

Spring framework

Spring 이란?



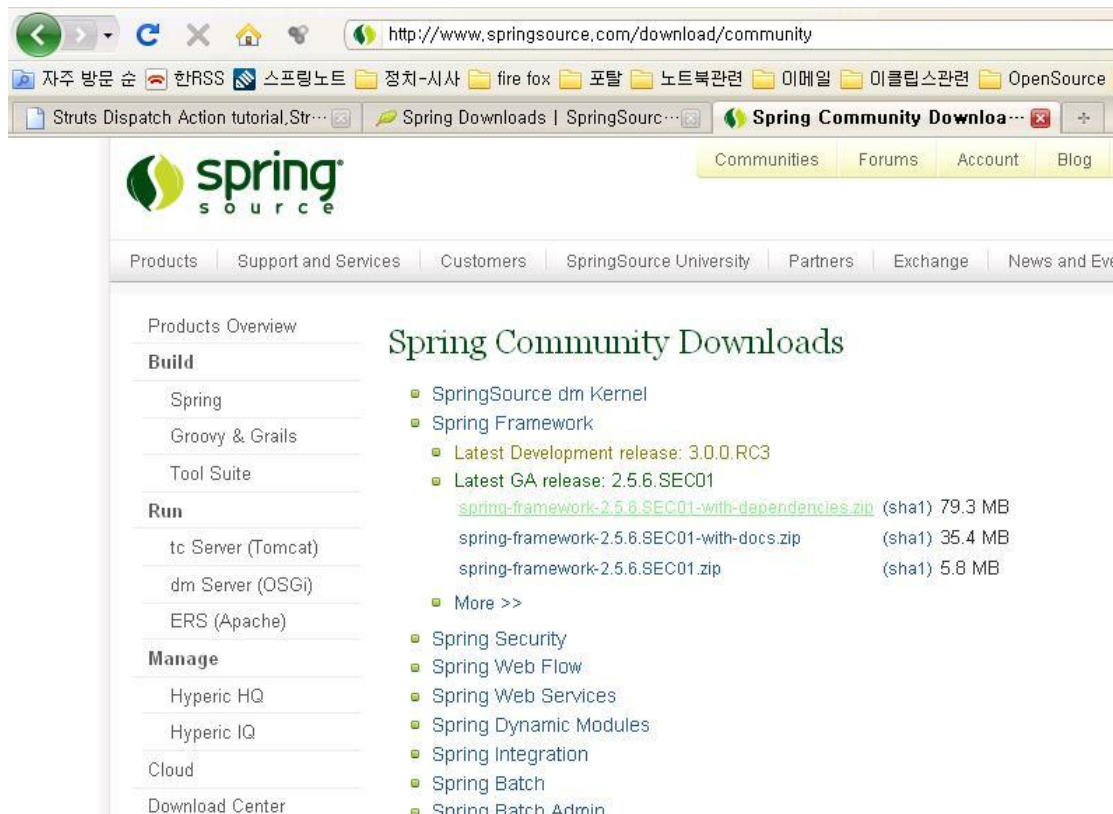
- 오픈 소스 프레임워크
 - Rod Johnson 창시
 - Expert one-on-one J2EE Design - Development, 2002, Wrox
 - Expert one-on-one J2EE Development without EJB, 2004, Wrox
 - 엔터프라이즈 어플리케이션 개발의 복잡성을 줄여주기 위한 목적
 - EJB 사용으로 수행되었던 모든 기능을 일반 POJO(Plain Old Java Object) 를 사용해서 가능하게 함.
 - 경량 컨테이너(light weight container)
 - www.springframework.org
- 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 checked="" type="checkbox"/> Core / Spring IDE	
<input checked="" type="checkbox"/> Extensions (Incubation) / Spring IDE	
<input checked="" type="checkbox"/> Extensions / Spring IDE	
<input checked="" type="checkbox"/> Integrations / Spring IDE	
<input checked="" type="checkbox"/> Resources / Spring IDE	

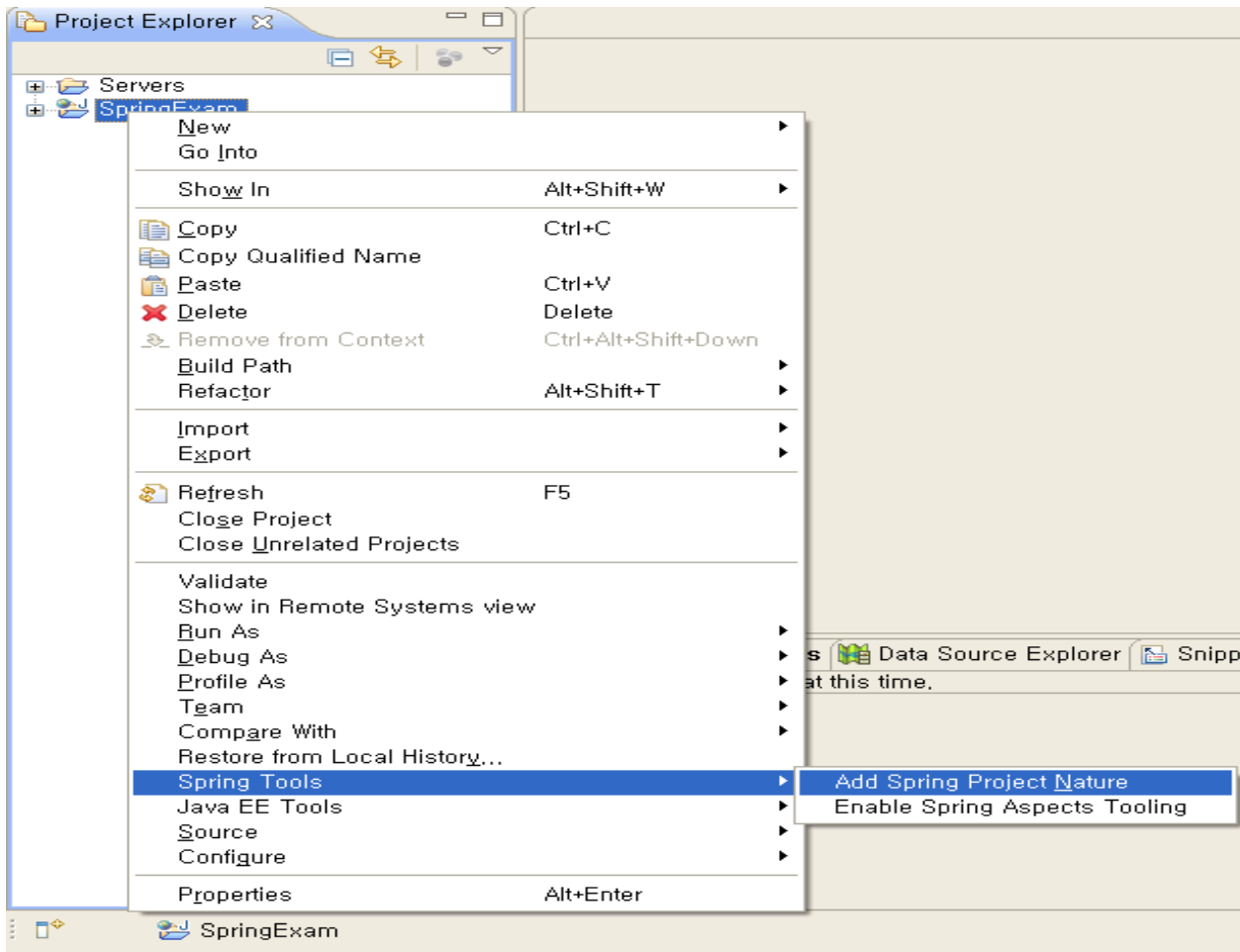
Details

☒ Show only the latest versions of available software
☒ Group items by category
☒ Contact all update sites during install to find required software

☐ Hide items that are already installed
What is [already installed](#)?

