

Spring framework

Spring 이란?



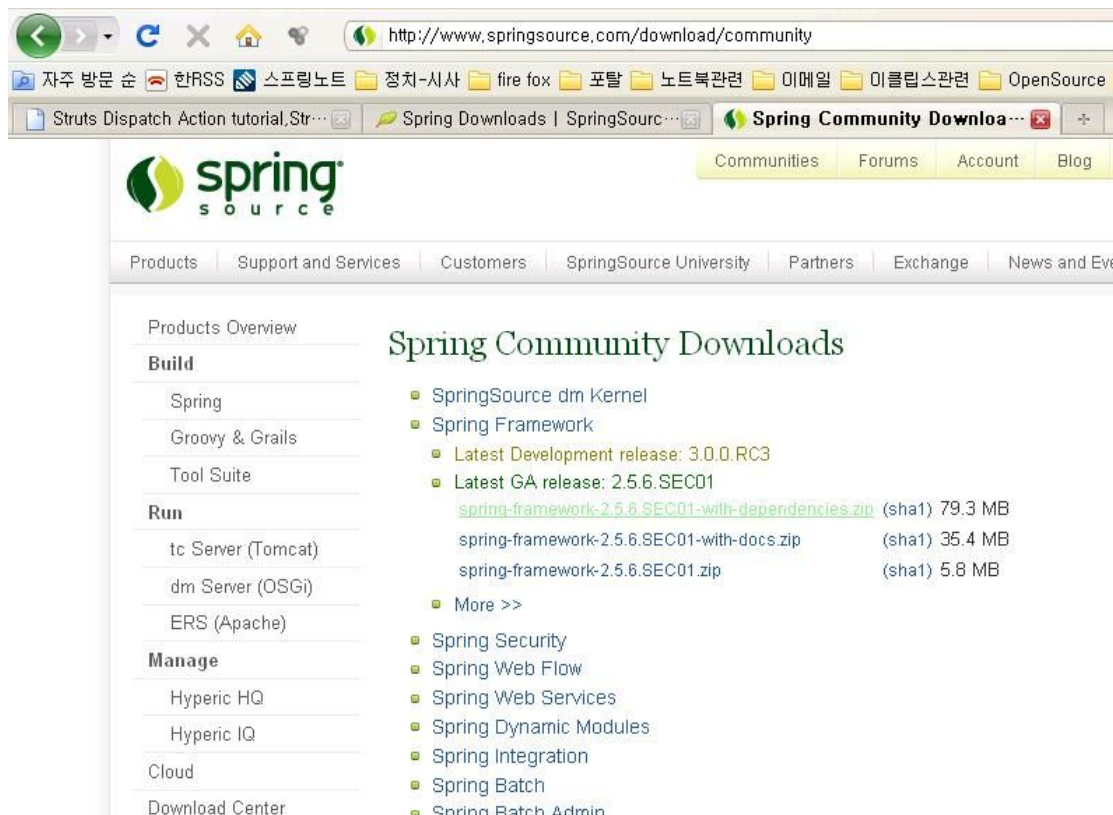
- 오픈 소스 프레임워크
 - Rod Johnson 창시
 - Expert one-on-one J2EE Design - Development, 2002, Wrox
 - Expert one-on-one J2EE Development without EJB, 2004, Wrox
 - 엔터프라이즈 어플리케이션 개발의 복잡성을 줄여주기 위한 목적
 - EJB 사용으로 수행되었던 모든 기능을 일반 POJO(Plain Old Java Object) 를 사용해서 가능하게 함.
 - 경량 컨테이너(light weight container)
 - www.springframework.org
- 주요 개념
 - 의존성 주입(**Dependency Injection**)
 - 관점 지향 프로그래밍(**Aspect-Oriented Programming**)

Spring 장점

- 경량 컨테이너 - 객체의 라이프 사이클 관리,
Java EE 구현을 위한 다양한 API제공
- DI (Dependency Injection) 지원
- AOP (Aspect Oriented Programming) 지원
- POJO (Plain Old Java Object) 지원
- 다양한 API와의 연동 지원을 통한 Java EE 구현
가능

Spring Container 설치

- 스프링 커뮤니티 사이트 <http://www.springsource.org/>
- 다운로드
<http://www.springsource.org/download/community>
- spring-framework-XXX-with-dependencies.zip 을 다운 받는다.



Spring IDE 이클립스 Plugin 설정 (1/2)

- 상단 메뉴 : help-install new software
 - update site :
<http://dist.springframework.org/release/IDE>

Available Software

Check the items that you wish to install.



Work with:

Find more software by working with the 'Available Software Sites' preferences.

type filter text

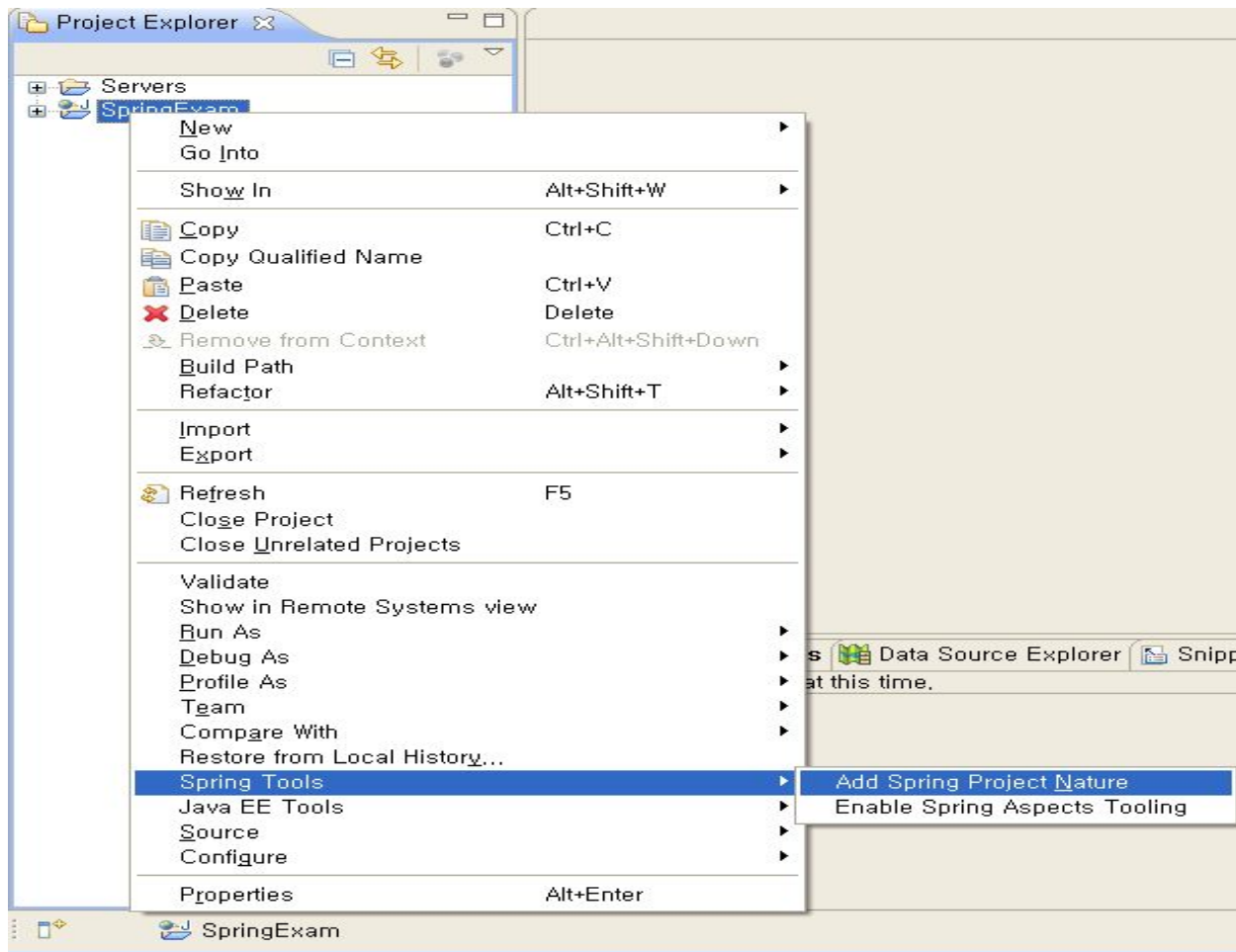
Name	Version
<input type="checkbox"/> Core / Spring IDE	
<input type="checkbox"/> Extensions (Incubation) / Spring IDE	
<input type="checkbox"/> Extensions / Spring IDE	
<input type="checkbox"/> Integrations / Spring IDE	
<input type="checkbox"/> Resources / Spring IDE	

Details

☒ Show only the latest versions of available software ☐ Hide items that are already installed
☒ Group items by category What is [already installed?](#)
☒ Contact all update sites during install to find required software

Spring IDE 이클립스 Plugin 설정 (2/2)

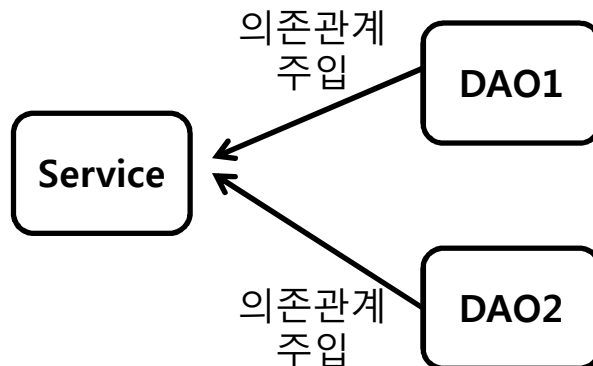
- Project 생성
- 오른 마우스 메뉴-Spring Tools-Add Spring Project Nature 선택



