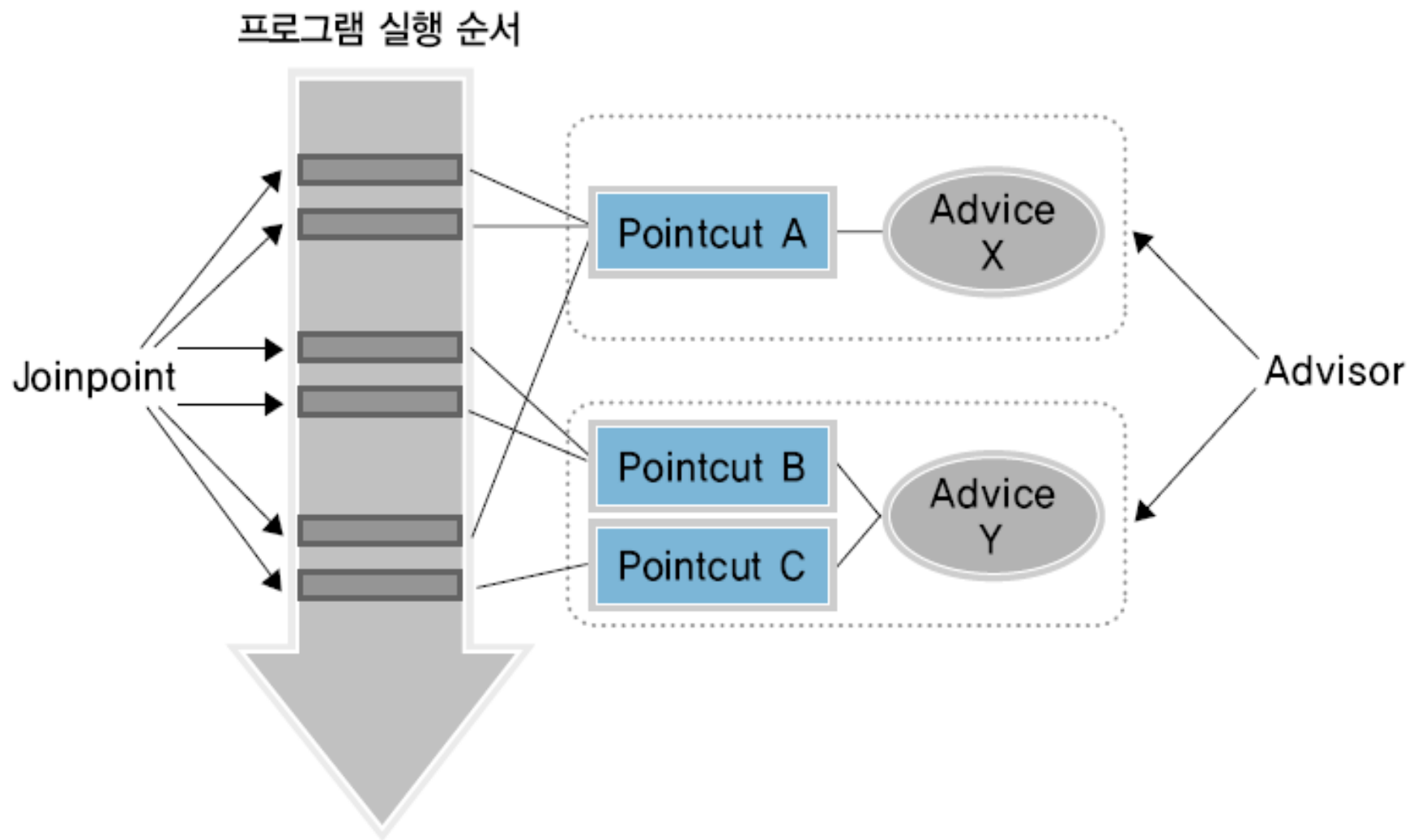
▲ AOP의 횡단 관점의 분리와 위빙

스프링 AOP 용어

- **Joinpoint** - 「클래스의 인스턴스 생성 시점」, 「메소드 호출 시점」 및 「예외 발생 시점」과 같이 애플리케이션을 실행할 때 특정 작업이 시작되는 시점을 Joinpoint라 한다. 즉, 내가 원하는 위치에 특정 공통모듈을 삽입하기 위해 시작점을 알아야 하는데 그러한 시작 기준점을 뜻함.
- **Pointcut** - 여러 개의 Joinpoint를 하나로 결합한(묶은) 것을 Pointcut이라고 부른다.
- **Advice** - Joinpoint에 삽입되어져 동작할 수 있는 코드를 Advice라 한다.(실제적인 코딩부분)

스프링 AOP 용어

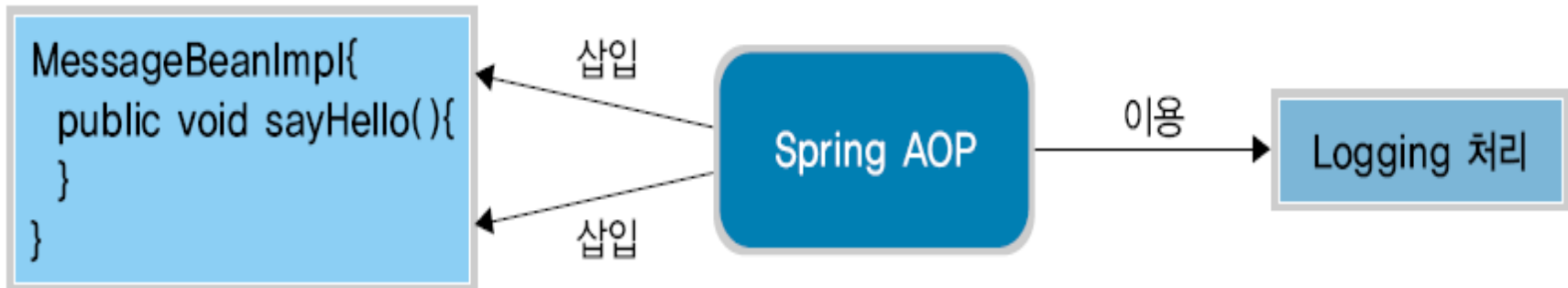
- **Advisor** - Advice와 Pointcut를 하나로 묶어 취급한 것을 Advisor라 부른다.
- **Weaving** - Advice를 핵심 로직 코드에 삽입하는 것을 Weaving이라 부른다.(AOP프레임웍에서 직접 호출해서 전달 할 수 있도록하는 것을 위빙이라 함.)
- **Target** - 핵심 로직을 구현하는 클래스를 말한다.
- **Aspect** - 여러 객체에 공통으로 적용되는 공통 관점 사항을 Aspect라 부른다.



▲ 스프링 AOP에서의 용어와 개념

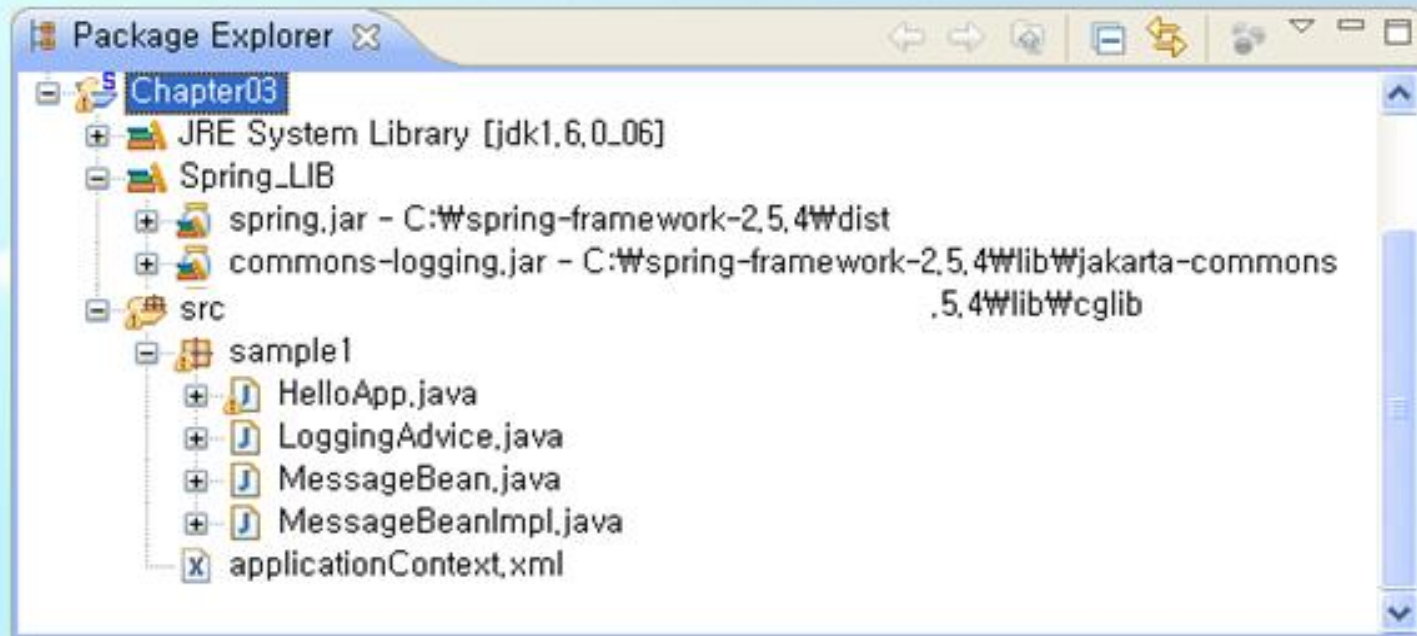
AOP를 이용한 logging 구현 예제

- 이 예제에서는 AOP 구조를 활용하여 메소드 트레이스 정보의 Logging 처리를 MessageBeanImpl의 sayHello() 메소드 호출 전 후에 삽입한다.
- 로깅 처리 자체 및 그 호출은 MessageBeanImpl에는 기술하지 않는다.
- 스프링이 제공하는 기능인 「스프링 AOP」가 그 역할을 담당한다.



▲ 예제 개요 그림

▼ 파일 구성



sayHello() 메소드가 핵심 로직이고 이 메소드를 멤버로 갖는 MessageBeanImpl는 타겟 클래스가 된다.
LoggingAdvice 클래스가 로깅처리를 담당하고 있게 된다.

스프링에서 AOP를 구현하는 방법

1. 각 Advice 타입에 대응하는 인터페이스 구현
2. AspectJ 스타일 AOP의 이용
3. 어노테이션으로 AOP 설정

각 Advice 타입에 대응하는 인터페이스 구현

1. Advice 클래스를 작성한다. – 특정 인터페이스구현
2. 설정 파일에 Pointcut을 설정한다.
3. 설정 파일에 Advice와 Pointcut을 묶어 놓는 Advisor를 설정한다.
4. 설정 파일에 ProxyFactoryBean 클래스를 이용하여 대상 객체에 Advisor를 적용한다.
5. `getBean()` 메소드로 빈 객체를 가져와 사용한다.