Spring IDE 이클립스 Plugin 설정 (2/2)

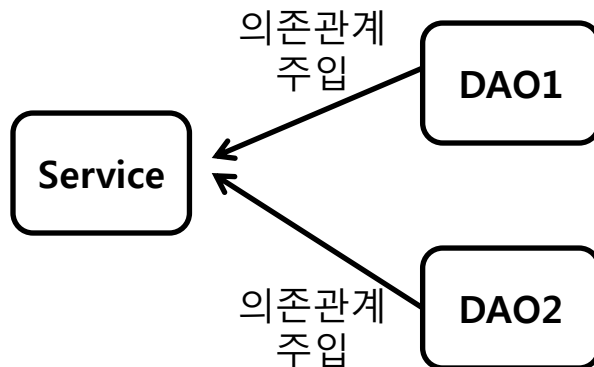
- Project 생성
- 오른 마우스 메뉴-Spring Tools-Add Spring Project Nature 선택



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 짐.
 - 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>
```

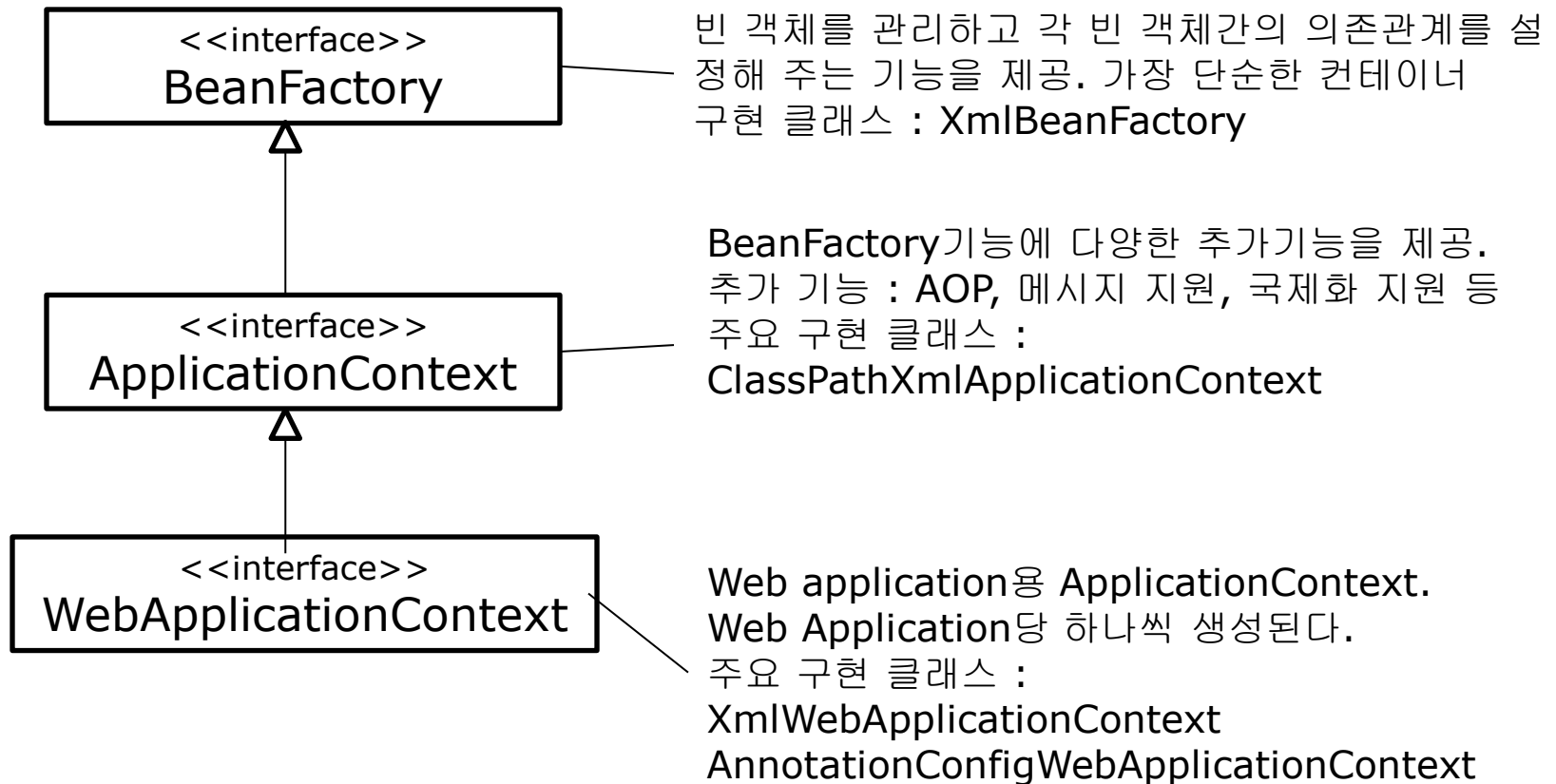
Bean객체 주입 받기 - 설정 Bean 사용(2/2)

- 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)

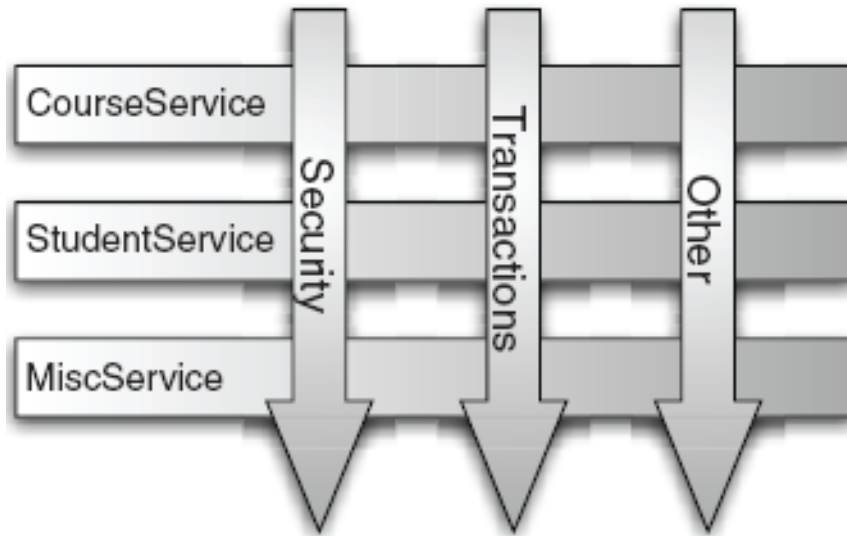


Figure 4.1
Aspects modularize cross-cutting concerns, applying logic that spans multiple application objects.

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){  
  
}
```

A diagram with two arrows. One arrow starts from the attribute **returning** in the XML tag and points to the parameter **retValue** in the Java method signature. The other arrow starts from the attribute **method** in the XML tag and points to the method name **returnLogging** in the Java signature.

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 배열개체가 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.

Web에서 Spring

– ContextLoaderListener

- 설정파일 <context-param>으로 등록된 설정파일을 바탕으로
WebApplicationContext객체를 생성해 ApplicationScope에 속성으로
binding

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

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


Web에서 Spring

- WebApplicationContextUtil 를 통해 WebApplicationContext 조회
 - getWebApplicationContext(ServletContext) : WebApplicationContext