Dependency Injection (의존성 주입)

의존성 주입 (Dependency Injection, DI)

- 의존 관계 주입 (dependency injection)
 - 객체간의 의존관계를 객체 자신이 아닌 외부의 조립기가 수행한다.
 - 제어의 역행 (inversion of control, IoC) 이라는 의미로 사용되었음.
 - Martin Fowler, 2004
 - 제어의 어떠한 부분이 반전되는가라는 질문에 '의존 관계 주입'이라는 용어를 사용
 - 복잡한 어플리케이션은 비즈니스 로직을 수행하기 위해서 두 개 이상의 클래스들이 서로 협업을 하면서 구성됨.
 - 각각의 객체는 협업하고자 하는 객체의 참조를 얻는 것에 책임성이 있음.
 - 이 부분은 **높은 결합도(highly coupling)**와 테스트하기 어려운 코드를 양산함.
 - DI를 통해 시스템에 있는 각 객체를 조정하는 외부 개체가 객체들에게 생성시에 의존관계를 주어 줌.
 - 즉, 의존이 객체로 주입됨.
 - 객체가 협업하는 객체의 참조를 어떻게 얻어낼 것인가라는 관점에서 책임성의 역행(inversion of responsibility)임.
 - 느슨한 결합(loose coupling)이 주요 강점
 - 객체는 인터페이스에 의한 의존관계만을 알고 있으며, 이 의존관계는 구현 클래스에 대한 차이를 모르는채 서로 다른 구현으로 대체가 가능



Spring의 DI 지원

- Spring Container가 DI 조립기(Assembler)를 제공
 - 스프링 설정파일을 통하여 객체간의 의존관계를 설정한다.
 - Spring Container가 제공하는 api를 이용해 객체를 사용한다.

Spring 설정파일

- Spring Container가 어떻게 일할 지를 설정하는 파일
 - Spring container는 설정파일에 설정된 내용을 읽어 Application에서 필요한 기능들을 제공한다.
- XML 기반으로 작성한다.
- Root tag는 `<beans>` 이다
- 파일명은 상관없다.

예) 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">

</beans>
```

Bean객체 주입 받기 - 설정파일 설정(1/2)

- 주입 할 객체를 설정파일에 설정한다.
 - **<bean>** : 스프링컨테이너가 관리할 Bean객체를 설정
 - 기본 속성
 - **name** : 주입 받을 곳에서 호출 할 이름 설정
 - **id** : 주입 받을 곳에서 호출할 이름 설정 ('/' 값으로 못 가짐)
 - **class** : 주입할 객체의 클래스
 - **factory-method** : 객체를 생성해 주는 **factory** 메소드 호출 시
 - » 주로 Singleton 패턴 구현 클래스 객체 호출 시

```
<?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="dao" class="spring.di.model.MemberDAO"/>

</beans>
```

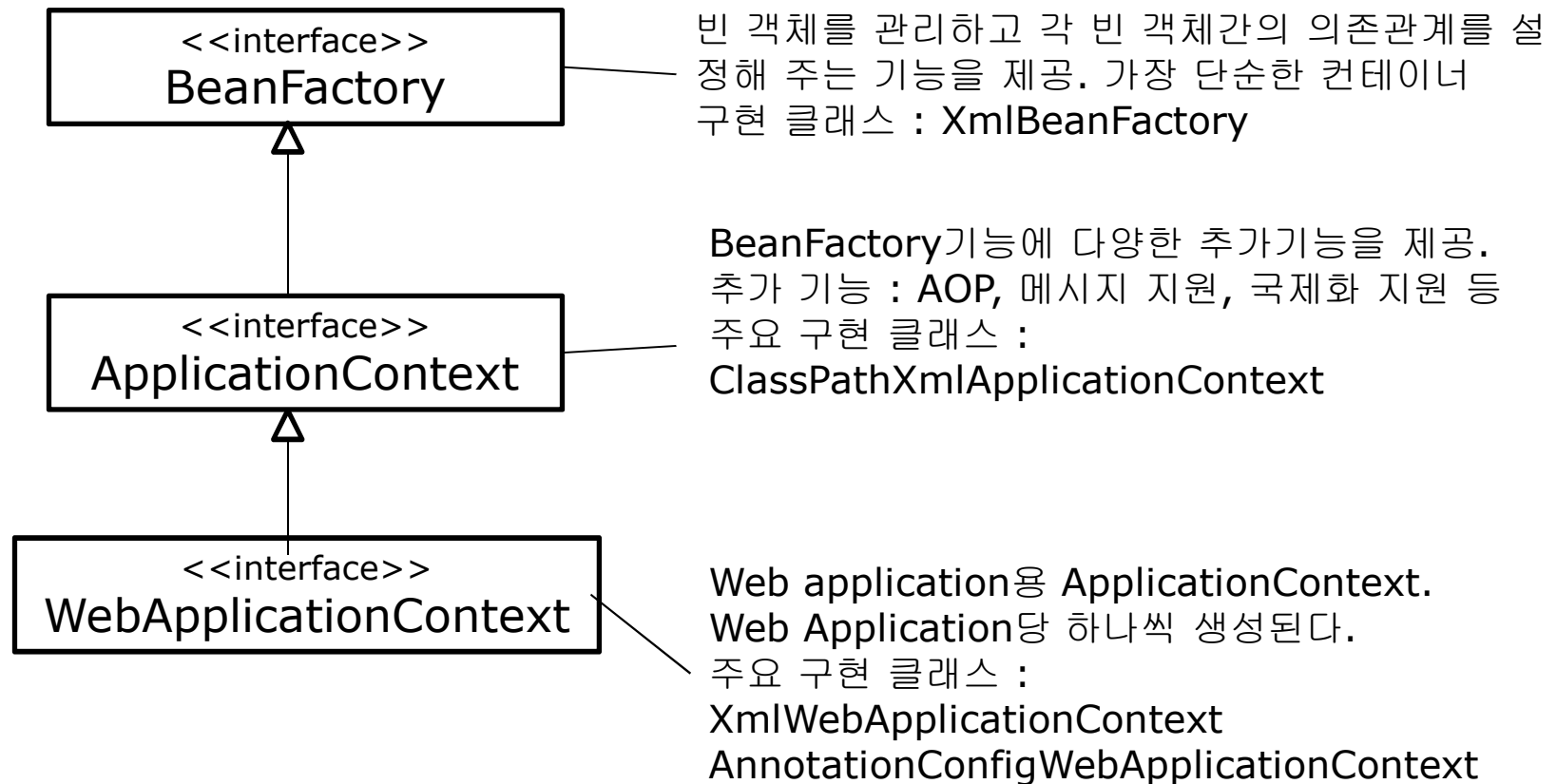
Bean객체 주입 받기 - 설정 Bean 사용(2/2)

- 설정 파일에 설정한 내용을 바탕으로 Spring API를 통해 객체를 주입 받는다.
 - 설정파일이 어디 있는지 설정
 - 객체를 만들어 주는 (Assembler) 객체 생성

```
public static void main(String [] args){  
  
    //스프링 컨테이너 객체 생성  
    ApplicationContext ctx =  
        new ClassPathXmlApplicationContext("applicationContext.xml");  
    //설정파일에 설정한 <bean> 태그의 id/name을 통해 객체를 받아온다.  
    MemberDAO dao = (MemberDAO)ctx.getBean("dao");  
  
}
```

Spring Container 객체

- Spring Container : 객체를 관리하는 컨테이너.
 - 다음 아래의 interface들을 구현한다.



설정을 통한 객체 주입 – Constructor를 이용(1/4)

- 객체 또는 값을 생성자를 통해 주입 받는다.
- **<constructor-arg>** : 하나의 argument 지정
 - **<bean>**의 하위태그로 설정한 bean 객체 또는 값을 생성자를 통해 주입하도록 설정
 - 설정 방법 : **<ref>**, **<value>**와 같은 하위태그를 이용하여 설정, 속성을 이용해 설정
 - 하위태그 이용
 - **<ref bean="bean name"/>** - 객체를 주입 시
 - **<value>값</value>** - 문자(String), Primitive data 주입 시
 - type 속성 : 값을 1차로 String으로 처리한다. 값의 타입을 명시해야 하는 경우 사용. ex) **<value type="int">10</value>**
 - 속성 이용
 - **ref="bean 이름"**
 - **value="값"**

설정을 통한 객체 주입 – Constructor를 이용(2/4)

값을 주입 받을 객체

```
package to;
public class PersonTO{
    private String id,
    private String name,
    private int age;

    public Person(String id){...}           //1번 생성자
    public Person(String id, String name){...} //2번 생성자
    public Person(int age){...}             //3번 생성자
}
```

1번 생성자에 주입 예

```
<bean id="person" class="to.PersonTO">
    <constructor-arg>
        <value>abcde</value>
    </constructor-arg>
</bean>
또는
<bean id="person" class="to.PersonTO">
    <constructor-arg value="abc"/>
</bean>
```

설정을 통한 객체 주입 – Constructor를 이용(3/4)

2번 생성자에 주입 예