각 Advice 타입에 대응하는 인터페이스 구현

Advice 클래스를 작성한다

(JoinPoint 앞뒤에서 실행되는 Advice)

```
public class LoggingAdvice implements MethodInterceptor {  
    public Object invoke(MethodInvocation invocation) throws Throwable {  
        String methodName = invocation.getMethod().getName();  
        Stopwatch sw = new Stopwatch();  
  
        sw.start(methodName);  
  
        System.out.println("[LOG] METHOD: " + methodName + " is calling.");  
        Object rtnObj = invocation.proceed();  
  
        sw.stop();  
  
        System.out.println("[LOG] METHOD: " + methodName + " was called.");  
        System.out.println("[LOG] 처리시간 " + sw.getTotalTimeMillis() / 1000 + "초");  
  
        return rtnObj;  
    }  
}
```


각 Advice 타입에 대응하는 인터페이스 구현

설정 파일 작성하기

```
<bean id="loggingAdvice" class="sample1.LoggingAdvice" />
```

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="targetBean"/>
  <property name="interceptorNames">
    <list>
      <value>advisor</value>
    </list>
  </property>
</bean>
```

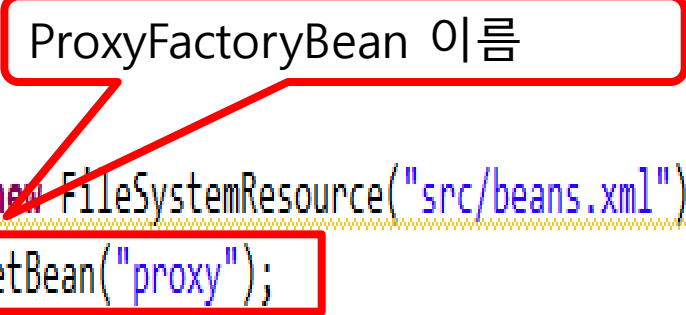
Advice와 Pointcut를 하나로 묶어 취급한 것

```
<bean id="advisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
  <property name="advice" ref="loggingAdvice"/>
  <property name="pointcut">
    <bean class="org.springframework.aop.support.JdkRegexMethodPointcut">
      <property name="pattern">
        <value>.*sayHello.*</value>
      </property>
    </bean>
  </property>
</bean>
```

각 Advice 타입에 대응하는 인터페이스 구현

getBean() 메소드로 빈 객체를 가져와 사용
한다

```
public class HelloApp {  
    public static void main(String[] args) {  
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource("src/beans.xml"));  
        MessageBean bean = (MessageBean)factory.getBean("proxy");  
  
        /*ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");  
        MessageBean bean = (MessageBean)context.getBean("proxy");*/  
  
        bean.sayHello();  
    }  
}
```



AspectJ 스타일 AOP의 이용

1. AspectJ는 제록스 팔로알토 연구소에서 개발했던 AOP 구현으로 현재 eclipse.org 개발프로젝트에서 관리하고 있음.
2. 특정 클래스나 인터페이스를 상속 /구현 하지 않고 POJO방식코딩
3. Xml 설정파일에서 AspectJ스타일로 Aop설정한다.

AspectJ 스타일 AOP의 이용

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

```
<bean id="loggingSample" class="sample1.LoggingSample" />
```

Aspect설정: Advice를 어떤
Pointcut에 적용할지 설정

```
<aop:config>
  <aop:aspect id="logAspect" ref="loggingSample">
    <aop:pointcut expression="execution(* sayHello())" id="logPointCut"/>
    <aop:around pointcut-ref="logPointCut" method="logAround"/> <!--LoggingSample의 logAround메소드 호출 -->
  </aop:aspect>
</aop:config>
```

AspectJ ~~스프링~~ AOP의 이용

```
public class LoggingSample {  
    public Object logAround(ProceedingJoinPoint pjp) throws Throwable {  
        String methodName = pjp.getKind();  
        Stopwatch sw = new Stopwatch();  
  
        sw.start(methodName);  
  
        System.out.println("[LOG] METHOD: " + methodName + " is calling.");  
        Object rtnObj = pjp.proceed();  
  
        sw.stop();  
  
        System.out.println("[LOG] METHOD: " + methodName + " was called.");  
        System.out.println("[LOG] 처리시간 " + sw.getTotalTimeMillis() / 1000 + "초");  
  
        return rtnObj;  
    }  
}
```

}

AspectJ 스타일 AOP의 이용

```
public class HelloApp {  
    public static void main(String[] args) {  
        ApplicationContext factory = new FileSystemXmlApplicationContext("src/beans.xml");  
        MessageBean bean = (MessageBean)factory.getBean("targetBean");  
  
        bean.sayHello();  
    }  
}
```

어노테이션으로 AOP 설정

- @AspectJ어노테이션을 Advice로 사용하는 클래스에 기술한다.
- Advice처리를 맡는 메서드에 @Around어노테이션을 기술한다.
- @Around어노테이션 값에 Pointcut정의를 작성한다.

- 어노테이션을 사용해 AOP를 설정하는 경우 설정파일에 <aop:aspectj-autoproxy/> 반드시 기술한다.
- <aop:config>요소 내용을 어노테이션으로 설정하므로 <aop:config>요소는 삭제한다.

어노테이션으로 AOP 설정

```
@Aspect
public class LoggingSample {
    @Around("execution(* sayHello())")
    public Object logAround(ProceedingJoinPoint pjp) throws Throwable {
        String methodName = pjp.getKind();
        Stopwatch sw = new Stopwatch();

        sw.start(methodName);

        System.out.println("[LOG] METHOD: " + methodName + " is 호출중입니다..");
        Object rtnObj = pjp.proceed();

        sw.stop();

        System.out.println("[LOG] METHOD: " + methodName + " 완료되었습니다..");
        System.out.println("[LOG] 처리시간 " + sw.getTotalTimeMillis() / 1000 + "초");

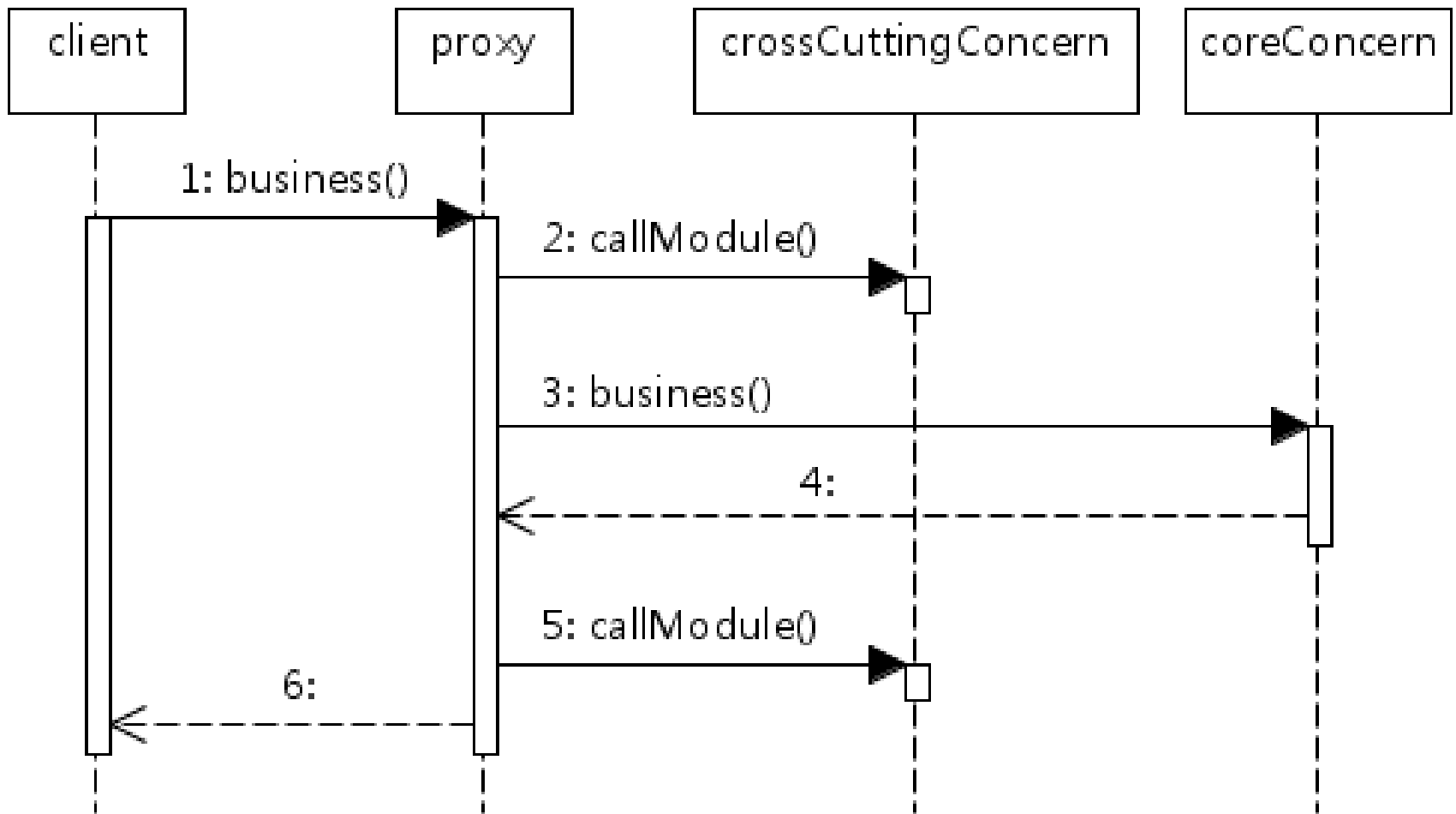
        return rtnObj;
    }
}
```


어노테이션으로 AOP 설정