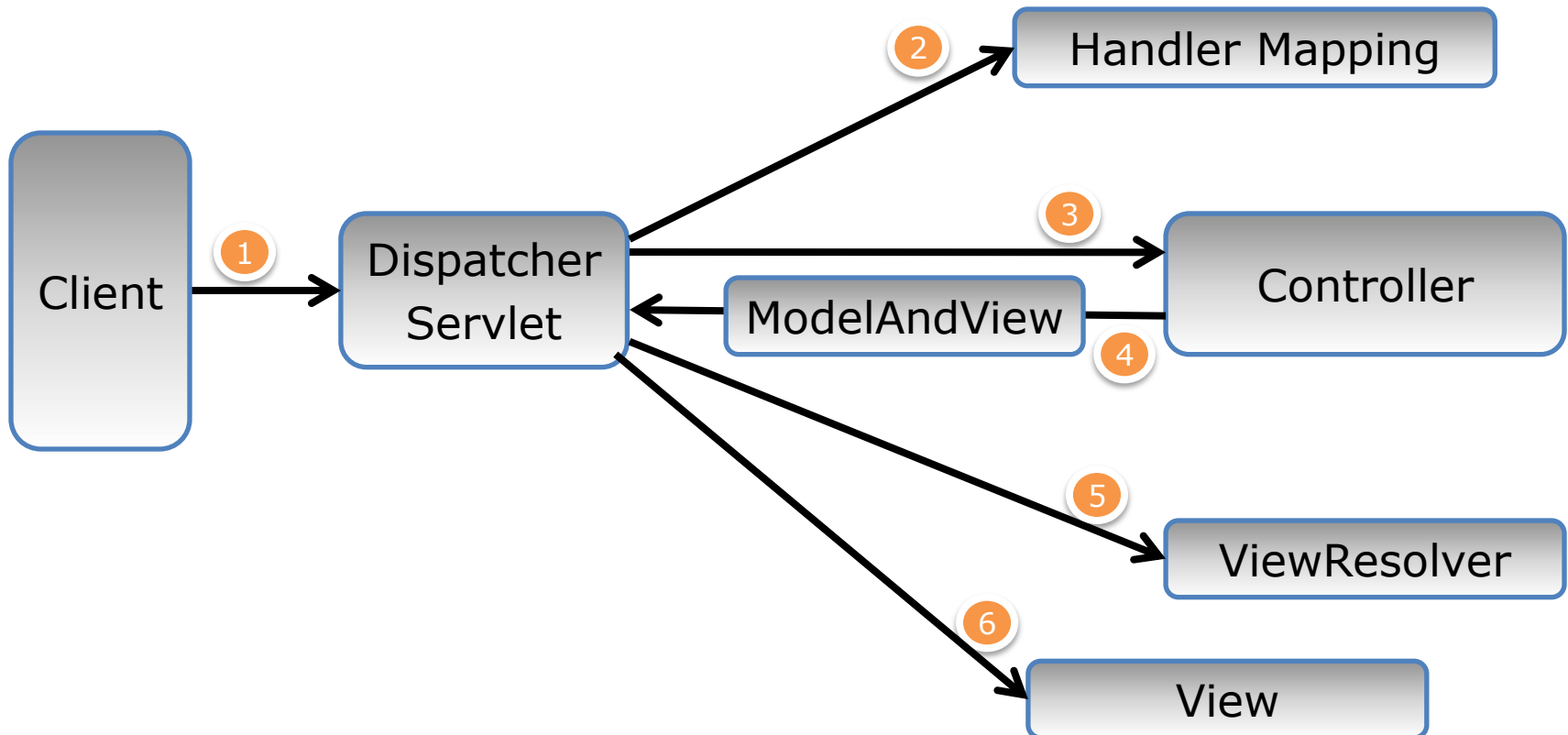
Spring MVC

Spring MVC 구성 주요 컴포넌트

- DispatcherServlet
 - Front Controller
- Controller
 - 클라이언트 요청 처리를 수행하는 Controller.
- HandlerMapping
 - 클라이언트의 요청을 처리할 Controller를 찾는 작업 처리
- View
 - 응답하는 로직을 처리
- ViewResolver
 - 응답할 View를 찾는 작업을 처리
- ModelAndView
 - 응답할 View와 View에게 전달할 값을 저장하는 용도의 객체

Spring MVC 흐름 (1/2)

- Spring MVC
 - MVC 패턴 기반 웹 개발 프레임워크



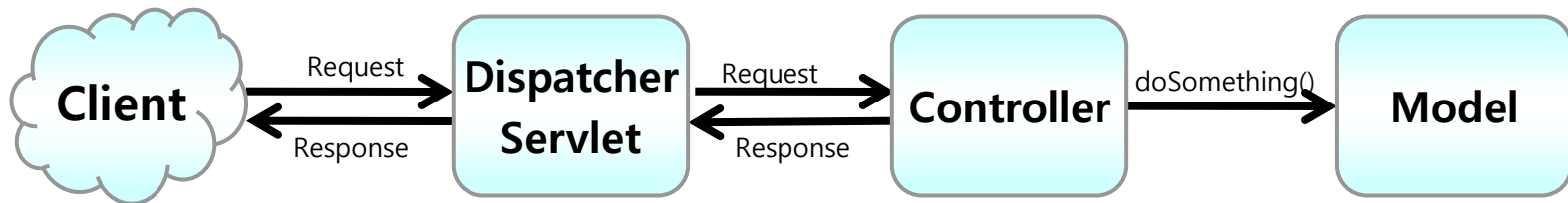
Spring MVC 흐름 (2/2)

- 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 컨트롤러에게 전송하고 컨트롤러는 요청을 처리한 후 결과 리턴
 - 비즈니스 로직 수행 후 결과 정보(Model)가 생성되어 JSP와 같은 뷰에서 사용됨
- ④ ModelAndView를 생성하여 DispatcherServlet에 리턴
- ⑤ ModelAndView 정보를 바탕으로 바탕으로 ViewResolver에게 View를 요청
- ⑥ View는 결과정보를 사용하여 화면을 표현함.

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. 컨트롤러와 JSP의 연결 위해 View Resolver Spring 설정 파일에 등록
 5. JSP(or View)작성 후 설정) 코드 작성



DispatcherServlet 설정과 ApplicationContext (1/2)

- 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 Container**는 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 (2/2)

- 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 Controller를 찾는 역할
- 다양한 HandlerMapping 클래스를 Springframework가 제공 하며 Spring 설정파일에 <bean> 으로 등록하여 설정한다.
- 종류
 - BeanNameUrlHandlerMapping
 - bean의 이름과 url을 mapping
 - SimpleUrlHandlerMapping
 - url pattern들을 properties로 등록해 처리
 - DefaultAnnotationHandlerMapping
 - Annotation기반 Controller 처리

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 작성

- Controller 종류
 - Controller (interface)
 - AbstractController
 - MultiActionController
 - Annotation기반 Controller
- 위의 interface/class를 상속하여 Controller 작성한다.(Annotation기반 Controller제외)

AbstractController (1/2)