```
<bean id="person" class="to.PersonTO">  
  <constructor-arg>  
    <value>abcde</value>  
  </constructor-arg>  
  <constructor-arg>  
    <value>Hong Gil Dong</value>  
  </constructor-arg>  
</bean>
```

또는

```
<bean id="person" class="to.PersonTO">  
  <constructor-arg value="abc"/>  
  <constructor-arg value="Hong Gil Dong"/>  
</bean>
```

3번 생성자에 주입 예

```
<bean id="person" class="to.PersonTO">  
  <constructor-arg>  
    <value type="int">30</value>  
  </constructor-arg>  
</bean>
```

또는

```
<bean id="person" class="to.PersonTO">  
  <constructor-arg value="30" type="int"/>  
</bean>
```


설정을 통한 객체 주입 - Constructor를 이용(4/4)

- bean객체를 주입

값을 주입 받을 객체

```
public class BusinessService{  
    private Dao dao = null;  
    public BusinessService(Dao dao){  
        this.dao = dao;  
    }  
}
```

```
<bean id="dao" class="spring.di.model.OracleDAO"/>  
<bean id="service" class="spring.di.model.service.BusinessService">  
    <constructor-arg>  
        <ref bean = "dao"/>  
    </constructor-arg>  
</bean>  
또는  
<bean id="service" class="spring.di.model.service.BusinessService">  
    <constructor-arg ref="dao">  
</bean>
```

설정을 통한 객체 주입 – Property를 이용(1/5)

- property를 통해 객체 또는 값을 주입 받는다.- setter 메소드
 - 주의 : setter를 통해서만 하나의 값을 받을 수 있다.
- <property> : <bean>의 하위태그. 설정한 bean 객체 또는 값을 property를 통해 주입하도록 설정
 - 속성 : name – 값을 주입할 property 이름 (setter의 이름)
 - 설정 방법
 - <ref>, <value>와 같은 하위태그를 이용하여 설정
 - 속성을 이용해 설정
 - xml namespace를 이용하여 설정

설정을 통한 객체 주입 – Property를 이용(2/5)

- 하위태그를 이용한 설정
 - `<ref bean="bean name"/>` - 객체를 주입 시
 - `<value>값</value>` - 문자(String) Primitive data 주입 시
 - type 속성 : 값의 타입을 명시해야 하는 경우 사용.
- 속성 이용
 - `ref="bean 이름"`
 - `value="값"`
- XML Namespace를 이용
 - `<beans>` 태그의 스키마설정에 namespace등록
 - `xmlns:p="http://www.springframework.org/schema/p"`
 - `<bean>` 태그에 속성으로 설정
 - 기본데이터 주입 : `p:propertyname="value". ex) <bean p:id="a">`
 - bean 주입 : `p:propertyname-ref="bean_id"`
`ex) <bean p:dao-ref="dao">`

설정을 통한 객체 주입 – Property를 이용(3/5)

Primitive Data Type 주입

값을 주입 받을 객체

```
package spring.to;
public class Person{
    private String id,
    private String name,
    private int age;

    public void setId(String id) {...}
    public void setName(String name) {...}
    public void setAge(int age) {...}
```

```
<bean id="person" class="to.Person">
    <property name="name">
        <value>hong</value>
    </property>
    <property name="id" value="abcde"/>
    <property name="age" value="20"/>
</bean>
```

설정을 통한 객체 주입 – Property를 이용(4/5)

Bean 객체 주입

값을 주입 받을 객체

```
public class BusinessService{  
    private Dao dao = null;  
    public void setDao(Dao dao){...}  
}
```

```
<bean id="dao" class="spring.di.model.OracleDAO"/>  
<bean id="service" class="spring.di.model.service.BusinessService">  
    <property name="dao">  
        <ref bean="dao"/>  
    </property>  
</bean>  
또는  
<bean id="service" class="spring.di.model.service.BusinessService">  
    <property name="dao" ref="dao">  
</bean>
```

설정을 통한 객체 주입 – Property를 이용(5/5)

XML Namespace를 이용한 주입

값을 주입 받을 객체

```
public class BusinessService{
    private Dao dao = null;
    private int waitingTime = 0;
    public void setDao (Dao dao){...}
    public setWaitingTime(int wt){...}
}
```

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"
>
<bean name="dao" class="spring.di.model.OracleDAO"/>
<bean name="service" class="service.BusinessService"
      p:waitingTime="20"
      p:dao-ref="dao"/>
</beans>
```

Collection 객체 주입하기 (1/5)

- <property> 또는 <constructor-arg>의 하위 태그로 Collection 값을 설정하는 태그를 이용해 값 주입 설정

- 설정 태그

태그	Collection종류	설명
<list>	java.util.List	List 계열 컬렉션 값 목록 전달
<set>	java.util.Set	Set 계열 컬렉션 값 목록 전달
<map>	java.util.Map	Map 계열 컬렉션 에 key-value 의 값 목록 전달
<props>	java.util.Properties	Properties 에 key(String)-value(String)의 값 목록 전달

- Collection에 값을 설정 하는 태그
 - <ref> : <bean>으로 등록된 객체
 - <value> : 기본데이터
 - <bean> : 임의의 bean
 - <list>,<map>,<props>,<set> : 컬렉션
 - <null> : null

Collection 객체 주입하기 (2/5)

- <list>
 - List 계열 컬렉션이나 배열에 값들을 넣기.
 - 속성 : value-type - 값들의 type 지정. Fullyname으로 지정한다.
 - <ref>, <value> 태그를 이용해 값 설정
 - <ref bean="bean_id"/> : bean 객체 list에 추가
 - <value [type="type"]>값</value> : 문자열(String), Primitive 값 list에 추가

```
public void setMyList(List list){...}
```

```
<bean id="otherbean" class="to.OtherBean"/>
<bean id="myBean" class="to.MyTO">
  <property name="myList">
    <list>
      <value>10</value> ->String으로 저장됨
      <value type="java.lang.Integer">20</value> ->Integer
                                                           로 저장됨
      <ref bean="otherbean"/>
    </list>
  </property>
</bean>
```


Collection 객체 주입하기 (3/5)

- <map>
 - Map 계열의 Collection에 객체들을 넣기
 - 속성 : key-type, value-type : key와 value의 타입을 고정시킬 경우 사용
 - <entry>를 이용해 key-value를 map에 등록
 - 속성
 - key, key-ref : key 설정
 - value, value-ref : 값 설정

```
public void setMyMap(Map map){...}
```

```
<bean id="otherbean" class="to.OtherBean"/>
<bean id="myBean" class="to.MyTO">
  <property name="myMap">
    <map>
      <entry key="id" value="abc"/>
      <entry key="other" value-ref="otherbean"/>
    </map>
  </property>
</bean>
```

Collection 객체 주입하기 (4/5)

- <props>
 - java.util.Properties 값(문자열)을 넣기
 - <prop>를 이용해 key-value를 properties에 등록
 - 속성
 - key : key값 설정
 - 값은 태그 사이에 넣는다. : <prop key="id">abcde</prop>

```
public void setJdbcProperty (Properties props){...}
```

```
<bean id="myDAO" class="dao.MyDAO">  
  <property name="jdbcProperty">  
    <props>  
      <prop key="driver">JDBC Driver</prop>  
      <prop key="url">jdbc:url://127.0.0.1/mydb</prop>  
      <prop key="user">dbUser</prop>  
      <prop key="pwd">dbPassword</prop>  
    </props>  
  </property>  
</bean>
```

Collection 객체 주입하기 (5/5)

- <set>
 - java.util.Set에 객체를 넣기
 - 속성 : value-type : value 타입 설정
 - <value>, <ref>를 이용해 값을 넣는다.

```
public void setMySet(Set props){...}
```

```
<bean id="otherbean" class="to.OtherBean"/>
<bean id="myBean" class="to.Bean">
  <property name="mySet">
    <set>
      <value>10</value>
      <value>20</value>
      <ref bean="otherbean"/>
    </set>
  </property>
</bean>
```

Bean 객체의 생성 단위 (1/2)

- BeanFactory를 통해 Bean을 요청시 객체생성의 범위(단위)를 설정
- <bean> 의 scope 속성을 이용해 설정
 - scope의 값

값	
singleton	컨테이너는 하나의 빈 객체만 생성한다. - default
prototype	빈을 요청할 때 마다 생성한다.
request	Http 요청마다 빈 객체 생성
session	HttpSession 마다 빈 객체 생성

- request, session은 WebApplicationContext에서만 적용 가능

Bean 객체의 생성 단위 (2/2)

- 빈(bean) 범위 지정
 - singleton과 prototype
 - `<bean id="dao" class="dao.OracleDAO" scope="prototype"/>`
 - prototype은 Spring 어플리케이션 컨텍스트에서 `getBean`으로 빈(bean)을 사용시마다 새로운 인스턴스를 생성함.
 - singleton은 Spring 어플리케이션 컨텍스트에서 `getBean` 으로 빈(bean)을 사용시 동일한 인스턴스를 생성함.