```
3 <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xmlns:aop="http://www.springframework.org/schema/aop"
6     xsi:schemaLocation="http://www.springframework.org/schema/beans
7     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8     http://www.springframework.org/schema/aop
9     http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
10
11     <bean id="loggingSample" class="sample1.LoggingSample" />
12
13     <aop:aspectj-autoproxy/>
14
15     <bean id="targetBean" class="sample1.MessageBeanImpl">
16         <property name="name">
17             <value>Spring</value>
18         </property>
19     </bean>
20 </beans>
```

세 가지 Weaving 방식

- 컴파일 시에 Weaving
 - AspectJ에서 사용하는 방식
- 클래스로딩 시에 Weaving
 - AspectJ 5/6 버전이 컴파일 방식과 더불어 제공
- 런타임 시에 Weaving
 - 프록시를 이용하여 AOP를 적용한다.
 - 핵심 객체에 직접 접근하지 않고 중간에 프록시를 통하여 핵심 로직을 구현한 객체에 접근하게 된다.



** 메서드가 호출될 때에만 Advice를 적용할 수 있기 때문에 필드 값 변경과 같은 Joinpoint에 대해서는 적용할 수 없다.

스프링에서의 AOP

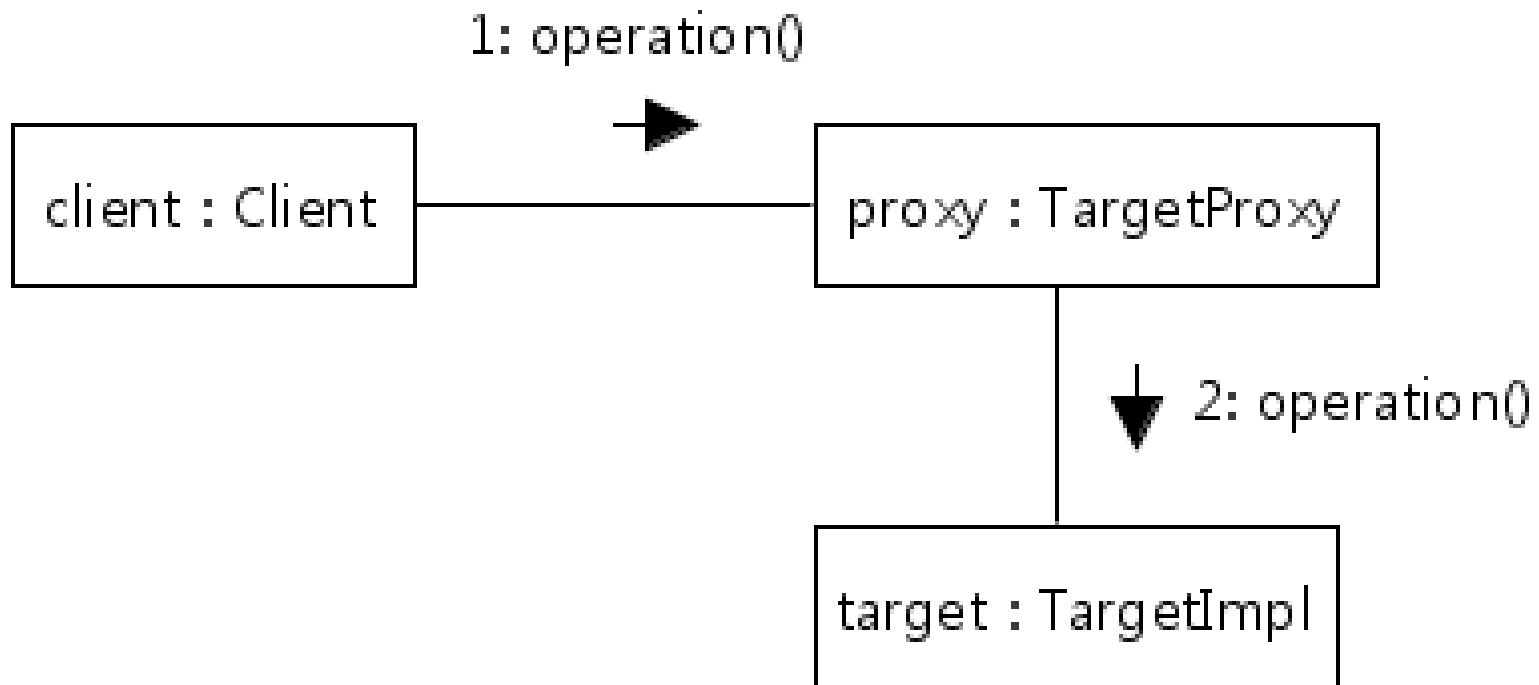
- 스프링 AOP는 메서드 호출 Joinpoint만을 지원한다.
- 필드 값 변경과 같은 Joinpoint를 사용 하려면 AspectJ 같은 풍부한 기능은 지원하는 AOP 도구를 사용해야 한다.

스프링 AOP 구현 방식

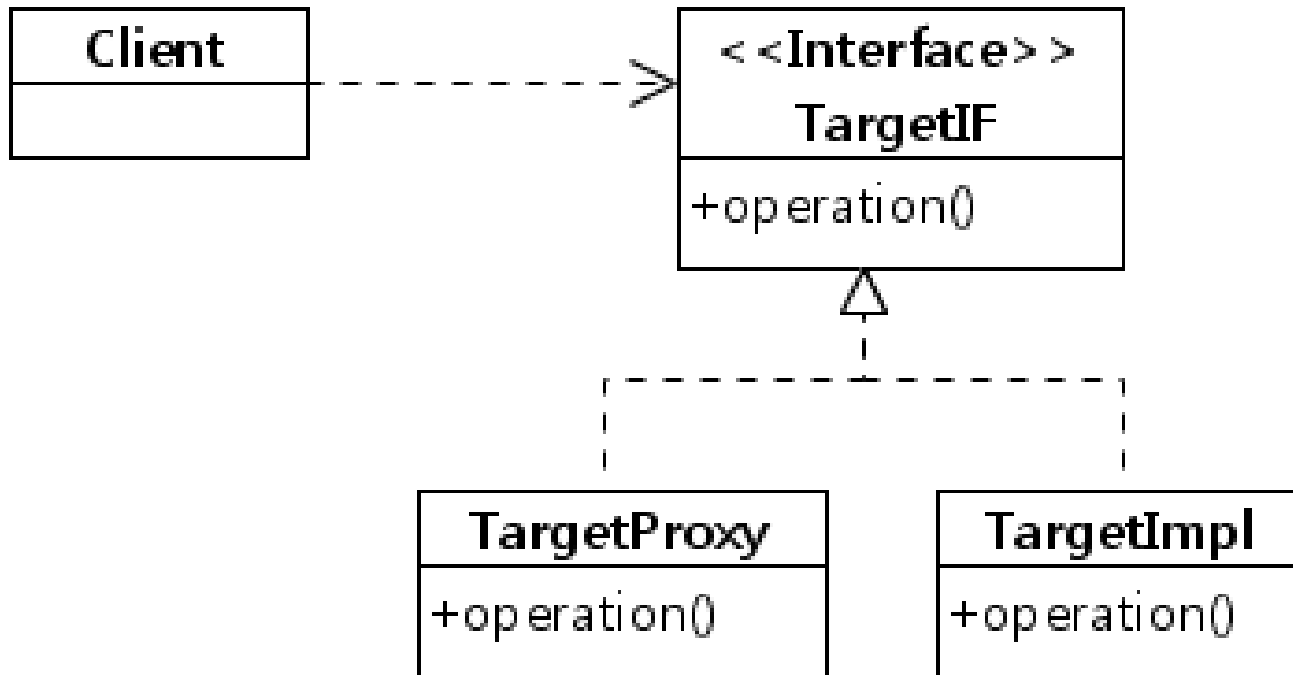
- XML 스키마 기반의 POJO 클래스를 이용한 AOP 구현
- AspectJ 5/6에서 정의한 @AspectJ 어노테이션 기반의 AOP 구현
- 스프링 API를 이용한 AOP 구현

** 어떤 방식을 사용하더라도 내부적으로는 프록시를 이용하여 AOP가 구현되므로 메서드 호출에 대해서만 AOP를 적용할 수 있다.

프록시를 이용한 AOP 구현



스프링은 Aspect의 적용대상이 되는 객체에 대한 프록시를 만들어 제공하며, client는 프록시를 통하여 간접적으로 대상객체에 접근하게 된다.



- ** 대상객체가 인터페이스를 구현
 - > `java.lang.reflect.Proxy`를 이용하여 프록시 객체 생성
- ** 대상객체가 인터페이스를 구현하고 있지 않을때
 - > CGLIB을 이용하여 프록시 객체를 생성
 - > 대상객체 및 메서드가 `final`이 될 수 없다.

구현 가능한 Advice의 종류

종류	설명
Before Advice	대상 객체의 메서드 호출 전에 공통 기능을 실행한다.
After Returning Advice	대상 객체의 메서드가 예외 없이 실행한 이후에 공통 기능을 실행한다.
After Throwing Advice	대상 객체의 메서드를 실행하는 도중 예외가 발생한 경우에 공통기능을 실행한다.
After Advice	대상 객체의 메서드를 실행하는 도중에 예외가 발생했는지의 여부와 상관없이 메서드 실행 후 공통 기능을 실행한다.
Around Advice	대상 객체의 메서드 실행 전, 후 또는 예외 발생 시점에 공통 기능을 실행한다.

**** 대상 객체의 메서드의 실행하기 전/후에 원하는 기능을 삽입 할 수 있기 때문에 Around Advice를 범용적으로 사용함.**

XML 스키마 기반의 POJO 클래스를 이용한 AOP 구현

- 스프링 2 버전 부터 스프링 API를 사용하지 않은 POJO 클래스를 이용하여 Advice를 적용하는 방법이 추가됨.

XML 스키마 이용 AOP 구현 과정

- 관련 jar를 클래스 패스에 추가한다.
- 공통기능을 제공하는 Advice 클래스를 구현한다.
- XML 설정파일에서 <aop:config>를 이용하여 Aspect를 설정한다.
- Advice를 어떤 Pointcut에 적용할지를 지정하게 된다.

공통 기능을 제공할 Advice클래스 작성

```
public class ProfilingAdvice {  
  
    public Object trace(ProceedingJoinPoint joinPoint) throws Throwable {  
        String signatureString = joinPoint.getSignature().toShortString();  
        System.out.println(signatureString + " 시작");  
        long start = System.currentTimeMillis();  
        try {  
            Object result = joinPoint.proceed();  
            return result;  
        } finally {  
            long finish = System.currentTimeMillis();  
            System.out.println(signatureString + " 종료");  
            System.out.println(signatureString + " 실행 시간 : " + (finish - start)  
                               + "ms");  
        }  
    }  
}
```

- ** 특정 Joinpoint에서 실행될 trace()메서드를구현
- ** 매개변수로 전달받은 ProceedingJoinPoint를 통해 AroundAdvice를 구현 할 수 있음.

Advice 객체의 생성

```
<!-- Advice 클래스를 빈으로 등록 -->  
<bean id="performanceTraceAdvice"  
      class="madvirus.spring.chap05.aop.pojo.ProfilingAdvice" />
```

Aop의 설정

```
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

<!-- Aspect 설정: Advice를 어떤 Pointcut에 적용할 지 설정 -->
<aop:config>
    <aop:aspect id="traceAspect1" ref="performanceTraceAdvice">
        <aop:pointcut id="publicMethod"
            expression="execution(public * madvirus.spring.chap05.board..*(..))" />
        <aop:around pointcut-ref="publicMethod" method="trace" />
    </aop:aspect>
</aop:config>
```

madvirus.spring.chap05 패키지의 모든 public 메서드를 pointcut으로 설정

Target이 되는 객체의 생성

```
<bean id="writeArticleService"
      class="madvirus.spring.chap05.board.service.WriteArticleServiceImpl">
  <constructor-arg>
    <ref bean="articleDao" />
  </constructor-arg>
</bean>

<bean id="articleDao"
      class="madvirus.spring.chap05.board.dao.MySQLArticleDao" />

<bean id="memberService"
      class="madvirus.spring.chap05.member.service.MemberServiceImpl" />
```