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

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");  
    }  
}
```

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

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

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/applName/member.do?mode=registerMember>

Annotation기반 Controller

- @Controller
 - 컨트롤러 클래스 표시
- @RequestMapping
 - 요청 URL 등록, 처리할 요청방식 지정
 - 구문
 - RequestMapping("요청 url")
 - RequestMapping(value="요청URL" method=요청방식)
 - Controller 클래스에 등록
 - Controller 메소드에 등록
- Controller 클래스 스프링 설정파일에 등록
 1. <bean>을 이용해 등록
 2. 자동 스캔
 - <context:component-scan base-package="package"/>

Annotation기반 Controller

- Controller 메소드에서 요청파라미터 처리
 - Transfer Object(Command) 이용
 - 요청파라미터와 매칭되는 이름의 property를 가진 TO
 - 요청파라미터 name을 이용한 매개 변수 사용
 - 같은 이름으로 여러 개 값이 넘어올 경우 String[] 사용
 - @RequestParam Annotation 사용
 - 속성
 - value : 요청파라미터 이름 설정
 - required : 필수 여부. 안넘어오면 400오류 발생. 기본 : false
 - defaultValue : 값이 안넘어 올 경우 설정할 기본 값

Annotation기반 Controller

- Controller 메소드 이용가능 매개변수 타입
 - HttpServletRequest
 - HttpServletResponse
 - HttpSession
 - 요청파라미터 연결 변수
 - 요청파라미터를 설정할 Transfer Object 객체
 - @CookieValue 적용 매개변수 – 쿠키 값 매핑
 - @CookieValue(value="name", required=false)
 - Map, Model, ModelMap
 - View에 전달할 모델 데이터 설정시 사용

Annotation기반 Controller

- Controller 메소드 설정 가능 return type
 - **ModelAndView** : View 정보와 응답 데이터 설정
 - View에 전달할 값 설정
 - Map
 - Model
 - View는 요청 URL로 결정됨
 - **String** : View의 이름 리턴
 - View 객체
 - void
 - Controller 메소드 내에서 응답을 직접처리 시 사용

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 view이름을 통해 알맞은 view를 찾는 역할
 1. Controller는 ModelAndView 객체에 응답할 view이름을 넣어 return.
 2. DispatcherServlet은 ViewResolver에게 응답할 view를 요청한다.
 3. ViewResolver는 View 이름을 이용해 알맞는 view 객체를 찾아 DispatcherServlet에게 전달.
- ViewResolver – Spring 설정파일에 등록한다.

ViewResolver (2/3)

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.

ViewResolver (3/3)

- 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 뷰객체를 Xml 파일에 설정해 놓는 것이 차이.
 - Bean 등록시 location 프라퍼티에 xml 파일을 지정

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

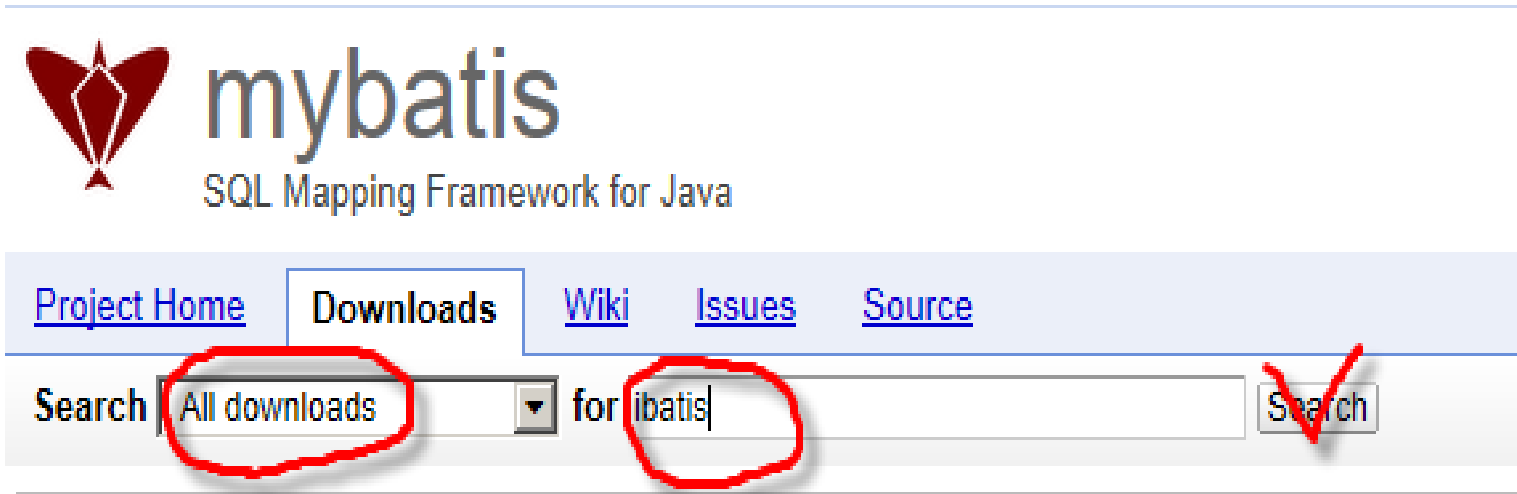
# **iBATIS**

# 개요

- JDBC 코드의 패턴
  - Connection -> Statement -> 쿼리전송->연결 close
  - 모든 JDBC 코드는 위의 패턴을 가진다.
  - 이 패턴을 캡슐화 하여 JDBC 코드를 간편하게 사용할 수 있도록 Framework화 가능
- iBATIS 란
  - SQL 실행 결과를 자바빈즈 혹은 Map 객체에 매핑해 주는 Persistence 솔루션으로 SQL을 소스코드가 아닌 XML로 따로 분리해 관리하도록 지원
- 장점
  - SQL 문장과 프로그래밍 코드의 분리
  - JDBC 라이브러리를 통해 매개변수를 전달하고 결과를 추출하는 일을 간단히 처리가능
  - 자주 쓰이는 데이터를 변경되지 않는 동안에 임시 보관 (Cache) 가능
  - 트랜잭션처리 제공

# iBatis 설치

- <http://blog.mybatis.org> 에서 받는다.
- iBatis2.X.X 다운로드





# iBATIS 설치

- 압축을 풀면 lib 디렉토리에 api가 있다.
- API를 Application에 설치
  - Standalone Application
    - 어플리케이션 start 스크립트에 클래스 패스 정의
    - `java -cp ibatis-2.3.0.677.jar:... MyMainClass`
  - Web Application
    - WEB-INF/lib에 추가
    - 주) 다른 경로에 추가하면 설정파일을 찾지 못하는 문제가 발생할 수 있다.

# iBATIS Quick Start (1/3)

- iBATIS SqlMap Config(sqlmap-config.xml) 작성 – iBATIS framework에 대한 설정

```
<?xml version="1.0" encoding="EUC-KR"?>
<!DOCTYPE sqlMapConfig
 PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
 "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
 <transactionManager type="JDBC">
 <dataSource type="SIMPLE">
 <property name="JDBC.Driver"
 value="oracle.jdbc.driver.OracleDriver" />
 <property name="JDBC.ConnectionURL"
 value="jdbc:oracle:thin:@127.0.0.1:1521:XE" />
 <property name="JDBC.Username"
 value="hr" />
 <property name="JDBC.Password"
 value="hr" />
 </dataSource>
 </transactionManager>

 <sqlMap resource="User.xml" />
</sqlMapConfig>
```

# iBATIS Quick Start (2/3)

- SqlMap 파일 작성(User.xml) – SQL 문을 등록하는 설정 파일

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
 PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN" "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="User">
 <typeAlias alias="User" type="myibatis.domain.User"/>

 <resultMap id="userMap" class="User">
 <result property="userId" column="userId"/>
 <result property="password" column="password"/>
 <result property="name" column="name"/>
 <result property="email" column="email"/>
 <result property="userType" column="userType"/>
 </resultMap>

 <select id="selectAllUsers" resultMap="userMap">
 select
 USER_ID as userId,
 PASSWORD as password,
 NAME as name,
 EMAIL as email
 from USER
 order by USER_ID ASC
 </select>
</sqlMap>
```

# iBATIS Quick Start (3/3)

- iBATIS Data Access Object 작성
  - UserIBatisDao.java

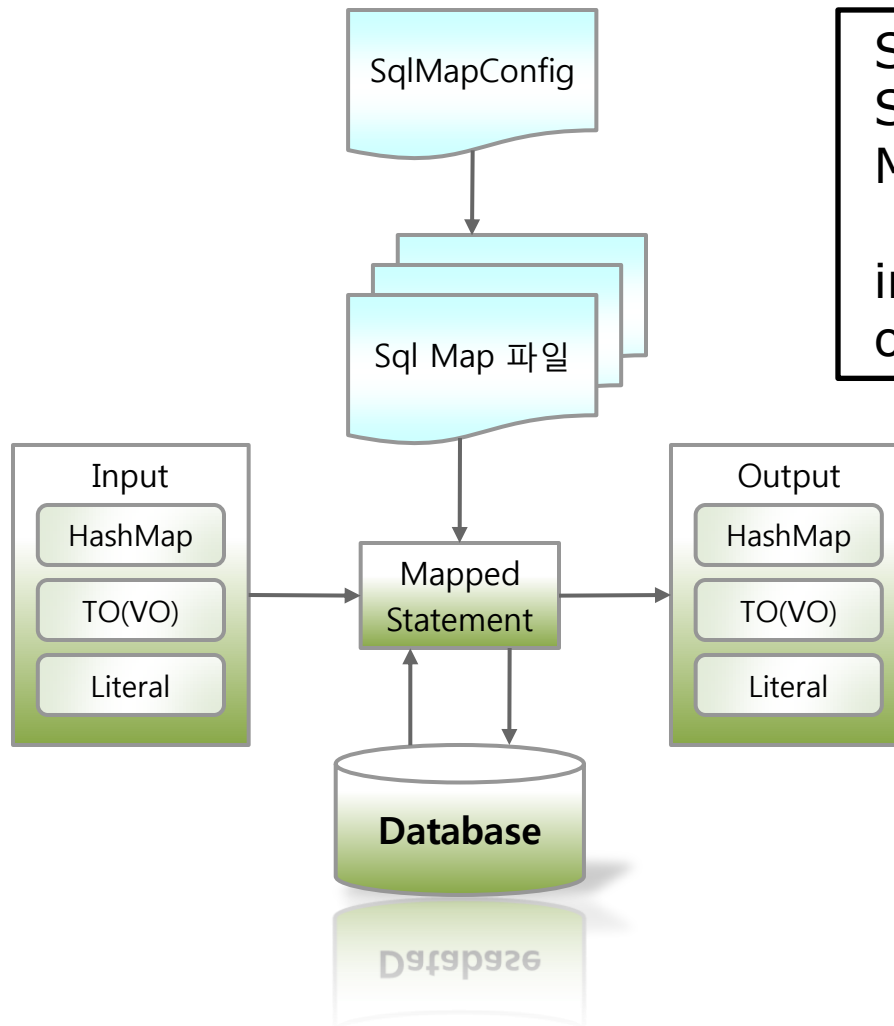
```
package myibatis.dao
import java.io.IOException;
import java.io.Reader;
import java.sql.SQLException;
import java.util.List;
import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;
import myibatis.domain.User;

public class UserIBatisDao {
 private static final String resource = "sqlmap-config.xml";
 private SqlMapClient client;

 public UserIBatisDao() {
 try {
 Reader reader = Resources.getResourceAsReader(resource);
 this.client = SqlMapClientBuilder.buildSqlMapClient(reader);
 } catch (IOException e) {
 throw new RuntimeException("SqlMapClient 생성중 오류발생", e);
 }
 }

 public List<User> findAllUsers() {
 try {
 List<User> users = client.queryForList("selectAllUsers", null);
 return users;
 } catch (SQLException e) {
 throw new RuntimeException("사용자 조회중 오류 발생", e);
 }
 }
}
```