Factory 메소드를 통한 Bean 주입

- Factory 메소드로부터 빈(bean) 생성

```
public class OracleDAO{  
    private OracleDAO() {}  
    private static OracleDAO instance;  
    public static OracleDAO getInstance(){  
        if(instance==null)  
            instance = new OracleDAO();  
        return instance;  
    }  
}
```

Singleton 클래스는 static factory 메소드를 통해서 인스턴스 생성이 가능하면 단 하나의 인스턴스만을 생성함.

```
<bean id="dao" class="OracleDAO"  
      factory-method="getInstance"/>
```

* 주 : `getBean()`으로 호출 시 `private` 생성자도 호출 하여 객체를 생성한다.
그러므로 위의 상황에서 `factory` 메소드로만 호출 해야 객체를 얻을 수 있는 것은 아니다.

Spring AOP

Spring AOP 개요 (1/2)

- Application을 두가지 관점에 따라 구현
 - 핵심 관심 사항(core concern)
 - 공통 관심 사항 (cross-cutting concern)
- 기존 OOP 보완
 - 공통관심사항을 여러 모듈에서 적용하는데 한계가 존재
 - AOP 는 핵심 관심 사항과 공통관심 사항 분리하여 구현

Spring AOP 개요 (2/2)

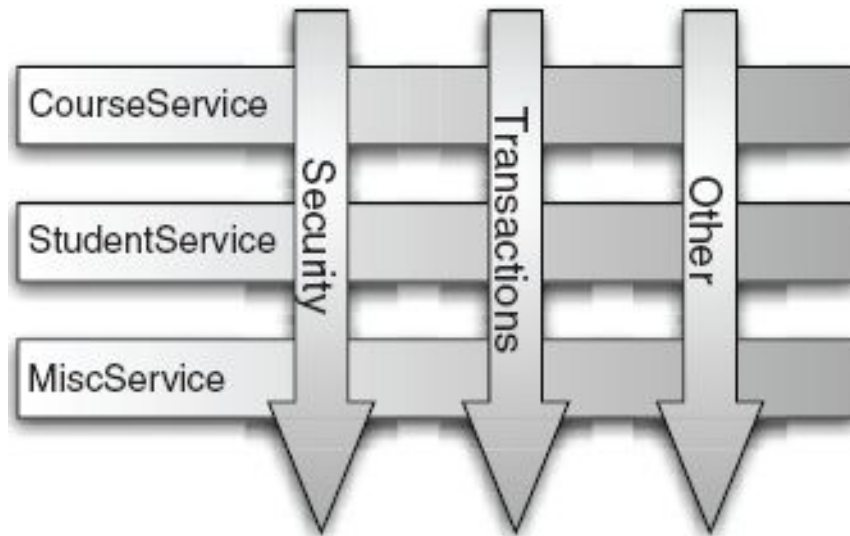


Figure 4.1
Aspects modularize cross-cutting concerns, applying logic that spans multiple application objects.

핵심관심사항 : CourseService, StudentService, MiscService

공통관심사항 : Security, Transactions, Other

- 핵심관심사항에 공통관심사항을 어떻게 적용시킬 것인가
-> **AOP**

Spring AOP 용어

- Target – 핵심사항(Core) 가 구현된 객체
- JoinPoint – 공통관심사항이 적용 될 수 있는 지점(ex:메소드 호출시, 객체생성시 등)
- Pointcut – JoinPoint 중 실제 공통사항이 적용될 대상을 지정.
- Advice
 - 공통관심사항(Cross-Cutting) 구현 코드 + 적용시점.
 - 적용 시점 : 핵심로직 실행 전, 후, 정상 종료 후, 비정상 종료 후, 전/후가 있다.
- Aspect – Advice + Pointcut
- Weaving – Proxy를 생성하는 것. (컴파일 시점, Class Loading 시점, 런타임 시점 Weaving이 있다.)

Spring에서 AOP 구현 방법

- AOP 구현
 - POJO Class를 이용한 AOP구현
 - Spring 설정 파일을 이용한 설정
 - 어노테이션(Annotation)을 이용한 설정
 - 스프링 API를 이용한 AOP구현

POJO 기반 AOP구현

- 설정파일에 AOP 설정.
 - XML 스키마 확장기법을 통해 설정파일을 작성한다.
- POJO 기반 공통관심사항 로직 클래스 작성

POJO 기반 AOP구현 - 설정파일 작성 (1/5)

- XML 스키마를 이용한 AOP 설정
 - aop 네임스페이스와 XML 스키마 추가

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:p="http://www.springframework.org/schema/p"
  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-2.5.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

</beans>
```

POJO 기반 AOP구현 - 설정파일 작성 (2/5)

- XML 스키마를 이용한 AOP 설정 예

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:p="http://www.springframework.org/schema/p"
  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-2.5.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <bean id="writelog" class="org.kosta.spring.LogAspect"/>

  <aop:config>
    <aop:pointcut id="publicmethod" expression="execution(public * org.kosta.spring..*.*(..))"/>
    <aop:aspect id="loggingAspect" ref="writelog">
      <aop:around pointcut-ref="publicmethod" method="logging"/>
    </aop:aspect>
  </aop:config>

  <bean id="targetclass" class="org.kosta.spring.TargetClass"/>

</beans>
```

POJO 기반 AOP구현 - 설정파일 작성 (3/5)

- AOP 설정 태그

1. <aop:config> : aop설정의 root 태그. Aspect 설정들의 묶음
2. <aop:aspect> : Aspect 설정 - 하나의 Aspect 설정
 - Aspect가 여러 개일 경우 <aop:aspect> 태그가 여러 개 온다.
3. <aop:pointcut> : Advice에서 참조할 pointcut 설정
4. Advice 설정태그들
 - A. <aop:before> - 메소드 실행 전 실행될 Advice
 - B. <aop:after-returning> - 메소드 정상 실행 후 실행될 Advice
 - C. <aop:after-throwing> - 메소드에서 예외 발생시 실행될 Advice
 - D. <aop:after> - 메소드 정상 또는 예외 발생 상관없이 실행될 Advice - finally
 - E. <aop:around> - 모든 시점에서 적용시킬 수 있는 Advice 구현

POJO 기반 AOP구현 - <aop:aspect> (4/5)

- 한 개의 Aspect (advice + pointcut)을 설정
- 속성
 - ref : 공통관심사항을 설정한 Bean(Advice 빈) 참조
 - id : 식별자
 - 다른 Aspect 태그와 구별하기 위한 식별자
- 자식태그
 - <aop:pointcut> : pointcut 지정
 - advice관련 태그가 올 수 있다.

```
<aop:config>
  <aop:aspect id="loggingAspect" ref="writelog">
    <aop:pointcut id="publicmethod"
      expression="execution(public * org.myspring..*.* (..))"/>
    <aop:around pointcut-ref="publicmethod" method="logging"/>
  </aop:aspect>
</aop:config>
```


POJO 기반 AOP구현 - <aop:pointcut> (5/5)

- Pointcut(공통기능이 적용될 곳)을 지정하는 태그
 - <aop:config>나 <aop:aspect>의 자식 태그
 - AspectJ 표현식을 통해 pointcut 지정
 - 속성 :
 - id : 식별자로 advice 태그에서 사용됨
 - expression : pointcut 지정
- ```
<aop:pointcut id="publicmethod"
 expression="execution(public * org.myspring..*.*(..))"/>
```

```
<aop:config>
 <aop:aspect id="loggingAspect" ref="writelog">
 <aop:pointcut id="publicmethod"
 expression="execution(public * org.myspring..*.*(..))"/>
 <aop:around pointcut-ref="publicmethod" method="logging"/>
 </aop:aspect>
</aop:config>
```

# POJO 기반 AOP구현 - AspectJ 표현식 (1/3)

- AspectJ에서 지원하는 패턴 표현식
- 스프링은 메서드 호출관련 명시자만 지원

명시자(pattern)  
-?는 생략가능

- 명시자
  - execution : 메소드 구문을 기준으로 지정
  - within : Class 명을 기준으로 지정
  - bean : 설정파일에 지정된 빈의 이름(name속성)을 이용해 지정. 2.5버전에 추가됨.

# POJO 기반 AOP구현 - AspectJ 표현식 (2/4)

- 표현

명시자(패턴)

-패턴은 명시자마다 다름.

예) **execution(public \* abc.def.\*Service.set\*(..))**

- 패턴문자.

- \* : 1개의 모든 값을 표현
  - argument에서 쓰인 경우 : 1개의 argument
  - package에 쓰인 경우 : 1개의 하위 package
  - 이름(메소드, 클래스)에 쓰일 경우 : 모든 글자들
- .. : 0개 이상
  - argument에서 쓰인 경우 : 0개 이상의 argument
  - package에 쓰인 경우 : 0개의 이상의 하위 package

- execution

- execution(수식어패턴? 리턴타입패턴 패키지패턴?.클래스명패턴.메소드명패턴 (argument패턴))
- 수식어패턴 : public, protected, 생략
- argument에 type을 명시할 경우 객체 타입은 **fullName**으로 넣어야 한다.
  - java.lang은 생략가능
- 위 예 설명

적용 하려는 메소드들의 패턴은 **public** 제한자를 가지며 리턴 타입에는 모든 타입이 다 올 수 있다. 이름은 **abc.def** 패키지와 그 하위 패키지에 있는 모든 클래스 중 **Service**로 끝나는 클래스들에서 **set**으로 시작하는 메소드이며 **argument**는 0개 이상 오며 타입은 상관 없다.