```
public class MainQuickStart {

    public static void main(String[] args) {
        String[] configLocations = new String[] { "acQuickStart.xml" };
        ApplicationContext context = new ClassPathXmlApplicationContext(
            configLocations);

        WriteArticleService articleService = (WriteArticleService) context
            .getBean("writeArticleService");
        articleService.write(new Article());

        MemberService memberService = context.getBean("memberService",
            MemberService.class);
        memberService.regist(new Member());
    }
}
```

AOP 관련 정보 설정

<aop:config>	AOP 설정 정보임을 나타냄
<aop:aspect>	Aspect를 설정
<aop:pointcut>	Pointcut을 설정
<aop:around>	Around Advice를 설정

Advice 정의 관련 태그

태그	설명
<aop:before>	메서드 실행 전에 적용되는 Advice
<aop:after-returning>	메서드가 정상적으로 실행 된 후에 적용되는 Advice
<aop:after-throwing>	메서드가 예외를 발생 시킬때 적용되는 Advice
<aop:after>	메서드가 정상적으로 실행되는지 예외를 발생시키는지 여부에 상관없이 적용되는 Advice
<aop:around>	메서드 호출 이전, 이후, 예외발생 등 모든 시점에 적용가능한 Advice

<aop:around>에 pointcut을 직접 설정

```
<aop:config>
  <aop:aspect id="traceAspect1" ref="performanceTraceAdvice">
    <aop:pointcut id="publicMethod"
      expression="execution(public * madvirus.spring.chap05.board..*(..))" />
    <aop:around pointcut-ref="publicMethod" method="trace" />
  </aop:aspect>

  <aop:aspect id="traceAspect2" ref="performanceTraceAdvice">
    <aop:around pointcut="execution(public * madvirus.spring.chap05.member..*(..))"
      method="trace" />
  </aop:aspect>
</aop:config>
```

Advice 타입 별 클래스 작성

```
<aop:aspect id="traceAspect2" ref="performanceTraceAdvice">  
  <aop:around pointcut="execution(public * madvirus.spring.chap05.member..*(..))"  
    method="trace" />  
</aop:aspect>
```

AroundAdvice가 제공

Before Advice

```
<aop:config>
  <aop:aspect id="loggingAspect" ref="loggingAdvice">
    <aop:pointcut id="publicMethod"
      expression="execution(public * madvirus.spr
    <aop:before method="before" pointcut-ref="publicMethod">
    <aop:after-returning method="afterReturning" pointcut-ref="publicMethod">
    <aop:after-throwing method="afterThrowing" pointcut-ref="publicMethod">
    <aop:after method="afterFinally" pointcut-ref="publicMethod">
  </aop:aspect>
```

```
public void before()
{
    // 대상 객체의 메서드 실행 이전에 적용할 기능 구현
}
```

```
public void before(JoinPoint joinPoint)
// 대상 객체 및 호출되는 메서드에 대한 정보가 필요할 경우에 사용
{
    ..
}
```

** Before Advice에서 예외를 발생시키면 대상 객체의 메서드가 호출되지 않기 때문에 메서드를 실행하기 전에 접근 권한을 검사해서 권한이 없을 경우 예외를 발생 시키도록 하는 것이 적합하다.

After Returning Advice

- 대상 객체의 메서드가 정상적으로 실행 된 후 공통 기능을 적용하고자 할 때 사용

```
<bean id="logging" class="madvirus.spring.chap05.aop.pojo.LoggingAdvice" />
<aop:config>
    <aop:aspect id="loggingAspect" ref="logging">
        <aop:pointcut
            expression="execution(public * madvirus.spring.chap05.board..*(..))"
            id="publicMethod"/>
        <aop:after-returning method="afterReturning"
            pointcut-ref="publicMethod" />
    </aop:aspect>
</aop:config>
```

```
public void afterReturning() {
    System.out.println("[LA] 메서드 실행 후 후처리 수행");
}
```

리턴 값을 사용하고자 할때

```
<aop:after-returning method="afterReturning"  
    pointcut-ref="publicMethod" returning="ret"/>
```

전달받을 파라미터 이름을 명시

```
public void afterReturning(Object ret) {  
    // returnning 속성에 명시한 이름을 갖는 파라미터를 이용해서  
    // 값을 전달 받는다.  
}  
  
public void afterReturning(Article ret) {  
    //리턴 객체의 특정 타입인 경우에 한해서 처리하고자 할때  
}  
  
public void afterReturning(JoinPoint joinPoint, Article ret) {  
    //대상객체 및 호출되는 메서드에 대한 정보나 전달되는 파라미터에  
    //에 대한 정보가 필요할때  
}
```

After Throwing Advice

- 대상 객체의 메서드가 예외를 발생시킨 경우에 적용
- <aop:after-throwing>태그를 이용

```
<bean id="logging" class="madvirus.spring.chap05.aop.pojo.LoggingAdvice" />
<aop:config>
    <aop:aspect id="loggingAspect" ref="logging">
        <aop:pointcut
            expression="execution(public * madvirus.spring.chap05.board..*(..))"
            id="publicMethod"/>
        <aop:after-throwing method="afterThrowing"
            pointcut-ref="publicMethod"/>
    </aop:aspect>
</aop:config>
```

```
public void afterThrowing() {
    System.out.println("[LA] 메서드 실행 중 예외 발생");
}
```

대상 객체의 메서드가 발생시킨 예외 객체가 필요한 경우

```
<aop:after-throwing method="afterThrowing"  
    pointcut-ref="publicMethod" throwing="ex"/>
```

예외를 받을 파라미터 이름을 명시한다.

```
public void afterThrowing(Throwable ex) {  
    System.out.println("[LA] 메서드 실행 중 예외 발생, 예외=" + ex.getMessage());  
}
```

```
public void afterThrowing(ArticleNotFoundException ex) {  
  
}
```

```
public void afterThrowing(JoinPoint joinPoint, Exception ex) {  
  
}
```


After Advice

- 대상 객체의 메서드가 정상적으로 실행 되었는지 예외를 발생 시켰는지의 여부에 상관없이 적용

```
<aop:config>
  <aop:aspect id="loggingAspect" ref="logging">
    <aop:pointcut
      expression="execution(public * madvirus.spring.chap05.board..*(..))"
      id="publicMethod"/>
    <aop:after method="afterFinally"
      pointcut-ref="publicMethod" />
  </aop:aspect>
</aop:config>
```

```
public void afterFinally() {
```

```
}
```

```
public void afterFinally(JoinPoint joinPoint) {
```

```
}
```

Around Advice

- Before, After, Returning, After Throwing, After Advice를 모두 구현할 수 있음.
- <aop:around> 태그를 이용하여 설정

```
<bean id="cache" class="madvirus.spring.chap05.aop.pojo.ArticleCacheAdvice" />
<aop:config>
    <aop:aspect id="cacheAspect" ref="cache">
        <aop:around method="cache"
            pointcut="execution(public * *..ReadArticleService.*(..))"/>
        </aop:aspect>
    </aop:config>
```

Around Advice의 사용 예

```
package madvirus.spring.chap05.aop.pojo;

import java.util.HashMap;

public class ArticleCacheAdvice {

    private Map<Integer, Article> cache = new HashMap<Integer, Article>();

    public Article cache(ProceedingJoinPoint joinPoint) throws Throwable {
        Integer id = (Integer) joinPoint.getArgs()[0];
        Article article = cache.get(id);
        if (article != null) {
            System.out.println("[ACA] 캐시에서 Article[" + id + "] 구함");
            return article;
        }
        Article ret = (Article) joinPoint.proceed();
        if (ret != null) {
            cache.put(id, ret);
            System.out.println("[ACA] 캐시에 Article[" + id + "] 추가함");
        }
        return ret;
    }
}
```

@Aspect 어노테이션을 이용한 AOP

- AspectJ 5 버전에 추가된 어노테이션
- Xml 파일에 Advice 및 Pointcut 설정을 하지 않고 자동으로 Advice를 적용
- 스프링 2 버전 부터 @Aspect 어노테이션을 지원

XML 스키마 기반의 AOP와 차이점

- `@Aspect` 어노테이션을 이용해서 Aspect 클래스를 구현한다.
- Aspect클래스는 Advice를 구현한 메서드와 Pointcut을 포함한다.
- XML 설정에서 `<aop:aspectj-autoproxy/>`를 설정한다.

```

@Aspect
public class ProfilingAspect {

    @Pointcut("execution(public * madvirus.spring.chap05.board..*(..))")
    private void profileTarget() {}

    @Around("profileTarget()")
    public Object trace(ProceedingJoinPoint joinPoint) throws Throwable {
        String signatureString = joinPoint.getSignature().toShortString();
        System.out.println(signatureString + " 시작");
        long start = System.currentTimeMillis();
        try {
            Object result = joinPoint.proceed();
            return result;
        } finally {
            long finish = System.currentTimeMillis();
            System.out.println(signatureString + " 종료");
            System.out.println(signatureString + " 실행 시간 : " + (finish - start)
                               + "ms");
        }
    }
}

```

@Pointcut이 적용된 메소드의 리턴값은 void 여야 한다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:aop="http://www.springframework.org/schema/aop"
  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-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

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

```
<!-- Aspect 클래스를 빈으로 등록 -->
```

```
<bean id="performanceTraceAspect"
  class="madvirus.spring.chap05.aop.annot.ProfilingAspect" />
```

```
<bean id="writeArticleService"
  class="madvirus.spring.chap05.board.service.WriteArticleServiceImpl">
  <constructor-arg>
    <ref bean="articleDao" />
  </constructor-arg>
</bean>
```

```
<bean id="articleDao"
  class="madvirus.spring.chap05.board.dao.MySQLArticleDao" />
```

```
<bean id="memberService"
  class="madvirus.spring.chap05.member.service.MemberServiceImpl" />
```

```
</beans>
```

```

public class MainQuickStart2 {

    public static void main(String[] args) {
        String[] configLocations = new String[] { "acQuickStart2.xml" };
        ApplicationContext context = new ClassPathXmlApplicationContext(
            configLocations);

        WriteArticleService articleService = (WriteArticleService) context
            .getBean("writeArticleService");
        articleService.write(new Article());

        MemberService memberService = context.getBean("memberService",
            MemberService.class);
        memberService.regist(new Member());
    }
}

```


Advice 타입 별 클래스 작성

- Before Advice
- After Returning Advice
- After Throwing Advice
- After Advice
- Around Advice

Before Advice

- @Before 어노테이션을 사용한다.