# iBATIS 실행 흐름도



SqlMapConfig : 전역정보설정파일  
SqlMaps : SQL문 관련 설정 파일  
MappedStatement : 설정된 쿼리를  
실행하는 iBATIS 객체  
input : SQL문 실행 시 필요한 값  
output : select 실행 결과

# iBATIS 설정 파일

- 설정파일은 XML기반으로 작성
- SqlMapConfig
  - 전역 설정 위한 파일 : iBATIS Framework가 실행되는 데 필요한 여러 설정들을 한다.
  - Transaction 관리 정보, DataSource 생성을 위한 설정 정보, SqlMap 파일의 위치 등
- SqlMap
  - SQL문을 등록
  - SQL문을 실행하기 위해 필요한 input Data와 Output Data에 대한 설정을 한다.
- 설정 파일들은 작성 후 classpath내 저장한다.

# SqlMapConfig 설정 (1/4)

- **SQLMapConfig** 설정 파일  
(SqlMapConfig.xml)

전역 설정 옵션

```
<?xml version="1.0" encoding="EUC-KR"?>
<!DOCTYPE sqlMapConfig
 PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
 "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<sqlMapConfig>

 <settings useStatementNamespaces="false" cacheModelsEnabled="true"/>
```

트랜잭션 관리

```
 <transactionManager type="JDBC">
```

DataSource 설정

```
 <dataSource type="SIMPLE">
```

```
 <property name="JDBC.Driver" value=" " DriverClassName" />
```

```
 <property name="JDBC.ConnectionURL" value="Url" />
```

```
 <property name="JDBC.Username" value="User" />
```

```
 <property name="JDBC.Password" value="Password" />
```

```
 </dataSource>
```

```
 </transactionManager>
```

SqlMap 파일 참조

```
 <sqlMap resource="sample/config/sqlMap.xml" />
</sqlMapConfig>
```

## SqlMapConfig 설정(2/4)

- **<typeAlias>** 요소
    - 설정파일에서 사용할 클래스의 별칭(alias) 설정
    - SqlMap에서도 설정 가능
    - iBATIS는 정의된 alias로 언제든지 원래 type에 접근 가능
    - SqlMapConfig에 설정하면 모든 SqlMap에서 사용가능
- ```
<typeAlias alias="mto" type="member.to.MemberTO"/>
```
- Framework에서 미리 정의하여 제공하는 typeAlias
 - Transaction manager : JDBC, JTA, EXTERNAL
 - Data types : string, int, long, double, boolean, hashmap, arraylist, object 등
 - Data source factory : SIMPLE, DBCP, JNDI

SqlMapConfig 설정 (3/4)

- <transactionManager> 요소
 - Transaction Manager 타입 설정
 - JDBC : 단순 JDBC 기반의 Transaction Manager를 제공함
 - JTA : application이 동작하는 컨테이너 기반의 Transaction Manager를 제공함
 - EXTERNAL : 트랜잭션 관리를 iBATIS에서 하지 않음
 - <dataSource> : iBATIS에서 사용할 DataSource를 생성하는 DataSource Factory 지정
 - SIMPLE : iBATIS 자체 제공 하는 DataSourceFactory 사용
 - DBCP : Jakarta Commons Database Connection Pool 구현함
 - JNDI : Naming서버에 등록된 DataSource를 사용함
보통 Container가 제공하는 것을 사용
 - 하위 태그를 이용하여 필요한 property들을 설정한다.
 - driver, url, 계정정보 등

SqlMapConfig 설정 (4/4)

- <sqlMap> 요소
 - SQL문을 가지고 있는 설정파일인 SQL Map파일의 위치 지정
 - resource 속성 : class path 상의 SQL Map 파일
 - 여러 파일 지정 가능

iBATIS를 통한 SQL문 실행

- iBATIS의 JavaBean
- select
- insert
- update
- delete
- Parameter 매핑
- Result 매핑

JavaBeans 기초

- iBATIS의 Java property 접근

- Property 구성

- private instance variable과 public setter, getter 메소드

```
private String name;  
public String getName();  
public void setName(String name);
```

- property 타입이 boolean일 경우

```
private boolean isStudent;  
public boolean isStudent();  
public void setStudent(boolean isStudent);
```

- Beans 탐색

Java code	Dot 표기법
anOrder.getAccount().getUserName()	anOrder.account.userName
anOrder.getOrderItem().get(0).getProductId()	anOrder.orderItem[0].productId
anObject.getID()	anObject.ID
anObject.getNAME()	anObject.NAME

iBATIS를 통한 SQL 실행 - 개요

- Sql Map – SQL문 설정, Input Data, Output Data를 설정하는 xml 기반 설정파일
 - Sql Map 파일은 SqlMapConfig에 등록한다.
- SqlMapClient interface : SQL문 실행 메소드를 정의
 - package : com.ibatis.sqlmap.client
 - SqlMapClientBuilder를 통해 얻어온다.

```
//iBATIS Framework가 실행할때 필요한 SqlMapConfig
Reader reader = Resources.getResourceAsReader("config/SqlMapConfig.xml");
//SqlMapConfig의 내용을 적용하여 실행할 SqlMapClient 생성
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
```

- Sql Map에 SQL 문을 등록하고 프로그램에서는 SqlMapClient의 메소드를 이용해 등록된 쿼리를 실행 시킨다.