# POJO 기반 AOP구현 - AspectJ 표현식 (3/4)

- within
  - within(패키지패턴.클래스명패턴)
- bean
  - bean(bean이름 패턴)

# POJO 기반 AOP구현 - AspectJ 표현식 (4/4)

- 예

```
execution(* test.spring.*.*())
execution(public * test.spring..*.*())
execution(public * test.*.*.get*(*))
execution(String test.spring.MemberService.registMember(..))
execution(* test.spring..*Service.regist*(..))
execution(public * test.spring..*Service.regist*(String, ..))

within(test.spring.service.MemberService)
within(test.spring..MemberService)
within(test.spring.aop..*)

bean(memberService)
bean(*Service)
```

# POJO 기반 AOP구현

- POJO 기반 Advice클래스 작성
- 설정파일에 AOP 설정
  - Advice class를 Bean으로 설정
  - <aop:aspect> 태그를 이용해 Advice, Pointcut을 설정한다.

# POJO 기반 AOP구현 - Advice 설정 관련 태그

- 시점에 따른 5가지 태그
  - before, after-returning, after-throwing, after, around
- 공통 속성
  - pointcut-ref : pointcut 참조.
    - <aop:pointcut>태그의 id명을 넣어 pointcut지정
  - pointcut : 직접 pointcut을 설정 한다.
  - method : Advice bean에서 호출할 메소드명 지정

```
<bean id="writelog" class="org.kosta.spring.LogAspect"/>

<aop:config>
 <aop:aspect id="loggingAspect" ref="writelog">
 <aop:pointcut id="publicmethod"
 expression="execution(public * org.my.spring..*.*(..))"/>
 <aop:before pointcut-ref="publicmethod" method="logging"/>
 </aop:aspect>
</aop:config>
```

# POJO 기반 AOP구현 – Advice 클래스 작성(1/6)

- POJO 기반의 클래스로 작성한다.
  - 클래스 명이나 메서드 명에 대한 제한은 없다.
  - 설정파일에서 **Advice** 등록시 메소드 명을 등록한다.
    - **Advice** 태그의 **method** 속성에서 설정한다.
  - 메소드 구문은 호출되는 시점에 따라 달라 질 수 있다.



## POJO 기반 AOP구현 – Advice 클래스 작성(2/6)

- Before Advice

- 핵심 관심사항 메소드가 실행되기 전에 실행됨
- return type : 상관없으나 void로 한다.
- argument : 없거나 JoinPoint 객체를 받는다.

```
<aop:before pointcut-ref="publicmethod"
 method="beforeLogging" />
```



```
public void beforeLogging(){ }
```

# POJO 기반 AOP구현 – Advice 클래스 작성(3/6)


- After Returning Advice

- 핵심 관심사항 메소드 실행이 정상적으로 끝난 뒤 실행됨
- return type : 상관없으나 void로 한다.
- argument :
  - 없거나 JoinPoint 객체를 받는다. JoinPoint는 항상 첫 argument로 사용된다.
  - 대상 메소드에서 리턴 되는 값을 argument로 받을 수 있다.  
type : Object 또는 대상 메소드에서 return하는 value의 type

```
<aop:after-returning pointcut-ref="publicmethod"
 method="returnLogging"
 returning="retValue"/>
```

```
public void returnLogging(Object retValue){

}
```



# POJO 기반 AOP구현 – Advice 클래스 작성(4/6)

- After Throwing Advice
  - 핵심 관심사항 메소드 실행 중 예외가 발생한 경우 실행됨
  - return type : 상관없으나 void로 한다.
  - argument :
    - 없거나 JoinPoint 객체를 받는다. JoinPoint는 항상 첫 argument로 사용된다.
    - 대상메소드에서 전달되는 예외객체를 argument로 받을 수 있다.

```
<aop:after-throwing pointcut-ref="publicmethod"
 method="throwingLogging"
 throwing="ex"/>
```

```
public void throwingLogging(MyException ex){
 //대상객체에서 리턴되는 값을 받을 수는 있지만 수정할 수는 없다.
}
```

# POJO 기반 AOP구현 – Advice 클래스 작성(5/6)

- After Advice

- 핵심 관심사항 메소드 실행이 종료된 뒤 오류발생 여부와 상관없이 무조건 실행 된다.
- return type : 상관없으나 void로 한다.
- argument :
  - 없거나 JoinPoint 객체를 받는다.

```
<aop:after pointcut-ref="publicmethod"
 method="afterLogging" />
```

```
public void afterLogging(){
}
```



# POJO 기반 AOP구현 – Advice 클래스 작성(6/6)

- Around Advice

- 앞의 네 가지 Advice를 다 구현 할 수 있는 Advice.
- return type : Object 또는 void
- argument
  - [없거나] org.aspectj.lang.ProceedingJoinPoint를 argument로 지정한다.

```
<aop:around pointcut-ref="publicmethod" method="aroundLogging" />

public Object aroundLogging(ProceedingJoinPoint joinPoint) throws Throwable{
 //before 코드
 try{
 Object retValue = joinPoint.proceed(); //대상객체의 메소드 호출
 //after-returning 코드
 return retValue; //호출 한 곳으로 리턴 값 넘긴다. - 넘기기 전 수정 가능
 }catch(Throwable e){
 //after-Throwing 코드
 throw e;
 }finally{
 //after 코드
 }
}
```

# JoinPoint

- 대상객체에 대한 정보를 가지고 있는 객체로 Spring container로부터 받는다.
- org.aspectj.lang 패키지에 있음
- 반드시 Advice 메소드의 첫 argument로 와야 한다.
- 메소드들

Object getTarget() : 대상객체를 리턴

Object[] getArgs() : 파라미터로 넘겨진 값들을 배열로 리턴. 넘어온 값이 없으면 빈 배열객체가 return 됨.

Signature getSignature () : 호출 되는 메소드의 정보

- Signature : 호출 되는 대상객체에 대한 구문정보를 가진 객체

String getName() : 대상 메소드 명 리턴

String toShortString() : 대상 메소드 명 리턴

String toLongString() : 대상 메서드 전체 syntax를 리턴

String getDeclaringTypeName() : 대상메소드가 포함된 type을 return. (package명.type명)

# @Aspect 어노테이션을 이용한 AOP

- @Aspect 어노테이션을 이용하여 Aspect 클래스에 직접 Advice 및 Pointcut등을 직접 설정
- 설정파일에 <aop:aspectj-autoproxy/> 를 추가 해야함
- Aspect class를 <bean>으로 등록
- 어노테이션(Annotation)
  - @Aspect : Aspect 클래스 선언
  - @Before("pointcut")
  - @AfterReturning(pointcut="", returning="")
  - @AfterThrowing(pointcut="", throwing="")
  - @After("pointcut")
  - @Around("pointcut")
- Around를 제외한 나머지 메소드들은 첫 argument로 JoinPoint를 가질 수 있다.
- Around 메소드는 argument로 ProceedingJoinPoint를 가질 수 있다.