```
@Aspect
public class LoggingAspect {

    @Before("execution(public * madvirus.spring.chap05..*(..))")
    public void before() {
        System.out.println("[LA] 메서드 실행 전 전처리 수행");
    }
}
```

@Before 어노테이션 값으로는 AspectJ의 Pointcut 표현식이나 @PointCut 어노테이션이 적용된 메서드이름이 올 수 있다.

Before Advice 구현 메서드는 madvirus.spring.chap05 패키지 또는 그 하위에 있는 모든 public 메서드가 호출되기 전에 호출된다.

After Returning Advice

- @AfterReturning 어노테이션을 구현 메서드에 적용

```
@Aspect
public class LoggingAspect {

    @AfterReturning("madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()")
    public void afterReturning() {
        System.out.println("[LA] 메서드 실행 후 후처리 수행");
    }
}
```

대상 객체가 리턴 한 값을 사용 하고자 할 때

```
@Pointcut("execution(public * exam.*(..))")  
private void loggingTarget() {}
```

```
@AfterReturning(pointcut="loggingTarget()",  
                returning="ret")  
public void afterReturning(JoinPoint joinPoint, Object ret  
{  
    //System.out.println( joinPoint.getSignature().getName()  
    System.out.println(joinPoint.getSignature().getName() + "  
}
```

대상객체의 반환 값이 특정 타입인 경우에 한해서 메서드를 실행

```
@Aspect
public class LoggingAspect {

    @AfterReturning(
        pointcut="madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()",
        returning="ret")
    public void afterReturning(Article ret) {
        System.out.println("[LA] 메서드 실행 후 후처리 수행, 리턴값="+ret);
    }
}
```

대상 객체 및 호출되는 메서드의 정보가 필요한 경우

```
@Aspect
public class LoggingAspect {

    @AfterReturning(
        pointcut="madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()",
        returning="ret")
    public void afterReturning(JoinPoint joinPoint, | Article ret) {
        System.out.println("[LA] 메서드 실행 후 후처리 수행, 리턴값="+ret);
    }
}
```

After Throwing Advice

- @AfterThrowing 어노테이션 사용

```
@Aspect
public class LoggingAspect {
    @AfterThrowing("madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()")
    public void afterThrowing() {
        System.out.println("[LA] 메서드 실행 중 예외 발생");
    }
}
```

대상객체의 메서드가 발생시킨 예외에 접근

```
@Aspect
public class LoggingAspect {
    @AfterThrowing(
        pointcut="madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()",
        throwing="ex")
    public void afterThrowing(Throwable ex) {
        System.out.println("[LA] 메서드 실행 중 예외 발생, 예외 =" + ex.getMessage());
    }
}
```


특정 타입의 예외에 대해서만 처리

```
@Aspect
public class LoggingAspect {
    @AfterThrowing(
        pointcut="madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()",
        throwing="ex")
    public void afterThrowing(FileNotFoundException ex) {
        System.out.println("[LA] 메서드 실행 중 예외 발생, 예외 =" + ex.getMessage());
    }
}
```

대상 객체 및 호출 메서드에 대한 정보가 필요한 경우

```
@Aspect
public class LoggingAspect {
    @AfterThrowing(
        pointcut="madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()",
        throwing="ex")
    public void afterThrowing(JoinPoint joinPoint, Exception ex) {
        System.out.println("[LA] 메서드 실행 중 예외 발생, 예외 =" + ex.getMessage());
    }
}
```

After Advice

- @After 어노테이션 사용

```
@Aspect
public class LoggingAspect {

    @After("madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod() ")
    public void afterFinally() {
        System.out.println("[LA] 메서드 실행 완료");
    }
}
```

```
@Aspect
public class LoggingAspect {

    @After("madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod() ")
    public void afterFinally(JoinPoint joinPoint) {
        System.out.println("[LA] 메서드 실행 완료");
    }
}
```

Around Advice의 사용 예

```
@Aspect
public class ArticleCacheAspect {

    private Map<Integer, Article> cache = new HashMap<Integer, Article>();

    @Around("execution(public * *..ReadArticleService.*(..))")
    public Article cache(ProceedingJoinPoint joinPoint) throws Throwable {
        Integer id = (Integer) joinPoint.getArgs()[0];
        Article article = cache.get(id);
        if (article != null) {
            System.out.println("[ACA] 캐시에서 Article[" + id + "] 구함");
            return article;
        }
        Article ret = (Article) joinPoint.proceed();
        if (ret != null) {
            cache.put(id, ret);
            System.out.println("[ACA] 캐시에 Article[" + id + "] 추가함");
        }
        return ret;
    }
}
```

@Pointcut 어노테이션을 이용한 Pointcut설정

```
<aop:config>
  <aop:aspect id="traceAspect1" ref="performanceTraceAdvice">
    <aop:pointcut id="publicMethod"
      expression="execution(public * madvirus.spring.chap05.board..*(..))" />
    <aop:around pointcut-ref="publicMethod" method="trace" />
  </aop:aspect>
</aop:config>
```

```
@Aspect
public class ProfilingAspect {

  @Pointcut("execution(public * madvirus.spring.chap05.board..*(..))")
  private void profileTarget() {}

  @Around("profileTarget()")
  public Object trace(ProceedingJoinPoint joinPoint) throws Throwable {

  }
}
```

Pointcut 어노테이션이 적용된 메소드는 리턴값이 void 여야 하고, 메소드 몸체에 코드를 갖지 않으며, 코드를 가져도 의미가 없다.

@Pointcut의 참조

- 같은 클래스에서
 - 메서드 이름만 입력
- 같은 패키지에서
 - 클래스이름.메서드이름
- 다른 패키지에서
 - 완전한클래스이름.메서드이름

```
package madvirus.spring.chap05.aop.annot;

import org.aspectj.lang.annotation.Pointcut;

public class PublicPointcut {

    @Pointcut("execution(public * madvirus.spring.chap05..*(..))")
    public void publicMethod() { }
}
```

@Aspect

```
public class LoggingAspect {

    @Before("PublicPointcut.publicMethod()")
    public void before() {
        System.out.println("[LA] 메서드 실행 전 전처리 수행");
    }

    @AfterReturning(
        pointcut = "madvirus.spring.chap05.aop.annot.PublicPointcut.publicMethod()",
        returning = "ret")
    public void afterReturning(Object ret) {
        System.out.println("[LA] 메서드 실행 후 후처리 수행, 리턴값=" + ret);
    }
}
```

JoinPoint 사용

- Around Advice를 제외한 나머지 Advice 타입을 구현한 메서드는 JoinPoint 객체를 선택적으로 전달 받을 수 있다.

```
public void afterLogging(Object refVal, JoinPoint joinPoint) {  
}
```

JoinPoint를 첫 번째 매개변수로 받아야 한다. 그렇지 않으면 예외를 발생시킨다

JoinPoint의 메소드

- Signature getSignature()
 - 호출되는 메서드의 정보
- Object getTarget()
 - 대상 객체를 반환
- Object[] getArgs()
 - 파라미터 목록을 반환

Signature의 메서드

- `String getName()`
 - 메서드 이름을 반환
- `String toLongString()`
 - 메서드의 리턴타입, 파라미터 타입의 정보를 반환
- `String toShortString()`
 - 메서의 이름을 반환

타입을 이용한 파라미터 접근

- JoinPoint를 사용하지 않고 Advice 메서드에서 직접 파라미터 이용해서 메서드 호출시 사용된 인자에 접근할 수 있다.
1. Advice 구현 메서드에 인자를 전달 받을 파라미터를 명시한다.
 2. Pointcut 표현식에서 args() 명시자를 사용해서 인자 목록을 지정한다.

Advice 구현 메서드에 인자를 전달 받을 파라미터를 명시한다.

```
public class UpdateMemberInfoTraceAspect {  
  
    public void traceReturn(String memberId, UpdateInfo info) {  
        System.out.println("[TA] 정보 수정: 대상회원-" + memberId  
            + ", 수정정보=" + info);  
    }  
  
}
```

Pointcut 표현식에서 args() 명시자를 사용해서 인자 목록을 지정한다.

```
<bean id="traceAdvice"
      class="madvirus.spring.chap05.aop.pojo.UpdateMemberInfoTraceAdvice"/>

<aop:config>
  <aop:aspect id="traceAspect" ref="traceAdvice">
    <aop:after-returning method="traceReturn"
      pointcut="args(memberId, info)"/>
  </aop:aspect>
</aop:config>
```

Advice가 적용될 메서드

```
public interface MemberService {  
    boolean update(String memberId, UpdateInfo info);  
}
```

args() 명시자의 타입과 다를 때

```
public interface MemberService {  
    boolean update(String memberId, Object info);  
}
```

메서드 선언에 사용된 객체의 타입이 args() 명시자의 타입과 다르다 하더라도 실제로 메서드에 전달되는 인자의 타입이 args()에서 지정한 것과 동일하다면 Advice가 적용된다.

```
MemberService service = context.getBean("memberService", MemberService.class);  
UpdateInfo updateInfo = new UpdateInfo();
```

```
//실제로 전달되는 객체의 타입이 UpdateInfo이므로 적용됨  
service.update("홍길동", updateInfo);
```

```
@Aspect
public class UpdateMemberInfoTraceAspect {

    @AfterReturning(pointcut="args(memberId,info)", returning="result")
    public void traceReturn(JoinPoint joinPoint, Boolean result,
                           String memberId, UpdateInfo info) {

    }
}
```

@Aspect 어노테이션을 사용하는 경우에는 Pointcut 표현식에 args()명시자를 사용한다.

인자의 이름 매핑 처리

```
@Aspect
public class UpdateMemberInfoTraceAspect {

    @AfterReturning(pointcut="args(memberId, info)", argNames="memberId, info")
    public void traceReturn(String memberId, UpdateInfo info) {
        //
    }
}
```

argNames는 파라미터 이름을 순서대로 표시해서 Pointcut 표현식에서 사용된 이름이 몇번째 파라미터인지 검색할 수 있도록 한다.

```
@Aspect
```

```
public class UpdateMemberInfoTraceAspect {
```

```
    @AfterReturning(pointcut="args(memberId,info)", argNames="joinPoint.memberId,info")
```

```
    public void traceReturn(JoinPoint joinPoint, String memberId, UpdateInfo info) {
```

```
        //
```

```
    }
```

첫 번째 파라미터의 타입이 JoinPoint나 ProceedingJoinPoint라면 첫번째 파라미터를 제외한 나머지 파라미터의 이름을 argNames 속성에 입력한다.