Sql Map (1/3)

- 주요 태그
 - SQL 문 등록 태그

태그명	속성	하위 요소	용도
<select>	id, parameterClass, parameterMap, resultClass, resultMap, cacheModel	모든 dynamic 요소	조회
<insert>	id, parameterClass, parameterMap	모든 dynamic 요소, selectKey	입력
<update>	id, parameterClass, parameterMap	모든 dynamic 요소	수정
<delete>	id, parameterClass, parameterMap	모든 dynamic 요소	삭제

– 외부 Parameter, Result 매핑 설정

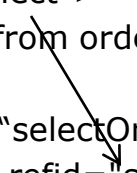
태그명	속성	하위 요소	용도
<parameterMap>	id, class	<parameter>	외부 Parameter Map 설정의 Root 태그
<parameter>	property, javaType, jdbcType, nullValue	N/A	하나의 Parameter Map 설정
<resultMap>	id, class	<result>	외부 Result Map 설정
<result>	property, column, jdbcType, javaType, nullValue	N/A	하나의 Result Map 설정

Sql Map (2/3)

- 주요 태그
 - SQL 문 생성 태그

태그명	속성	하위 요소	용도
<sql>	id	모든 dynamic 요소	재사용가능한 SQL문 등록
<include>	refid	모든 dynamic 요소	SQL등록 태그에서 sql에 등록한 태그를 재사용

```
<sql id="select">
  select * from order
</sql>
<select id="selectOrderByid" resultClass="map">
  <include refid="select" />
  where order_id = #orderId#
</select>
```



Sql Map (3/3)

member.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="member">
  <typeAlias type="member.to.MemberTO" alias="mto">
    <insert id="insertMember" parameterClass="mto">
      INSERT INTO MEMBER (ID, PASSWORD, NAME, ADDRESS, MILEAGE)
      VALUES(#id#, #password#, #name#, #address#, #mileage#)
    </insert>
    <update id="updateMember" parameterClass="mto">
      UPDATE MEMBER
      SET   PASSWORD=#password#, NAME=#name#,
          ADDRESS=#address#, MILEAGE=#mileage#
      WHERE ID=#id#
    </update>
    <delete id="deleteMember">
      DELETE FROM MEMBER WHERE ID=#value#
    </delete>
    <select id="getMemberByID" resultClass="mto">
      SELECT ID, PASSWORD, NAME, ADDRESS, MILEAGE FROM MEMBER
      WHERE ID=#value#
    </select>
  </sqlMap>
```


Parameter mapping (1/3)

- Sql Map에 쿼리 등록시 세가지 요소 설정 필요
 - SQL 문
 - Parameter : Input Data
 - Select문의 경우 Result : Output Data
- Parameter Mapping
 - Parameter : SQL 문에 넣을 값
 - 인 라인 Parameter , 외부 Parameter 매핑 두 가지 방식이 있다.
 - 인 라인 - 값과 값을 넣을 위치를 SQL 안에 설정
 - 외부 parameter mapping - SQL 문에는 ? 로 설정하고 넣을 parameter 연결은 SQL문 밖에서 설정
 - SqlMapClient의 메소드를 통해 받은 값과 SQL 문에 값이 들어갈 곳을 Mapping 한다.
 - SQL문 내 mapping 방법
 - #parameter name# } 인라인 파라미터
 - \$parameter name\$ }
 - parameter name
 - » Transfer Object(TO, VO): getter와 matching
 - » Map : key와 matching
 - ? - 외부 파라미터 매핑

Parameter mapping (2/3)

- '#' 지시자로 인라인 parameter 사용
 - ?로 바꾼 뒤 값을 치환한다. - PreparedStatement 형식
 - 타입에 맞게 치환 처리 (String의 경우 '' 처리)
 - 값만 치환 가능

```
<select id="selectUser" resultMap="userMap">
  select
    USER_ID as userId,
    PASSWORD as password,
    NAME as name,
    EMPLOYEE_NO as employeeNo,
    EMAIL as email,
    PHONE_NUM as phoneNumber,
    ZIP_CODE as zipCode,
    ADDRESS,
    GRADE as grade
  from USER
  where USER_ID = #value#
</select>
```

...

where USER_ID = 'abc';

```
User user = (User)client.queryForObject("selectUser", "abc");
```

Parameter mapping (3/3)

- '\$' 지시자로 인라인 parameter 사용
 - 전달받은 값을 바로 치환한다. - copy & paste 개념
 - String일 경우 ``로 감싸 주어야 한다.
 - 값 뿐만 아니라 키워드등 쿼리문 무엇이든 치환 가능

```
<select id="selectUser" resultMap="userMap">
  select
    USER_ID as userId,
    PASSWORD as password,
    NAME as name,
    EMPLOYEE_NO as employeeNo,
    EMAIL as email,
    PHONE_NUM as phoneNumber,
    ZIP_CODE as zipCode,
    ADDRESS,
    GRADE as grade
  from USER
  where USER_NAME LIKE '%$value$%'
</select>
```

```
...
where user.NAME like '%송%';
```

```
List<User> allUsers = client.queryForList("selectAllUsers", "송");
```

Parameter 매핑 – 외부 매핑 (1/2)

- Parameter 매핑의 두가지 방안
 - 인라인 매핑
 - SqlMap Mapping문 내부에서 바로 기술함
 - 매핑이 복잡한 경우 명시적이지 못함
 - 외부 매핑
 - 외부에서 정의됨
 - 보다 명시적임
 - 인라인 매핑과 외부매핑은 같이 사용할 수 없다.
 - ? 와 매칭되어 값이 할당된다.

인라인 매핑의 예

```
<insert id="createCategory" parameterClass="deptTO">
  insert into departments values(
    #deptId#, #deptName#,
    #managerId#,#loc.locationId#
  )
</insert>
```

외부 매핑의 예

```
<parameterMap class="deptTO" id="deptParamMap">
  <parameter property="deptId"/>
  <parameter property="deptName"/>
  <parameter property="managerId"/>
  <parameter property="loc.locationId"/>
</parameterMap>
<insert id="insert" parameterMap="deptParamMap" >
  INSERT INTO DEPARTMENTS VALUES(?, ?, ?, ?)
</insert>
```

Parameter 매핑 – 외부 매핑 (2/2)

- 외부 Parameter 매핑

속성	설명
property	매핑문에 전달할 JavaBean property 명칭 또는 Map value의 key. 들어갈 이름을 매핑문에 필요한 만큼 반복해서 정의할 수 있음. 예) update 문에서 set 절과 where절에 동일한 이름이 반복적으로 들어갈 수 있음.
javaType	세팅될 parameter의 Java property 타입을 명시적으로 정의
jdbcType	세팅될 parameter의 데이터베이스 타입을 명시적으로 정의 예) Java 타입이 Date(java.util.Date)인 경우 jdbcType을 DATE인지 DATETIME인지 명시할 필요가 있음
nullValue	nullValue에 명시된 값이 JavaBean property에서 넘어오면 null 값을 넣어준다.

- 외부 Parameter 매핑이 유용한 경우

- 인라인 Parameter 매핑이 잘 동작하지 않을 경우
- 성능을 향상시킬 경우
- 명시적 매핑을 할 경우

Result 매핑 (1/3)

- 명시적 Result 매핑
 - <resultMap> : 하나의 resultClass에 대한 설정
 - 속성 : id, class
 - <result> : 하나의 property와 resultSet의 Column 매핑

result 태그	
속성	설명
property	결과 객체의 JavaBean property 또는 Map 내용
column	JDBC ResultSet의 column
columnIndex	ResultSet의 column명 대신 index를 주어 결과 값 매핑이 가능. 약간의 성능 향상이 있으며 필수 사항은 아님.
jdbcType	ResultSet column의 타입을 명시함. java.util.Date의 경우 해당되는 jdbcType이 여러가지(DATE, DATETIME)이므로 정확한 매핑을 위해 명시하는 것이 좋음. JDBC driver에 따라 정의할 필요가 없는 것도 있음.
javaType	세팅될 Java 타입을 명시함.
nullValue	데이터베이스에서 null이 넘어온 경우 null을 대신할 default 값 지정
select	객체 관계를 표현하며 복잡한 property 타입을 로드 할 수 있음 들어갈 값은 반드시 다른 매핑문의 이름이어야 함.

Result 매핑 (2/3)

- Primitive result
 - iBATIS는 primitive 형태의 결과를 허용하지 않음(int, long, double, ..)
 - Wrapping 된 형태의 타입을 리턴(Integer, Long, Double, ..)