# Spring MVC

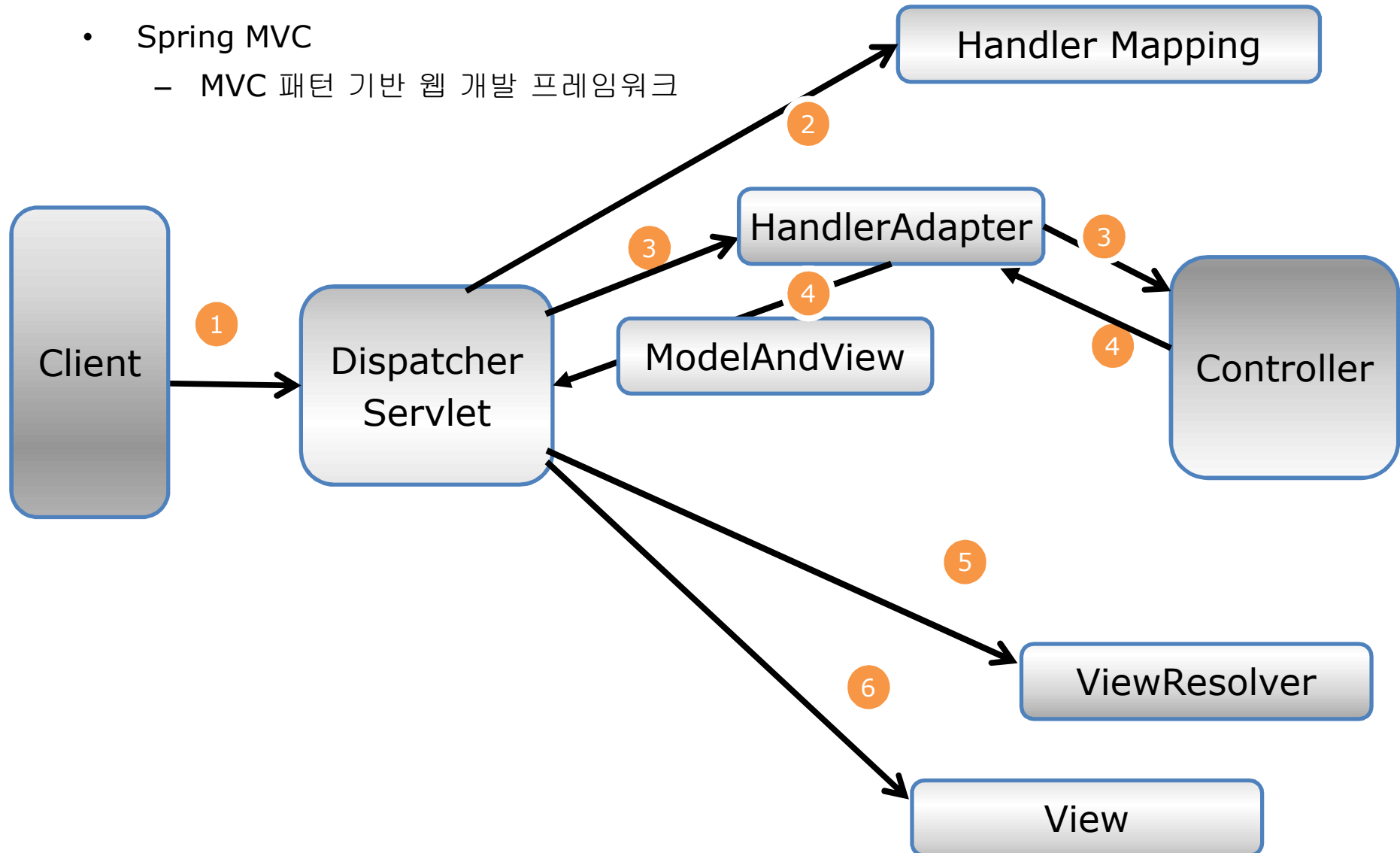


# Spring MVC 구성 주요 컴포넌트

- DispatcherServlet
  - Front Controller
- HandlerMapping
  - 클라이언트의 요청을 처리할 Controller를 찾는 작업 처리
- HandlerAdapter
  - Client의 요청 처리 Controller의 Handler메소드를 호출해주는 컴포넌트
- Controller(Handler)
  - 클라이언트 요청 처리를 수행하는 Controller.
- ViewResolver
  - 응답할 View를 찾는 작업을 처리
- View
  - 응답하는 로직을 처리
- ModelAndView
  - 응답할 View와 View에게 전달할 값을 저장하는 용도의 객체

# Spring MVC 흐름 (1/2)

- Spring MVC
  - MVC 패턴 기반 웹 개발 프레임워크



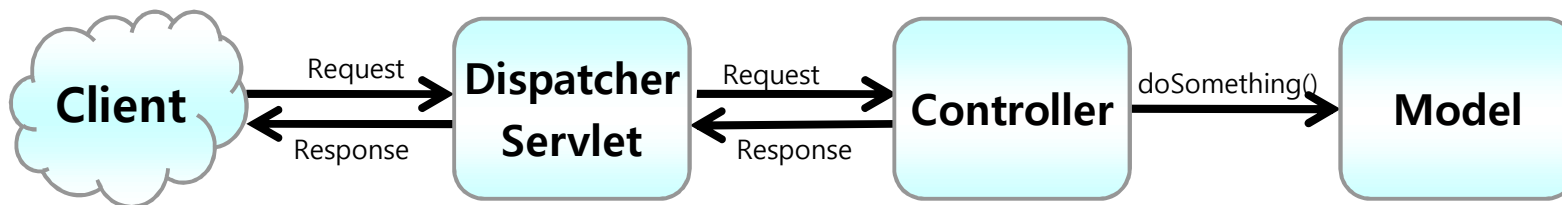
# Spring MVC 흐름 (2/2)

- 요청 처리 순서

- ① DispatcherServlet이 요청을 수신
  - 단일 Front controller servlet
  - 요청을 수신하여 처리를 다른 컴포넌트에 위임
  - 어느 컨트롤러에 요청을 전송할지 결정
- ② DispatcherServlet은 HandlerMapping에 어느 컨트롤러를 사용할 것인지 문의
- ③ DispatcherServlet은 요청을 HandlerAdapter를 이용해 컨트롤러의 Handler 메소드에게 전송한다.
  - 컨트롤러는 Business Logic등을 이용해 요청을 처리한다.
- ④ Controller는 요청을 처리하고 그 결과와 View에 대한 정보를 리턴하면 HandlerAdapter는 ModelAndView로 만들어 Dispatcher Servlet에게 전달한다.
- ⑤ DispatcherServlet은 HandlerAdapter에게 받은 ModelAndView 정보를 바탕으로 바탕으로 ViewResolver 에게 응답할 View를 요청
- ⑥ DispatcherServlet 은 받은 View를 이용해 응답 요청

# Spring MVC 구현 Step

- Spring MVC를 이용한 어플리케이션 작성 스텝
  1. web.xml에 DispatcherServlet 등록 및 Spring 설정파일 등록
  2. Spring 설정파일에 HandlerMapping 설정
  3. 컨트롤러 구현 및 Spring 설정파일에 등록
  4. View Resolver Spring 설정 파일에 등록
  5. JSP(or View) 작성 후 설정) 코드 작성



# DispatcherServlet 설정과 ApplicationContext (1/3)

- 공통 Spring 설정파일 등록(Root Application Context)
  - Root 레벨에서 사용할 설정 파일등록
  - Context 레벨 설정은 모든 설정이 공통적으로 사용할 설정(Beans 등)들을 등록한다.
  - 설정 파일 로드를 위한 Listener를 등록 하고 설정파일의 위치를 초기파라미터로 등록
  - 초기 파라미터로 설정파일 지정 안하면 /WEB-INF/applicationContext.xml 를 사용
  - 설정파일이 여러 개인 경우 공백이나 " , " 로 구분한다.

## Listener등록

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

## 설정파일 등록

```
<context-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>/WEB-INF/service-service.xml
 /WEB-INF/dao-data.xml
</param-value>
</context-param>
```

# DispatcherServlet 설정과 ApplicationContext (2/3)

- DispatcherServlet 설정
  - web.xml에 등록
  - 스프링 설정파일 : "<servlet-name>-servlet.xml" 이고 WEB-INF 아래 추가한다.
  - <url-pattern>은 DispatcherServlet이 처리하는 URL 매핑 패턴을 정의

```
<servlet>
 <servlet-name>dispatcher</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
 <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
 <servlet-name>dispatcher</servlet-name>
 <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

- **Spring Container는 설정파일의 내용을 읽어 ApplicationContext 객체를 생성한다.**
- 설정 파일명 : dispatcher-servlet.xml – MVC 구성 요소 (HandlerMapping, Controller, ViewResolver, View) 설정과 bean, aop 설정들을 한다.

# DispatcherServlet 설정과 ApplicationContext (3/3)

- Spring 설정파일 등록하기
  - <servlet>의 하위태그인 <init-param>에 contextConfigLocation 이름으로 등록
  - 경로는 Application Root부터 절대경로로 표시
  - 여러 개의 경우 , 또는 공백으로 구분

```
<servlet>
 <servlet-name>dispatcher</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
 <init-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>/WEB-INF/server-service.xml
 /WEB-INF/dao-service.xml
 </param-value>
 </init-param>
 <load-on-startup>1</load-on-startup>
</servlet>
```

# HandlerMapping

- Client요청을 처리할 Controller(Handler)를 찾아 DispatcherServlet에게 전달
- 클라이언트의 요청과 Controller 를 매칭하여 찾기 위한 다양한 HandlerMapping 제공됨
  - 기본 HandlerMapping : 따로 설정 하지 않아도 된다.
    - BeanNameUrlHandlerMapping
    - DefaultAnnotationHandlerMapping
  - 그 이외 HandlerMapping은 <bean>으로 등록한다.
  - 명시적으로 HandlerMapping 등록시 기본 HandlerMapping은 무시됨
  - HandelrMapping은 <bean> 으로 등록한다.



# HandlerMapping

- 주요 HandlerMapping
  - BeanNameUrlHandlerMapping
    - bean의 이름과 url을 mapping
    - Ant 패턴 문자를 이용해 패턴 매핑 가능
  - SimpleUrlHandlerMapping
    - url pattern들을 properties로 등록해 처리
  - DefaultAnnotationHandlerMapping
    - Annotation기반 Controller 처리

# HandlerMapping

- HandlerMapping 들 공통 설정 Property
  - order
    - 여러 개의 HandlerMapping 등록 시 적용되는 순서
  - defaultHandler
    - Client 요청 controller를 찾지 못했을 때 선택할 기본 controller 등록
    - 예

```
<bean class=".....BeanNameUrlHandlerMapping">
 <property name="defaultHandler" ref="defaultController"/>
</bean>
```

# HandlerMapping

## BeanNameUrlHandlerMapping 설정

```
<bean id= "handlerMapping"
class="org.springframework.web.servlet.Handler.BeanNameUrlHandlerMapping"/>
<bean name="/hello.do" class="controller.HelloController"/>
<bean name="/welcome.do" class="controller.WelcomeController"/>
```

## SimpleUrlHandlerMapping 설정