```
<aop:after-returning
```

```
    method="traceReturn"
```

```
    pointcut="args(memberId, info)"
```

```
    returning="result" arg-names="joinPoint,memberId,Info"/>
```

**** Pointcut 표현식에서 사용된 파라미터 개수와 실제 구현 메서드의 파라미터 개수가 다르다면 예외 발생.**

AspectJ의 Pointcut 표현식

```
<bean id="cacheAdvice" class="madvirus.spring.chap05.aop.pojo.ArticleCacheAdvice" />
<aop:config>
    <aop:aspect id="cacheAspect" ref="cacheAdvice">
        <aop:around method="cache"
            pointcut="execution(public * *..ReadArticleService.*(..))"/>
    </aop:aspect>
</aop:config>
```

<aop:태그>를 이용하여 Aspect를 설정하는 경우
Execution 명시자를 이용하여 Advice가 적용될 Pointcut를 설정함

AspectJ는 Pointcut를 명시할 수 있는 다양한 명시자를 제공하지만
스프링은 메서드 호출과 관련된 명시자만을 지원함.

Pointcut 명시자의 종류

- execution
- within
- bean

execution

Advice를 적용할 때 메서드를 명시할 때 사용

형식 :

`execution(접근명시자 리턴타입 클래스이름?메소드이름(파라미터))`

1. 접근명시자 패턴은 생략 가능
2. 각 패턴은 *을 사용하여 모든 값을 표현할 수 있다.
3. ..을 이용하여 0개 이상이라는 의미를 표현한다.

execution의 사용 예

- `execution(* madvirus.spring.chap05.*.*())`
 - `madvirus.spring.chap05` 패키지의 파라미터가 없는 모든 메서드 호출
- `execution(* madvirus.spring.chap05..*.*(..))`
 - `madvirus.spring.chap05` 패키지 및 하위 패키지에 있는 파라미터가 0개 이상인 메서드
- `execution(Integer madvirus.spring.chap05..WriteArticleService.write(..))`
 - 리턴 타입이 `Integer`인 `WriteArticleService` 인터페이스의 `write` 메서드 호출
- `execution(* get*(*))`
 - 이름이 `get`으로 시작하고 1개의 파라미터를 갖는 메서드 호출
- `execution(* get*(*,*))`
 - 이름이 `get`으로 시작하고 2개의 파라미터를 갖는 메서드 호출
- `execution(* read*(Integer, ..))`
 - 메서드 이름이 `read`로 시작하고 첫 번째 파라미터 타입이 `Integer`이며 1개 이상의 파라미터를 갖는 메서드 호출

within 명시자

- 메서드가 아닌 특정 타입에 속하는 메서드를 Pointcut으로 설정 할 때 사용
- **within(madvirus.spring.chap05.board.service.WriteArticleService)**
WriteArticleService 인터페이스의 모든 메서드 호출
- **within(madvirus.spring.chap05.board.service.*)**
madvirus.spring.chap05.board.service 패키지에 있는 모든 메서드 호출
- **within(madvirus.spring.chap05.board..*)**
madvirus.spring.chap05.board 패키지 및 하위 패키지에 있는 모든 메서드 호출

bean 명시자

- 스프링 2.5 버전 부터 추가
- bean 이름을 이용하여 Pointcut를 정의
- **bean(writeArticleService)**
이름이 writeArticleService인 빈의 메서드 호출
- **bean(*ArticleService)**
이름이 ArticleService로 끝나는 빈의 메서드 호출

Spring Web MVC 3.x

MVC란 ?

MVC(Model – View – Contrlloer) 패턴은 코드를 기능에 따라 Model, View, Controller 3가지 요소로 분리한다.

MVC패턴은 UI코드와 비즈니스 코드를 분리 함으로써 종속성을 줄이고 재사용성을 높이고 보다 쉬운 변경이 가능하도록 한다.

Model	어플리케이션의 데이터와 비즈니스 로직을 담는 객체이다.
View	Model의 정보를 사용자에게 표시한다.
Controller	Model과 view의 중계역할을 한다. 사용자의 요청을 받아 Model에 변경된 상태를 반영하고 응답을 위한 View를 선택한다.

Spring 3.x MVC

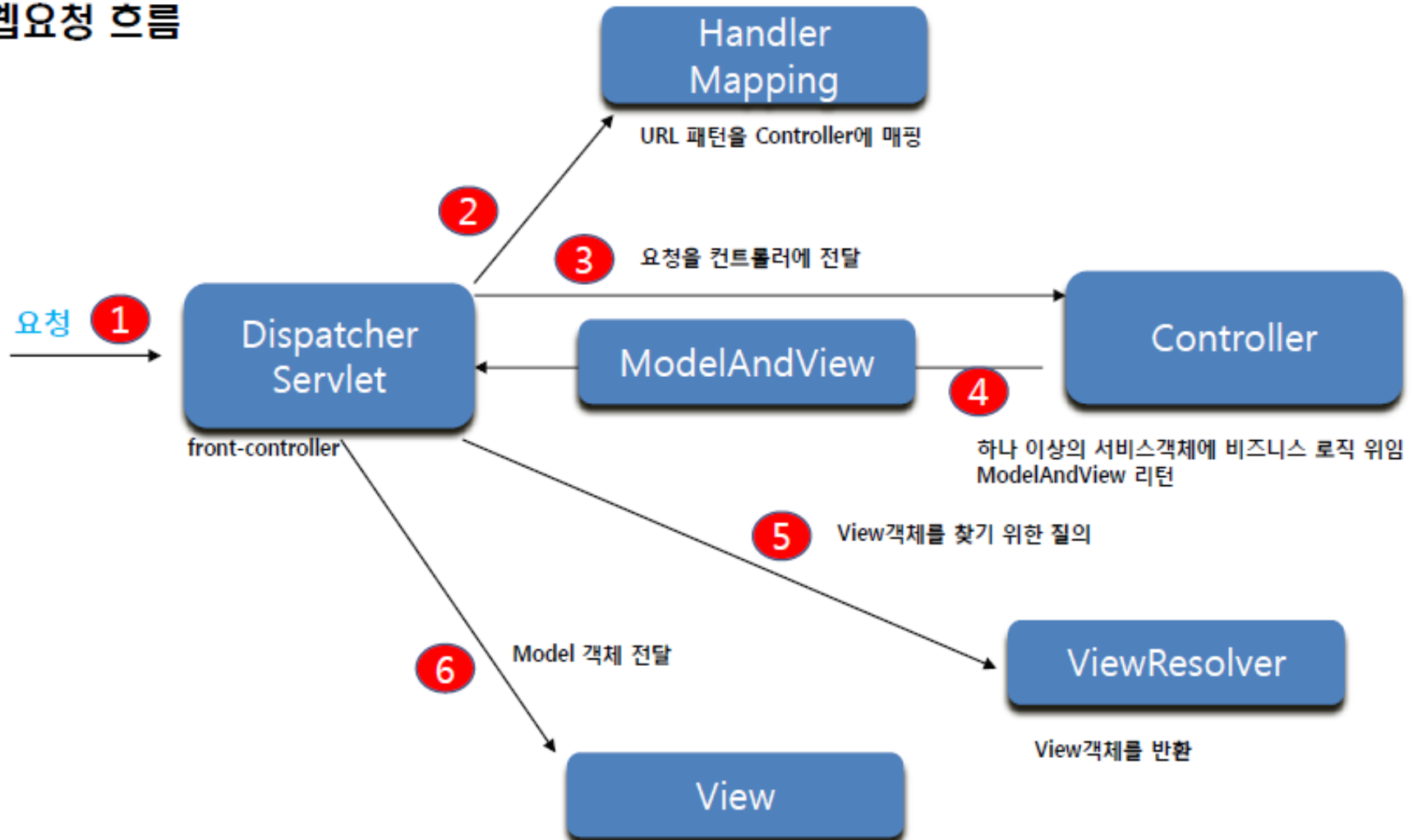
- ① 2.x에서는 특정 Controller를 상속 하거나 implements 해야 하나 3.x에서는 POJO 코딩 방식이 가능하다.
- ② 2.x에서는 Controller마다 클래스가 필요하여 많은 컨트롤러 클래스가 필요하고 설정문서가 길어 졌으나 3.x에서는 클래스 하나에 메소드로 사용자 요청 처리가 가능하다.
- ③ 3.X에서 Controller부분이 많이 변했으며 MVC의 어노테이션을 권장함.
- ④ org.springframework.web.servlet.mvc 안의 Abstractcontroller를 제외한 대부분이 Deprecated 되었음.

Spring MVC의 핵심 Component

- DispatcherServlet
 - Spring MVC Framework의 Front Controller, 웹요청과 응답의 Life Cycle을 주관한다.
- HandlerMapping
 - 웹요청시 해당 URL을 어떤 Controller가 처리할지 결정한다.
- Controller
 - 비즈니스 로직을 수행하고 결과 데이터를 ModelAndView에 반영한다.
- ModelAndView
 - Controller가 수행 결과를 반영하는 Model 데이터 객체와 이동할 페이지 정보(또는 View객체)로 이루어져 있다.
- ViewResolver
 - 어떤 View를 선택할지 결정한다.
- View
 - 결과 데이터인 Model 객체를 display한다.

Spring MVC 컴포넌트간의 관계와 흐름

웹요청 흐름



Spring MVC 컴포넌트간의 관계와 흐름

- Client의 요청이 들어오면 DispatcherServlet이 가장 먼저 요청을 받는다.
- HandlerMapping이 요청에 해당하는 Controller를 return한다.
- Controller는 비즈니스 로직을 수행(호출)하고 결과 데이터를 ModelAndView에 반영하여 return한다.
- ViewResolver는 view name을 받아 해당하는 View 객체를 return한다.
- View는 Model 객체를 받아 rendering한다.



Spring MVC 준비

- ① Dynamic WebProject를 생성한다.
- ② WebContent/WEB-INF/lib 폴더에 관련 라이브러 추가한다.
(3.x 라이브러리, jstl.jar , standard.jar , servlet-api.jar)
- ③ web.xml 문서에 DispatcherServlet 등록한다.
- ④ web.xml문서에 등록된 서블릿이름-servlet.xml 문서 만든다.

DispatcherServlet 등록

```
<servlet>
  <servlet-name>springWeb</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>springWeb</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
</web-app>
```

WEB-INF 폴더에
springWeb-servlet.xml 만들기

DispatcherServlet 등록

```
<servlet>
  <servlet-name>springWeb</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-
name>
    <param-value>
      /WEB-INF/springConfig/springWeb-servlet.xml
    </param-value>
  </init-param>
</servlet>
</web-app>
```

Servlet 설정 파일 원하는 폴더에 설정하
기

Controller관련 어노테이션

@Controller	해당 클래스가 Controller임을 나타내기 위한 어노테이션
@RequestMapping	요청에 대해 어떤 Controller, 어떤 메소드가 처리할지를 맵핑하기 위한 어노테이션
@RequestParam	Controller 메소드의 파라미터와 웹요청 파라미터와 맵핑하기 위한 어노테이션
@ModelAttribute	Controller 메소드의 파라미터나 리턴값을 Model 객체와 바인딩하기 위한 어노테이션
@SessionAttributes	Model 객체를 세션에 저장하고 사용하기 위한 어노테이션

@Controller

작성한 클래스에 @Controller를 붙여준다. 특정 클래스를 구현하거나 상속 할 필요없다.