```
<select id="getAllOrderCount" resultClass="int">
  select count(*) as cnt
  from order
</select>
```

```
Integer allCount =
  (Integer)client.queryForObject("getAllOrderCount");
```

- JavaBean(TO) 타입으로 Wrapping 할 경우 primitive 형태의 결과를 얻을 수 있음

```
public class ResultTO{
  private int orderCount;
  public int getOrderCount() {
    return orderCount;
  }
  public void setOrderCount(int orderCount) {
    this.orderCount = orderCount;
  }
}
```

```
<resultMap id="primitiveResultMapExample" class="ResultTO">
  <result property="orderCount" column="cnt" />
</resultMap>
```

Result 매핑 (3/3)

- JavaBean과 Map Result의 장단점

방식	장점	단점
Bean	성능 향상 컴파일 타임의 이름 체크 IDE에서 Refactoring 지원 가능 Type casting이 적음	get, set메소드 필요
Map	코드량이 적음	느리다 컴파일 타임 체크 불가 런타임 오류가 많음 Refactoring 지원이 없음

SELECT 구문 실행 (1/4)

- **SqlMap** : `<select>` 태그 이용
 - 속성
 - `id` : 프로그램에서 호출할 이름
 - `resultClass, resultMap` : SELECT 문 실행 결과를 담은 객체
 - `parameterClass, parameterMap` : PARAMETER를 받아올 객체
- **SqlMapClient** – `queryForObject()` : 0또는 1개의 row를 가져올 때 사용 – unique한 값으로 조회 시

<p>Object queryForObject(String id) throws SQLException; Object queryForObject(String id, Object parameter) throws SQLException; Object queryForObject(String id, Object parameter, Object result) throws SQLException;</p>
--

- 첫 번째 메소드 : `parameter` 없이 조회
- 두 번째 메소드 : SQL Map에 정의된 `select`문 `id`와 해당 파라미터로 사용
- 두 번째 메소드 : 결과 결과를 담은 객체를 넣음. `no-argument` 생성자가 없는 `resultClass`를 사용해야 하는 경우 사용
- 1개 이상의 row가 리턴 된 경우 **Exception** 발생

SELECT 구문 실행 (2/4)

- **SqlMapClient – queryForList()** 메소드
 - 1개 이상의 row가 리턴될 경우 사용
 - 결과를 List로 리턴함
 - select 결과의 1개의 row – resultClass의 객체
 - resultClass 객체 들 – List

```
List queryForList(String id) throws SQLException;  
List queryForList(String id, Object parameter) throws SQLException;  
List queryForList(String id, Object parameter, int skip, int max) throws SQLException;
```

- 첫 번째, 두 번째 메소드 : queryForObject와 동일
- 세 번째 메소드 : DB에서 가져온 전체 row중 일부만 사용하고 싶은 경우 사용
 - 예) DB에서 100건을 가져올 때 이 중 앞의 10건만 취하고 싶으면 skip=0, max=10

SELECT 구문 실행 (3/4)

- queryForMap() 메소드
 - 여러 건을 Map 형태로 리턴함

```
Map queryForMap(String id, Object parameter, String key) throws SQLException;  
Map queryForMap(String id, Object parameter, String key, String value) throws SQLException;
```

- 첫 번째 메소드 :
 - key – map의 key값으로 들어갈 property 또는 key를 입력
 - 결과 Map의 value는 resultClass에 지정한 클래스의 객체가 들어간다.
- 두 번째 메소드 :
 - key – map의 key값으로 들어갈 property 또는 key를 입력
 - 결과 Map의 value는 지정한 property 또는 key의 데이터만 들어간다.

queryForMap() 사용예

```
Map accountMap = sqlMap.queryForMap("selectAccountList",null,"accountId");  
Map accountMap = sqlMap.queryForMap("selectAccountList",null,"accountId", "ownerName");
```

SELECT 구문 실행 - 결과 매핑 (4/4)

- 자동 Result Map
 - select문 실행 결과를 자동으로 객체에 매핑 가능함
 - 단일 column 매핑
 - count(id) 값이 Integer 타입으로 자동 매핑된다.

```
<select id="selectAccountCnt" resultClass="int">
  select count(accountId)
  from Account
</select>
```

```
Integer cnt = (Integer)
client.queryForObject("selectAccountCnt");
```

- 다중 column 매핑
 - select 문의 column명과 결과 객체의 Bean property 명이 일치해야 함

```
<select id="selectAccountList" resultClass="AccountTO">
  select
    account_id as id,
    owner_name as name,
    balance
  from account
</select>
```

```
public class AccountTO{
  ...
  setId(String id){}
  setName(String name){}
  setBalance(long balance){}
  ...
}
```

INSERT

- SqlMap : <insert> 태그 사용
 - 속성
 - id : 프로그램에서 호출 할 이름
 - parameterClass, parameterMap : PARAMETER를 받아올 객체
 - Sub tag
 - <selectKey> : insert시 사용할 파라미터를 조회할 경우 사용
 - 속성 : resultClass, keyProperty
- SqlMapClient – insert() 사용

Object insert(String id) throws SQLException;
Object insert(String id, Object parameter) throws SQLException;

- return value : Object
 - <selectKey>를 통해 조회한 result객체

UPDATE

- SqlMap : <update> 태그 사용
 - 속성
 - id : 프로그램에서 호출 할 이름
 - parameterClass, parameterMap : PARAMETER를 받아 올 객체
- SqlMapClient – update() 사용

```
int update (String id) throws SQLException;  
int update(String id, Object parameter) throws SQLException;
```

- return value : int – update가 적용된 record 개수

DELETE

- SqlMap : <delete> 태그 사용
 - 속성
 - id : 프로그램에서 호출 할 이름
 - parameterClass, parameterMap : PARAMETER를 받아 올 객체
- SqlMapClient – delete() 사용

```
int delete(String id) throws SQLException;  
int delete(String id, Object parameter) throws SQLException;
```

- return value : int – 삭제가 적용된 record 개수

Dynamic SQL

- 개요
- Dynamic 태그

개요

- 언제 사용하는가?
 - 하나의 sql 태그에서 쿼리구문을 조건에 따라 동적으로 만들때 사용
 - 예를 들어 where 절의 비교구문이 파라미터객체의 조건에 따라 달라질 경우
 - 예) 조회시 제약조건에 사용할 검색 키워드가 있을 수도 null일 수도 있을 경우.

name=#name# 또는 name IS NULL

```
<select id="selectMember" parameterClass="to.MemberTO"
  resultClass="to.MemberTO">
  SELECT * FROM member
  <dynamic prepend="WHERE ">
    <isNull property="name">
      name IS NULL
    </isNull>
    <isNotNull property="name">
      writer =#name#
    </isNotNull>
  </dynamic>
</select>
```

파라미터로 넘어온 writer에 값이 없는 경우 (null인 경우)

파라미터 넘어온 writer에 값이 있는 경우

SQL 재 사용성 증가

Dynamic 태그

- Dynamic 태그 종류 및 속성
 - dynamic : Root 태그
 - 4가지 카테고리
 - binary, unary, parameter, iterate
 - 공통 속성
 - prepend, open, close

```
<dynamic prepend="WHERE ">
...
<isEmpty property="y">y=#y#</isEmpty>
<isNotNull property="x" removeFirstPrepend="true" prepend="AND"
  open="(" close=")">
  <isEmpty property="x.a" prepend="OR">a=#x.a#</isEmpty>
  <isEmpty property="x.b" prepend="OR">a=#x.b#</isEmpty>
  <isEmpty property="x.c" prepend="OR">a=#x.c#</isEmpty>
</isNotNull>
...
</dynamic>
```

WHERE 구문을 앞에 추가

prepend가 첫번째로
Add되는 경우는 생략한
다.

Dynamic 태그

- <dynamic> 태그
 - Dynamic 태그의 가장 상위 태그(Root 태그)
 - 다른 dynamic 태그 안에 포함될 수 없음
 - 접두/접미 구문을 위한 수단 제공(prepend, open, close)

prepend (옵션)	맨 앞에 접두 구문을 붙인다. (예: “WHERE”) Body에 내용이 없을 경우 접두 구문은 생략된다.
open (옵션)	Body의 내용 앞에 붙여짐. Body의 내용이 없을 경우 생략된다. prepend와 동시에 정의될 경우 prepend 다음에 나타난다.
close (옵션)	Body의 내용 뒤에 붙여짐. Body의 내용이 없을 경우 생략된다.

Dynamic 태그

- Binary 태그
 - 주어진 Parameter 객체간의 property 값을 비교
 - 태그의 조건이 충족되는 경우 Body내용이 나타난다.
 - 태그 속성

property(필수)	Parameter 객체의 property. compareValue 또는 compareProperty에서 비교하는데 쓰인다.
prepend(옵션)	맨 앞에 접두 구문을 붙인다. 이 구문이 생략되는 경우 1. Body의 내용이 없는 경우 2. 부모 태그 속성이 removeFirstPrepend="true" 이고 현재 태그가 부모 Body의 첫 번째 요소인 경우
open(옵션)	Body의 내용 앞에 붙여짐. Body의 내용이 없을 경우 생략된다. prepend와 동시에 정의될 경우 prepend 다음에 나타난다.
close(옵션)	Body의 내용 뒤에 붙여짐. Body의 내용이 없을 경우 생략된다.
removeFirstPrepend(옵션)	첫번째 자식 태그의 prepend를 생략시킴
compareProperty	property 와 비교할 property 이름
compareValue	property 속성에 의해 비교되는 값(상수)

Dynamic 태그

- Binary 태그(계속)
 - Binary 태그 종류

<isEqual>	property속성과 compareProperty/compareValue속성이 같은지 비교
<isNotEqual>	property속성과 compareProperty/compareValue속성이 다른지 비교
<isGreaterThan>	property속성이 compareProperty/compareValue속성보다 큰지 비교
<isGreaterEqual>	property속성이 compareProperty/compareValue속성보다 크거나 같은지 비교
<isLessThan>	property속성이 compareProperty/compareValue속성보다 작은지 비교
<isLessEqual>	property속성이 compareProperty/compareValue속성보다 작거나 같은지 비교

Dynamic 태그

- Binary 태그(계속)
 - Binary 태그 예제