```
<bean id= "handlerMapping"
class="org.springframework.web.servlet.Handler.SimpleUrlHandlerMapping">
 <property name="mappings">
 <props>
 <prop key="/register.do">registerContoller</prop>
 <prop key="/delete.do">deleteController</prop>
 </props>
 </property>
</bean>
<!--컨트롤러 bean으로 등록-->
<bean name="registerContoller"/>
<bean name="deleteController"/>
```

# Controller 종류

- Simple Controller
  - Controller Interface 상속 받아 구현
  - SimpleControllerHandlerAdapter와 연동
  - 기본 구현에 따라 여러 종류가 있다.
    - AbstractController
    - MultiActionController
  - 구현 : 기본 구현 클래스들을 상속받아 역할에 따라 구현
- Annotation기반 Controller
  - 클래스나 메소드에 어노테이션 설정을 통해 작성
  - POJO 기반으로 작성한다.
  - AnnotationMethodHandlerAdapter와 연동

# Simple Controller - AbstractController (1/2)

- 가장 기본이 되는 Controller
- 작성
  - AbstractController 상속한다.
  - `public ModelAndView handleRequestInternal`  
`(HttpServletRequest request,`  
`HttpServletRequest response)`  
`throws Exception`  
오버라이딩 하여 코드 구현
  - ModelAndView에 view가 사용할 객체와 view에 대한 id값을 넣어 생성 후 return

# Simple Controller - AbstractController (2/2)

```
public class HelloworldAbstractController extends AbstractController{
 protected ModelAndView handleRequestInternal(
 HttpServletRequest request,
 HttpServletResponse response)
 throws Exception {
 //Model 호출 - Business Logic 처리
 //ModelAndView를 통해 view로 수행 넘김
 return new ModelAndView("hello","message", "안녕");
 }
}
```

# Simple Controller - MultiActionController (1/3)

- 하나의 Controller에서 여러 개의 요청 처리 지원
  - 연관된 request들을 처리하는 controller로직을 하나의 controller로 묶을 경우 사용.

- 작성

- MultiActionController 상속
- client의 요청을 처리할 메소드 구현

public [ModelAndView|Map|String|void] 메소드이름(  
HttpServletRequest req, HttpServletResponse res  
[HttpSession|Command]) [throws Exception]{}

- return type : ModelAndView, Map, void 중 하나
- argument :
  - 1번 - HttpServletRequest, 2번 - HttpServletResponse
  - 3번 - 선택적이며 HttpSession 또는 Command
  - or 3번 HttpSession, 4번 - Command

# Simple Controller - MultiActionController (2/3)

- MethodNameResolver 등록
  - 역할 : 어떤 메소드가 클라이언트의 요청을 처리할 것인지 결정
  - Spring 설정파일에 <bean>으로 등록
  - controller에서는 property로 주입 받는다.
  - 종류
    - ParameterMethodNameResolver : 요청 parameter로 메소드 이름 전송
    - InternalPathMethodNameResolver : url 마지막 경로 메소드 이름으로 사용
    - PropertiesMethodNameResolver : URL과 메소드 이름 mapping을 property로 설정



# Simple Controller - MultiActionController (3/3)

## Controller class

```
public class MemberController extends MultiActionController{

 public ModelAndView registerMember(HttpServletRequest request,
 HttpServletResponse response) throws Exception{
 //Business Logic 구현
 ModelAndView mv = new ModelAndView();
 mv.setViewName("register_ok");
 return new ModelAndView();
 }
}
```

```
<bean id="methodNameResolver"
 class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver">
 <property name="paramName" value="mode"></property>
</bean>

<bean name="/member.do" class="controller.multiaction.MemberController">
 <property name="methodNameResolver">
 <ref bean="methodNameResolver"/>
 </property>
</bean>
```

호출 : <http://ip:port/appName/member.do?mode=registerMember>

# Annotation기반 Controller

- @Controller
  - 컨트롤러 클래스 표시
- @RequestMapping
  - 처리할 요청 URL 등록
  - 처리할 요청방식 지정
- Controller 클래스 구현 후 스프링 설정파일에 등록
  1. <bean>을 이용해 등록
  2. 자동 스캔
    - <context:component-scan base-package= "*package*"/>

# Annotation기반 Controller - @RequestMapping

- 처리할 요청 URL 등록
  - 타입(클래스)의 어노테이션으로 등록
  - Handler 메소드에 등록
  - 타입과 Handler 메소드 양쪽에 등록하는 경우 둘을 조합
- 요청 방식 등록
  - 처리한 HTTP 요청 방식 지정
  - RequestMethod enum에 정의된 값을 이용

# Annotation기반 Controller - @RequestMapping

- 속성
- String [] value()
  - 매핑할 URL 패턴 지정
  - 배열이므로 여러 개 지정 가능
    - @RequestMapping("/main.do")
    - @RequestMapping({"main.do", "index.do"});
  - ANT 패턴을 이용해 URL 지정 가능
    - ? : 1개의 문자와 매칭
    - \* : 0개 이상의 문자와 매칭
    - \*\* : 0개 이상의 디렉터리와 매칭
    - @RequestMapping("/view/\*.do")
    - @RequestMapping("/member/\*\*/\*")
  - {} 를 이용해 @PathVariable 로 받을 경로 지정
    - @RequestMapping("/user/{userid}")

# Annotation기반 Controller - @RequestMapping

- 속성
  - RequestMethod[] method()
    - 처리할 HTTP 요청 방식 지정
  - RequestMethod enum 타입에 정의된 7개의 HTTP 요청 메소드 값을 이용해 지정
  - @RequestMapping(value="/user/add",  
method=RequestMethod.GET)

## Annotation기반 Controller – Controller 메소드 구현

- 메소드는 public 메소드로 구현
- 메소드 이름의 제한은 없다.(식별자 규칙안에서)
- 매개변수와 return type은 제한된 범위에서 마음대로 정의 할 수 있다.
  - 자신의 사용에 맞게 최적화 된 메소드 설계가 가능

# Annotation기반 Controller – Controller 메소드 구현

- 메소드 매개 변수(파라미터)
  - HttpServletRequest
  - HttpServletResponse
  - HttpSession
    - 기존 Session이 없으면 Session을 만들어 제공한다.
  - @PathVariable 어노테이션 지정한 변수
    - @RequestMapping의 URL에 {} 로 들어가는 Path variable(패스변수)에 넣을 값 받을 변수
    - 요청파라미터를 쿼리 스트링대신 URL 경로로 받는 경우 사용

요청 URL : **/user/view/10**

```
@RequestMapping("/user/view/{viewId}")
public String view(@PathVariable("viewId") int id){...}
```

요청 URL : **/member/id-1/order/20**

```
@RequestMapping("/member/{memberid}/order/{orderid}")
public String lookup(@PathVariable("memberid") String memId,
 @PathVariable("orderid") int orderId){...}
```

# Annotation기반 Controller – Controller 메소드 구현

- 메소드 매개 변수(파라미터)
  - @RequestBody 선언 변수
    - 요청정보를 문자열로 받을 변수
  - Map, Model, ModelMap
    - Model정보(View에게 전달할 값)을 담는 객체
    - Map – java.util.Map으로 put(key, value) 메소드로 값 설정
    - Model, ModelMap – org.springframework.ui
      - addAttribute(String key, Object value) : Model/ModelMap
      - addAllAttributes(java.util.Map<String, ?>) : Model/ModelMap
  - @CookieValue
    - HTTP 요청과 함께 전달된 Cookie 값을 넣을 변수 지정
    - 속성
      - String value() : 쿠키 명
      - boolean required() : 필수 여부. 안 넘어오면 500오류 발생.  
기본 : true
      - String defaultValue() : 값이 안 넘어 올 경우 설정할 기본 값

```
public String check(@CookieValue("c_id") String id){...}
```



# Annotation기반 Controller – Controller 메소드 구현

- 메소드 매개변수(파라미터)
  - @RequestParam 선언 변수
    - 요청 파라미터를 메소드의 매개변수에 넣기 위해
    - 요청 파라미터의 **name**과 변수 명이 같은 경우 생략가능
    - 같은 이름으로 여러 개 넘어올 경우 **String []** 로 지정
    - 요청파라미터 이름을 지정하지 않고 **Map** 타입 변수를 지정하면 요청파라미터들을 **name-value**로 **map**에 넣어줌
    - 속성
      - **String value()** : 요청파라미터 이름 설정
      - **boolean required()** : 필수 여부. 안 넘어오면 400오류 발생.  
기본 : true
      - **String defaultValue()** : 값이 안 넘어 올 경우 설정할 기본 값

요청 파라미터 : **id=id-1&password=1111**