```
import org.springframework.stereotype.Controller;

@Controller
public class HelloController {

    ...
}
```

@RequestMapping

요청에 대해 어떤 Controller, 어떤 메소드가 처리 할지를 mapping하기 위한 어노테이션이다.

관련속성

이름	타입	설명
value	String[]	URL 값으로 맵핑 조건을 부여한다. @RequestMapping(value="/hello.do") 또는 @RequestMapping(value={"/hello.do", "/world.do"})와 같이 표기하며, 기본값이기 때문에 @RequestMapping("/hello.do")으로 표기할 수도 있다. "/myPath/*.do"와 같이 Ant-Style의 패턴매칭을 이용할 수도 있다.
method	RequestMethod[]	HTTP Request 메소드값을 맵핑 조건으로 부여한다. HTTP 요청 메소드값이 일치해야 맵핑이 이루어 지게 한다. @RequestMapping(method = RequestMethod.POST)같은 형식으로 표기한다. 사용 가능한 메소드는 GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE이다

@RequestMapping

관련속성

params	String[]	<p>HTTP Request 파라미터를 맵핑 조건으로 부여한다.</p> <p>params="myParam=myValue"이면 HTTP Request URL중에 myParam이라는 파라미터가 있어야 하고 값은 myValue이어야 맵핑한다.</p> <p>params="myParam"와 같이 파라미터 이름만으로 조건을 부여할 수도 있고, "!myParam"하면 myParam이라는 파라미터가 없는 요청 만을 맵핑한다.</p> <p>@RequestMapping(params={"myParam1=myValue", "myParam2", "!myParam3"})와 같이 조건을 주었다면,</p> <p>HTTP Request에는 파라미터 myParam1이 myValue값을 가지고 있고, myParam2 파라미터가 있어야 하고, myParam3라는 파라미터는 없어야 한다.</p>
--------	----------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

@RequestMapping

@RequestMapping은 클래스 단위(type level)나 메소드 단위(method level)로 설정할 수 있다.

/hello.do 요청이 오면 HelloController의 hello 메소드가 수행된다.

type level에서 URL을 정의하고 Controller에 메소드가 하나만 있어도 요청 처리를 담당할 메소드 위에

@RequestMapping 표기를 해야 제대로 맵핑이 된다.

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/hello.do")
public class HelloController {

    @RequestMapping
    public String hello() {
        ...
    }
}
```

@RequestMapping

/hello.do 요청이 오면 hello 메소드,

/helloForm.do 요청은 GET 방식이면 helloGet 메소드, POST 방식이면 helloPost 메소드가 수행된다.

```
@Controller
public class HelloController {

    @RequestMapping(value="/hello.do")
    public String hello(){
        ...
    }

    @RequestMapping(value="/helloForm.do", method = RequestMethod.GET)
    public String helloGet(){
        ...
    }

    @RequestMapping(value="/helloForm.do", method = RequestMethod.POST)
    public String helloPost(){
        ...
    }
}
```

@RequestMapping

type level, method level 둘 다 설정할 수도 있는데,

이 경우엔 type level에 설정한 @RequestMapping의 value(URL)를 method level에서 재정의 할수 없다.

/hello.do 요청시에 GET 방식이면 helloGet 메소드, POST 방식이면 helloPost 메소드가 수행된다.

```
@Controller
@RequestMapping("/hello.do")
public class HelloController {

    @RequestMapping(method = RequestMethod.GET)
    public String helloGet() {
        ...
    }

    @RequestMapping(method = RequestMethod.POST)
    public String helloPost() {
        ...
    }
}
```


@RequestParam

- @RequestParam은 Controller 메소드의 파라미터와 웹요청 파라미터와 맵핑하기 위한 어노테이션이다.
- 관련 속성

이름	타입	설명
value	String	파라미터 이름
required	boolean	해당 파라미터가 반드시 필수 인지 여부. 기본값은 true이다.

@RequestParam

- 해당 파라미터가 Request 객체 안에 없을때 그냥 null값을 바인드 하고 싶다면, 아래 예제의 pageNo 파라미터 처럼 required=false로 명시해야 한다.
- name 파라미터는 required가 true이므로, 만일 name 파라미터가 null이면 org.springframework.web.bind.MissingServletRequestParameterException이 발생한다.

```
@Controller
public class HelloController {

    @RequestMapping("/hello.do")
    public String hello(@RequestParam("name") String name,
                       @RequestParam(value="pageNo", required=false) String pageNo){
        ...
    }
}
```

@ModelAttribute

- @ModelAttribute은 Controller에서 2가지 방법으로 사용된다.
 1. Model 속성(attribute)과 메소드 파라미터의 바인딩.
 2. 입력 폼에 필요한 참조 데이터(reference data) 작성. - SimpleFormContrller의 referenceData 메소드와 유사한 기능.
- 관련 속성

이름	타입	설명
value	String	바인드하려는 Model 속성 이름.

Spring 한글 인코딩 설정

Web.xml에 추가

```
<filter>
  <filter-name>charaterEncoding</filter-name>
  <filter-class>
    org.springframework.web.filter.CharacterEncodingFilter
  </filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>charaterEncoding</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

@SessionAttribute

- @SessionAttributes는 model attribute를 session에 저장, 유지할 때 사용하는 어노테이션이다.
- @SessionAttributes는 클래스 레벨(type level)에서 선언할 수 있다.
- 관련 속성

이름	타입	설명
value	String[]	session에 저장하려는 model attribute의 이름
required	Class[]	session에 저장하려는 model attribute의 타입

@Controller 메소드

기존의 계층형 Controller에 비해 유연한 메소드 파라미터,리턴값을 갖는다.

- Servlet API - ServletRequest, HttpServletRequest, HttpServletResponse, HttpSession 같은 요청,응답,세션관련 Servlet API.
- org.springframework.web.context.request.WebRequest, org.springframework.web.context.request.NativeWebRequest
- java.util.Locale
- java.io.InputStream / java.io.Reader
- java.io.OutputStream / java.io.Writer
- @RequestParam - HTTP Request의 파라미터와 메소드의 argument를 바인딩하기 위해 사용하는 어노테이션.
- java.util.Map / org.springframework.ui.Model / org.springframework.ui.ModelMap - 뷰에 전달할 모델데이터.
- Command/form 객체 - HTTP Request로 전달된 parameter를 바인딩한 커맨드 객체, @ModelAttribute을 사용하면 alias를 줄 수 있다.
- org.springframework.validation.Errors / org.springframework.validation.BindingResult - 유효성 검사 후 결과 데이터를 저장한 객체.
- org.springframework.web.bind.support.SessionStatus - 세션폼 처리시에 해당 세션을 제거하기 위해 사용된다.

@Controller 메소드

메소드 리턴타입

- **ModelAndView** - 커맨드 객체, @ModelAttribute 적용된 메소드의 리턴 데이터가 담긴 Model 객체와 View 정보가 담겨 있다.
- **Model(또는 ModelMap)** - 커맨드 객체, @ModelAttribute 적용된 메소드의 리턴 데이터가 Model 객체에 담겨 있다.
View 이름은 RequestToViewNameTranslator가 URL을 이용하여 결정한다.
- **Map** - 커맨드 객체, @ModelAttribute 적용된 메소드의 리턴 데이터가 Map 객체에 담겨 있으며, View 이름은 역시 RequestToViewNameTranslator가 결정한다
- **String** - 리턴하는 String 값이 곧 View 이름이 된다. 커맨드 객체, @ModelAttribute 적용된 메소드의 리턴 데이터가 Model(또는 ModelMap)에 담겨 있다. 리턴할 Model(또는 ModelMap)객체가 해당 메소드의 argument에 선언되어 있어야 한다
- **void** - 메소드가 ServletResponse / HttpServletResponse 등을 사용하여 직접 응답을 처리하는 경우이다. View 이름은 RequestToViewNameTranslator가 결정한다.

ViewResolver 등록

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
  <property name="prefix" value="/WEB-INF/jsp/" /> //결과페이지가 있을 위치  
  <property name="suffix" value=".jsp" /> <!-- 확장자 지정하기 -->  
</bean>
```

```
ModelAndView mv = new ModelAndView();  
mv.setViewName("hello");
```

=> /WEB-INF/jsp/hello.jsp 뷰가 보여진다.

웹 어플리케이션을 위한 ApplicationContext 설정

Web.xml에 추가

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/springConfig/commonServlet.xml
  </param-value>
</context-param>
```

웹 어플리케이션을 위한 ApplicationContext 설정

실제로 ContextLoaderListener와 DispatcherServlet은 각각 WebApplicationContext객체를 생성한다.

ContextLoaderListener가 생성하는 WebApplicationContext는 웹 어플리케이션에서 루트 컨텍스트가 되며 DispatcherServlet이 생성하는 WebApplicationContext는 자식 컨텍스트가 된다.

이때 자식은 root가 제공하는 빈을 사용할수 있기 때문에 각각의 DispatcherServlet이 공통으로 필요로 하는 빈을 ContextLoaderListener를 이용하여 설정하는 좋다.

컨트롤러 클래스 자동 스캔

Servlet문서에 추가

```
<context:component-scan base-package= "sist.test.exam"/>
```

만약, 컨트롤러 클래스 자동 스캔시 property를 통해 DI를 적용해야 한다면 메소드에 @Autowired를 선언한다.

View이름을 Redirect View로 지정하는 방법

Controller를 실행 후 ViewResolver가 실행되는 것이 아니라 다른 Controller로 이동해야 하는 경우

ex) 게시물의 글을 등록 후 select하는 Controller로 이동

```
@RequestMapping(value="multiInsert.do")
public ModelAndView multi(UserList user, Model map){
    //B/L 실행
    return new ModelAndView("redirect:redirectTest.do");
}
```

```
@RequestMapping(value="multiInsert.do")
public String multi(UserList user, Model map){
    //B/L 실행
    return "redirect:redirectTest.do";
}
```

View이름을 Redirect View로 지정하는 방법

Controller에서 Controller로 이동할 때 인수 넘기기
ex) get방식의 형태로 넘어간다.

```
@RequestMapping(value="multiInsert.do")
public ModelAndView multi(UserList user, Model map){

    map.addAttribute("no", 10);
    map.addAttribute("message", "안녕");

    return new ModelAndView("redirect:redirectTest.do");
}
```

```
@RequestMapping(value="/redirectTest.do")
public String redirectTest(UserList user, int no, String message){
    System.out.println("no = " + no + " , message = " + message);

    return "insert_ok";
}
```

List type의 property

```
<tr >
  <td align="center"><input type="checkbox" name="list[0].state"></td>
  <td><input type="text" name="list[0].id"></td>
  <td><input type="text" name="list[0].name"></td>
  <td><input type="text" name="list[0].age"></td>
</tr>
<tr>
  <td align="center"><input type="checkbox" name="list[1].state"></td>
  <td><input type="text" name="list[1].id"></td>
  <td><input type="text" name="list[1].name"></td>
  <td><input type="text" name="list[1].age"></td>
</tr>
<tr>
  <td align="center"><input type="checkbox" name="list[2].state"></td>
  <td><input type="text" name="list[2].id"></td>
  <td><input type="text" name="list[2].name"></td>
  <td><input type="text" name="list[2].age"></td>
</tr>
```

List type의 property

선택	아이디	이름	나이
<input type="checkbox"/>			
<input type="checkbox"/>			
<input type="checkbox"/>			

전송

List type의 property

```
@RequestMapping(value="multiInsert.do")
public ModelAndView multi(UserList user){
    System.out.println("multi 실행됨.");
    // ...
}
```

```
public class UserList {
    ArrayList<UserListModel> list;

    public ArrayList<UserListModel> getList() {
        return list;
    }

    public void setList(ArrayList<UserListModel> list) {
        this.list = list;
    }
}
```

List type의 property

```
@RequestMapping(value="multiInsert.do")
public ModelAndView multi(UserList user){
    System.out.println("multi 실행됨.");
    // ...
}
```

```
public class UserList {
    ArrayList<UserListModel> list;

    public ArrayList<UserListModel> getList() {
        return list;
    }

    public void setList(ArrayList<UserListModel> list) {
        this.list = list;
    }
}
```

```
public class UserListModel {
    private String id;
    private String name;
    private int age;
    private boolean state;
}
```


PathVariable어노테이션을 이용한 URITemplate 설정

- 일반적으로 DB테이블에 type= notice/ faq / qna 로 보통 만듦
- 요청은 /boardDetail.jsp?type=notice&pk=1023 =>http방식
단순화 /boardDetail/notice/1023

PathVariable어노테이션을 이용한 URI템플릿 설정

- 보통 `http://hostName/contextPath/hello.do?id=dev.won` 와 같은 URI 요청 패턴을 구성한다.

만약 `http://hostName/contextPath/hello/dev.won` 과 같이 URI를 구성해야 한다면 과연 어떻게 Controller와 매핑할것인지 알아보자.

위의 URI와 같이 구성하는 경우는 다음과 같은 경우가 있을것으로 보인다.

1. REST 서비스를 위하여 구성하는 경우

2. SEO 최적화(permanent link, canonical url등) 를 위한 URL 단순 구성

- 이런경우에 PathVariable어노테이션을 이용하면 아주 간편하게 URI 템플릿을 구성하여 Controller와 매핑 할 수 있다.

PathVariable 어노테이션을 이용한 URI 템플릿 설정

이때 다음의 두가지만 추가로 작업해주면 된다.

1. @RequestMapping 어노테이션의 값으로 {템플릿변수} 를 사용한다.
2. @PathVariable 어노테이션을 이용해서 {템플릿변수} 와 동일한 이름을 갖는 파라미터를 요청처리 메소드에 추가한다.

요청 : http://localhost:8000/springMVC3Exam/board/notice/list/3.do

```
@RequestMapping(value="/{board}/{boardType}/{action}/{idx}.do")
public String action(@PathVariable int idx ,
                    @PathVariable String boardType,
                    @PathVariable String action,
                    @PathVariable String board,
                    Model model){

    model.addAttribute("boardType", boardType);
    model.addAttribute("idx" , idx);
    model.addAttribute("action" , action);
    model.addAttribute("board" , board);

    System.out.println("action [1] !");
    return "pathResult";//뷰이름
}
```

PathVariable 어노테이션을 이용한 URI 템플릿 설정

```
/*
 * 요청 : http://localhost:8000/springMVC3Exam/test/action.do
 * */
@RequestMapping("/{data1}/action.do")
public String action2(@PathVariable("data1")
                      String data ,
                      Model model){
    model.addAttribute("param1", data);

    System.out.println("action2=>" + data);
    return "pathResult";
}
```

PathVariable 어노테이션을 이용한 URI 템플릿 설정

```
/**
 * PathVariable 어노테이션을 이용한 URI 템플릿 설정.
 * 만약 {템플릿변수} 의 이름과 요청처리 메소드의 파라미터 변수 이름이 다르다면, PathVariable("템플릿변수명")으로 지정하여 매핑 할수 있다.
 * @param type
 * @param action
 * @param no
 * @param model
 * @return
 *
 * 요청 : http://localhost:8000/springMVC3Exam/rest/notice/list/3
 */
@RequestMapping(value="/{boardType}/{action}/{idx}")
public String action2(@PathVariable("boardType")
    String type,
    @PathVariable
    String action,
    @PathVariable("idx")
    int no ,
    Model model){
    model.addAttribute("boardType", type);
    model.addAttribute("action" , action);
    model.addAttribute("idx" , no);

    System.out.println("action [2] !");
    return "pathResult";
}
```

@ResponseBody 어노테이션 사용

웹서비스 또는 REST 요청의 응답 내용은 JSP 에 의해 렌더링 되는 HTML이 아닌 XML 문자열 자체이다.

@RequestBody 어노테이션과 @ResponseBody 어노테이션은 각각 HTTP 요청 몸체를 자바 객체로 변환하고 자바 객체를 HTTP 응답 몸체로 변환해주는 데 사용한다.

1. @RequestBody 어노테이션은 HTTP 요청 몸체를 자바 객체로 전달 받을 수 있다.

2. @ResponseBody 어노테이션은 자바객체를 HTTP응답 몸체로 전송할 수 있다.

@ResponseBody 어노테이션 사용

```
<form action="<%=request.getContextPath()%>/bodyAnnotation/action.do" method="post">
    <input type="text" name="id" value="dev.won" size="3"><br>
    <input type="text" name="pw" value="1234" size="3"><br>
    <input type="submit">
</form>
```

```
@RequestMapping("/bodyAnnotation/action.do")
@ResponseBody //=> 뷰를 거치지 말고 요기에서 응답을 바로 클라이언트에게 전달.
//http://localhost:8888/SpringMvc3.xSample2/bodyAnnotation/action.do
public String action(){
    //리턴타입이 String이면 리턴.jsp였으나 @ResponseBody 이기에 문자열을 바로 클라이언트에게 전송
    return "응답데이터";
}
```

@ResponseBody 어노테이션 사용

<!-- @ResponseBody 사용시 변환값에 한글이 있으면 깨지는 현상을 해결하기 위한 설정 조치 -->

```
<bean
class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" >
  <property name="messageConverters">
    <list>
      <bean class =
"org.springframework.http.converter.StringHttpMessageConverter">
        <property name = "supportedMediaTypes">
          <list>
            <value>text/plain; charset=UTF-8</value>
          </list>
        </property>
      </bean>
    </list>
  </property>
</bean>
```


예외처리

@RequestMapping 메서드는 모든 타입의 예외를 발생 시킬 수 있다. 예외를 발생시킬 경우 웹 브라우저는 500 응답코드와 함께 서블릿 컨테이너가 출력한 에러 페이지가 출력된다.

예외타입에 따라 스프링 MVC와 연동된 뷰를 이용해서 에러 페이지를 출력 할 수 있다. 예외발생시 사용자에게 보여줄 특정 페이지를 만들어 출력한다. (공통의 예외를 한 페이지에서 처리 가능.)

- 처리 방법

- @ExceptionHandler 어노테이션을 이용한 예외처리
- *SimpleMappingExceptionHandlerResolver* 클래스를 이용한 예외처리

예외처리

- @ExceptionHandler 어노테이션을 이용한 예외처리
=> @ExceptionHandler 메소드를 만든 Controller영역에서만 유효함.

```
@ExceptionHandler(ArrayIndexOutOfBoundsException.class)
public String arithmetic(ArrayIndexOutOfBoundsException e){
    System.out.println("ArrayIndexOutOfBoundsException 실행됨 : " + e);
    return "array";
}
```

예외처리

- *SimpleMappingExceptionHandler* 클래스를 이용한 예외처리
=> xml문서에서 설정함.(오류 종류에 따라 다른 페이지 이동)

```
<!-- Exception 등록 -->
<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionHandler">
  <property name="exceptionMappings">
    <props>
      <prop key="java.lang.ArithmeticException">
        exception
      </prop>
      <prop key="java.lang.NullPointerException">
        exception2
      </prop>
    </props>
  </property>
</bean>
```

예외처리

- *SimpleMappingExceptionHandler* 클래스를 이용한 예외처리/ 와 *@ExceptionHandler* 어노테이션을 함께 사용하고자 할 때 xml문서에 아래와 같이 bean을 선언한다.

```
<!-- SimpleMappingExceptionHandler를 등록하였을 경우 기본으로 되어있던 어노테이션을 사용하려면 bean등록해줘야 함. -->
```

```
<bean
```

```
class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionHandler"/>
```

파일 업로드

-파일 업로드를 위해서 *MultipartResolver* 를 설정한다.
=> multipartResolver의 경우에는 반드시 bean의 id를 *multipartResolver*로 지정해야 한다.

```
<bean id= "multipartResolver"  
class= "org.springframework.web.multipart.commons.CommonsMultipartResolver" />
```

파일 업로드

<h2> 파일 업로드 기능</h2>

<form action= "upload.do"

method="post"

enctype="multipart/form-data">

이름 : <input type= "text" name="name"/> <p>

파일 첨부 : <input type= "file" name="file"/>

<input type= "submit" value="전송"/>

</form>

파일 업로드

```
@RequestMapping(value="/upload.do")
public ModelAndView upload(String name ,
    @RequestParam(value="file") MultipartFile file){

    String fName = file.getOriginalFilename();//첨부된파일이름
    long size = file.getSize();//첨부된파일용량

    //파일저장
    try {
        file.transferTo(new File("D:/uploadSave/"+fName));
    } catch (Exception e) {
        System.out.println(e+"=> 파일저장실패");
    }

    ModelAndView m = new ModelAndView();
    m.addObject("fName", fName);
    m.addObject("size", size);
    m.addObject("name", name);
    m.setViewName("uplod");
    return m;
}
```

파일 다운로드

Controller가 리턴하는 ViewName과 동일한 이름을 갖는 Bean을(Bean id="[BeanName]" 뷰 객체로 사용한다.
주로 CustomView 클래스를 뷰로 사용해야 하는 경우에 사용.

```
<bean id="viewResolver"  
      class="org.springframework.web.servlet.view.BeanNameViewResolver" >  
    <property name="order" value="1" />  
</bean>
```

```
<!-- ViewResolver를 등록 -->  
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" >  
    <property name="prefix" value="/WEB-INF/view/" />  
    <property name="suffix" value=".jsp" />  
    <property name="order" value="2"/>  
</bean>
```


파일 다운로드

```
@RequestMapping(value="/down.do")
public ModelAndView down(String pathName){
    System.out.println("pathName =" + pathName);

    File f = new File("D:/uploadSave/"+pathName);

    ModelAndView m =new ModelAndView();
    m.addObject("downPath", f);
    m.setViewName("customView");//customView를 bean의 이름인식

    return m;
}
```

<!-- 다운로드 기능을 담당하는 클래스 선언 -->

<bean id="customView" class="soa.spring3.down.DownLoadCustomView"/>

파일 다운로드

```
public class DownloadCustomView extends AbstractView{  
    public DownloadCustomView(){  
        this.setContentType("application/download;charset=UTF-8");  
    }  
    @Override  
    protected void renderMergedOutputModel(Map<String, Object> model  
        HttpServletRequest request, HttpServletResponse response) throws Exception {  
        File file = (File) model.get("downPath");//downPath  
  
        response.setContentType(this.getContentType());  
        response.setContentLength((int)file.length());  
    }  
}
```