```
public String login(@RequestParam("id") String id,
 @RequestParam("password") String password){...}
```

# Annotation기반 Controller – Controller 메소드 구현

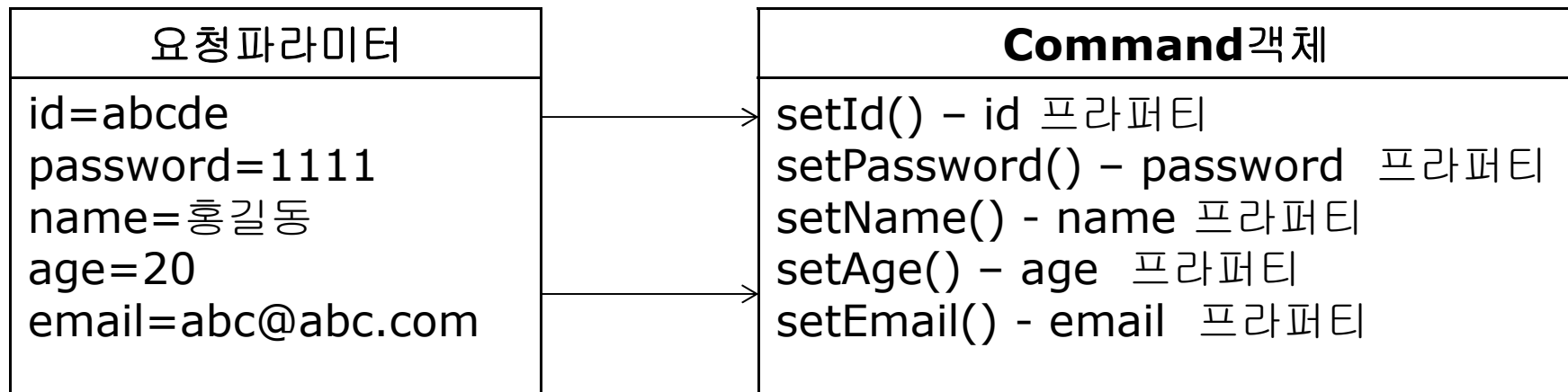
- 메소드 매개변수(파라미터)
  - @ModelAttribute 선언 변수
    - DTO를 이용해 여러 개의 요청파라미터 값을 받을 경우 사용
    - DTO의 프라퍼티 이름과 같은 요청파라미터의 이름의 요청파라미터를 프라퍼티의 값으로 설정해 준다.
      - @ModelAttribute로 등록한 객체를 Command 객체라 한다.
  - Command 객체는 Request Scope의 속성으로 등록됨
    - 속성명
      - » 기본 : Command 클래스(DTO) 의 이름 (첫글자 소문자로 변환)
      - » 설정 : @ModelAttribute("속성명")
- @ModelAttribute 생략 가능

# Annotation기반 Controller – Controller 메소드 구현

– @ModelAttribute - 예

```
id=abcde&password=1111&name=홍길동&age=20&email=abc@abc.com
```

```
@RequestMapping(value="join.do", method=RequestMethod.POST)
public String joinMember(@ModelAttribute("mto") MemberTO mto){
 ...
}
```



# Annotation기반 Controller – Controller 메소드 구현

- 메소드 리턴타입(Return Type)

- ModelAndView
- String
- void

- View이름으로 요청 URL에 사용한 mapping 정보 사용

<pre>@RequestMapping("/index") public void mainPage(){..}</pre>
View Name – index

- Model(Command)객체 타입

- Model로 사용할 DTO 객체를 return type으로 지정
  - View Name은 void 경우와 같다.
  - return 되는 객체는 클래스 이름을 Model 이름으로 해서 넘어간다.

<pre>@RequestMapping("/searchMember") public MemberTO searchMember(@){..}</pre>
View Name : searchMember Model : memberTO – MemberTO객체(return객체)

# Annotation기반 Controller – Controller 메소드 구현

- 메소드 리턴타입(Return Type)
  - Map, Model, ModelMap
    - View에게 전달할 Model을 리턴
    - View Name은 void 와 동일
  - View
    - View Name이 아니라 View객체를 직접 return

```
@RequestMapping("/index")
public View mainPage(){
 ...
 new InternalResourceView("/index.jsp");
}
```

- @ResponseBody 설정
  - 메소드 레벨의 어노테이션으로 return 값을 변환하여 응답처리

# ModelAndView (1/2)

- Controller 처리 결과 후 응답할 view와 view에 전달할 값을 저장.
- 생성자
  - ModelAndView(String viewName) : 응답할 view설정
  - ModelAndView(String viewName, Map values) : 응답할 view와 view로 전달할 값들을 저장 한 Map 객체
  - ModelAndView(String viewName, String name, Object value) : 응답할 view이름, view로 넘길 객체의 name-value
- 주요 메소드
  - setViewName(String view) : 응답할 view이름을 설정
  - addObject(String name, Object value) : view에 전달할 값을 설정 - requestScope에 설정됨
  - addAllObjects(Map values) : view에 전달할 값을 Map에 name-value로 저장하여 한번에 설정 - requestScope에 설정됨
- Redirect 방식 전송
  - view이름에 redirect: 접두어 붙인다.  
ex) mv.setViewName("redirect:/welcome.html");

## ModelAndView (2/2)

```
protected ModelAndView handleRequestInternal(HttpServletRequest req,
 HttpServletResponse res) throws Exception{
 //Business Logic 처리
 ModelAndView mv = new ModelAndView();
 mv.setViewName("/hello.jsp");
 mv.addObject("greeting", "hello world");
 return mv;
}
```

# ViewResolver (1/3)

- Controller가 넘긴 view이름(View Name)을 통해 알맞은 view를 찾는 전달하는 컴포넌트
  1. Controller는 응답을 처리할 view이름을 return.
  2. DispatcherServlet은 ViewResolver에게 받은 View Name을 전달하여 응답할 view를 요청한다.
  3. ViewResolver는 View 이름을 이용해 알맞는 view 객체를 찾아 DispatcherServlet에게 전달.
- 기본 ViewResovler
  - InternalResourceViewResolver
  - 여러 개 등록 가능하며 order 프라퍼티로 순서지정
  - 등록 시 Spring 설정파일에 <bean>으로 등록한다.



## ViewResolver (2/3)

- InternalResourceViewResolver
  - JSP나 HTML등의 내부 자원을 이용해 뷰 생성
  - InternalResourceView를 기본 뷰로 사용
- BeanNameViewResolver
  - 뷰의 이름과 동일한 이름을 가지는 빈을 View로 사용
  - 사용자 정의 View 객체를 사용하는 경우 주로 사용
- XmlViewResolver
  - BeanNameViewResolver와 동일 하나 view와 view name간의 매핑을 외부 Xml 파일에 설정

# ViewResolver (3/3)

## **Spring** 설정파일에 설정

```
<bean id="viewResolver"
 class="org.springframework.web.servlet.view.InternalResourceViewResolver">
 <property name="prefix" value="/WEB-INF/jsp/">
 <property name="suffix" value=".jsp"/>
</bean>
```

## **Controller**

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

위의 경우

/WEB-INF/jsp/hello.jsp 를 찾는다.

# View

- 응답을 처리하는 역할
- View interface를 implements 해 구현
  - 주로 `AbstractView`를 extends 해서 구현
  - overriding 메소드
    - `getContentType() : String`
      - View가 응답할 content의 타입을 String 으로 return
    - `renderMergedOutputModel(Map model, request, response)`
      - 응답 처리 메소드

# View

- 주요 View
  - InternalResourceView
    - RequestDispatcher의 forward(), include() 이용하는 View.
    - 주로 JSP로 이동할 때 사용
    - 직접 생성보다 InternalResourceViewResolver를 이용
  - RedirectView
    - HttpServletResponse의 sendRedirect() 를 통해 리다이렉트 방식으로 응답시 사용
    - Model 정보는 Query String으로 전달 된다.
    - 객체를 직접 생성 – `new RedirectView("/res.jsp");`
    - View Name으로 요청 – “redirect:” prefix 사용
      - ex) `new ModelAndView("redirect:res.jsp");`
  - MappingJacksonJsonView
    - Model을 JSON으로 변환해 응답
    - 응답 content type : application/json

# Spring MVC

파일 업로드 처리하기

# FileUpload - 파일 업로드 요청 페이지

- 호출 JSP(또는 HTML)
  - 요청 방식 : post
  - <form enctype="multipart/form-data">
  - input tag : <input type="file" name="upfile"/>
    - name 속성의 값은 upload정보를 저장할 TO(VO)의 Attribute와 매칭 된다.
    - 여러 개의 파일을 업로드 할 때 name속성의 값은 이름[0], 이름[1] 형식으로 작성
      - <input type="file" name="upfile[0]"/>
      - <input type="file" name="upfile[1]"/>

# FileUpload - Spring 설정파일

- multipartResolver 빈으로 등록
    - upload를 처리해 주는 bean
    - id/name은 반드시 multipartResolver 로 등록
- ```
<bean id="multipartResolver"
      class="org.springframework.web.
             multipart.common.
             CommonsMultipartResolver"/>
```
- Property
 - defaultEncoding – 기본 인코딩 설정
 - maxUploadSize – 업로드 허용 최대 size를 byte단위로 지정.
-1은 무제한
 - uploadTempDir – 업로드 파일일이 저장될 임시 경로 지정
 - maxInMemorySize – 업로드 파일을 저장할 최대 메모리 크기

FileUpload – Controller에서 처리

- Transfer Object를 통해 받기
 - 파일 요청 파라미터의 이름과 매칭되는 property작성
 - 파일의 정보를 저장할 property는 MultipartFile 타입으로 작성
- @RequestParam 을 통해 받기
 - Controller 메소드의 MultipartFile 타입의 매개변수 사용
- MultipartHttpServletRequest 이용
 - Controller 메소드의 매개변수로 MultipartHttpServletRequest를 선언
 - 주요 메소드
 - getFileNames() : Iterator<String>-업로드된 파일명들 조회
 - getFile(String name) : MultipartFile-업로드된 파일정보 조회
 - getFiles(String name):List<MultipartFile>-업로드된 파일정보들 조회

FileUpload - MultipartFile

- `org.springframework.web.multipart.MultipartFile`
 - 업로드된 파일정보를 저장하는 객체
 - `getName() : String` – 요청파라미터의 name
 - `getOriginalFilename() : String` – upload된 파일명
 - `getSize() : long` – 파일의 크기
 - `transferTo(File dest)` – upload된 파일을 특정 경로로 이동
 - `isEmpty() : boolean` – upload된 파일이 없으면 true
 - `getInputStream() : InputStream` – 업로드된 파일과 연결된 InputStream 리턴