```
<select id="getShippingType" parameterClass="CartTO"
  resultClass="ShippingTO">
  SELECT * FROM Shipping
  <dynamic prepend="WHERE ">
    <isGreaterEqual property="weight" compareValue="100">
      shippingType='FREIGHT'
    </isGreaterEqual>
    <isLessThan property="weight" compareValue="100">
      shippingType='STANDARD'
    </isLessThan>
  </dynamic>
</select>
```

CartTO.getWeight() >= 100 인지 체크

CartTO.getWeight() < 100 인지 체크

Dynamic 태그

- Unary 태그
 - 비교를 수행하지 않고 Bean property 상태를 체크한다.
 - 태그의 조건이 충족되는 경우 Body내용이 나타남
 - Unary 태그 속성

property(필수)	상태를 체크하기 위한 Parameter 객체의 property
prepend(옵션)	맨 앞에 접두 구문을 붙인다. 이 구문이 생략되는 경우 <ol style="list-style-type: none">1. Body의 내용이 없는 경우2. 부모 태그 속성이 removeFirstPrepend="true" 이고 현재 태그가 부모 Body의 첫번째 요소인 경우
open(옵션)	Body의 내용 앞에 붙여짐. Body의 내용이 없을 경우 생략된다. prepend와 동시에 정의될 경우 prepend 다음에 나타난다.
close(옵션)	Body의 내용 뒤에 붙여짐. Body의 내용이 없을 경우 생략된다.
removeFirstPrepend(옵션)	첫 번째 자식 태그의 prepend를 생략시킴

Dynamic 태그

- Unary 태그(계속)
 - Unary 태그 종류

<isPropertyAvailable>	특정 property가 존재하는지 체크. Map의 경우 key가 존재하는지 체크함.
<isNotPropertyAvailable>	특정 property가 존재하지 않는지 체크. Map의 경우 key가 존재하지 않는지 체크함.
<isNull>	특정 property가 null인지 체크.
<isNotNull>	특정 property가 null이 아닌지 체크.
<isEmpty>	특정 property가 null이거나 비어있는지 체크.
<isNotEmpty>	특정 property가 null이아니고 비어있지 않는지 체크.

```
<select id="getProducts" parameterClass="Product"
  resultClass="Product">
  SELECT * FROM Products
  <isNotEmpty property="productType">
    productType=#productType#
  </isNotEmpty>
</select>
```

*Product.getProductType != null 이고
Product.getProductType.length() > 0 인지
체크*

Dynamic 태그

- Parameter 태그
 - Parameter 가 매핑문에 들어왔는지 체크하는 태그
 - 태그 속성 : prepend, open, close, removeFirstPrepend
 - 태그 종류

<code><isParameterPresent></code>	Parameter가 존재하는지 체크
<code><isNotParameterPresent></code>	Parameter가 존재하지 않는지 체크

```
<select id="getProducts" resultClass="Product">
  SELECT * FROM Products
  <isParameterPresent prepend="WHERE ">
    <isNotEmpty property="productType">
      productType=#productType#
    </isNotEmpty>
  </isParameterPresent>
</select>
```

이 select 매핑문에 들어오는 어떤 Parameter든지 존재하는지 체크

Dynamic 태그

- <iterate> 태그
 - Collection 형태의 property로 SQL문의 반복적인 구간 생성
 - 각각의 구간은 "conjunction" 속성으로 분리됨
 - 태그 속성 : property(필수), prepend, open, close, conjunction, removeFirstPrepend

conjunction	반복되는 SQL 구문 사이에 구분자로 들어감 (예: ', ')
-------------	------------------------------------

```
<select id="getProducts" parameterClass="Product"
resultClass="Product">
  SELECT * FROM Products
  <dynamic prepend="WHERE">
    <iterate property="productType" open="(" close=")"
      conjunction="OR" >
      productType=#productType[]#
    </iterate>
  </dynamic>
</select>
```

*Product의 Collection 형태의
property*

반복 구간이 'OR' 로 구분됨

Transaction 관리

iBATIS Transaction 관리

- Auto Transaction
- Local Transaction

Auto Commit

- iBATIS의 자동 트랜잭션 지원
 - Statement를 수행하기만 해도 적용됨
 - 별다른 설정이 없음

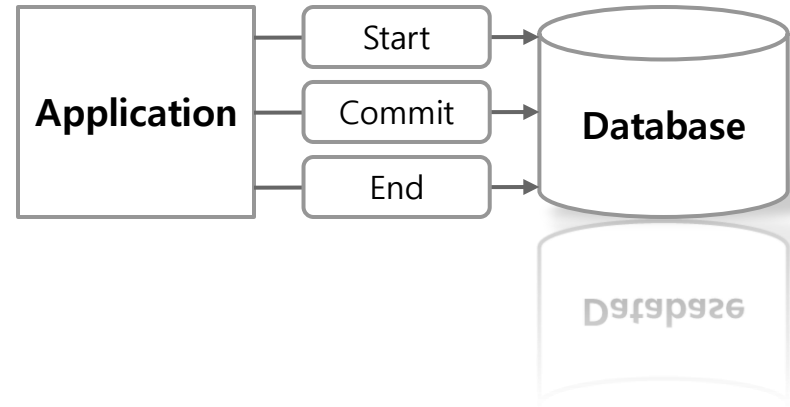
```
public void runStatementsUsingAutomaticTransactions() {  
    SqlMapClient sqlMapClient =  
        SqlMapClientConfig.getSqlMapClient();  
    Person p = (Person) sqlMapClient.queryForObject("getPerson",  
        new Integer(9));  
    p.setLastName("Smith");  
    sqlMapClient.update("updatePerson", p);  
}
```

Local Transaction

- 가장 일반적인 형태의 Transaction
- JDBC Transaction Manager 사용
 - sqlmap-config.xml에 설정

```
<transactionManager type="JDBC">  
  <dataSource type="SIMPLE">  
    <property .../>  
    <property .../>  
    <property .../>  
  </dataSource>  
</transactionManager>
```

Local 트랜잭션 Manager 설정



Local Transaction

- Local Transaction 적용 예

```
public void runStatementsUsingLocalTransactions() {
```

```
    SqlMapClient sqlMapClient = SqlMapClientConfig.getSqlMapClient();
```

```
    try {
```

```
        sqlMapClient.startTransaction();
```

```
        Person p = (Person) sqlMapClient.queryForObject("getPerson",  
            new Integer(9));
```

```
        p.setLastName("Smith");
```

```
        sqlMapClient.update("updatePerson", p);
```

```
        Department d = (Department) sqlMapClient.queryForObject("getDept",  
            new Integer(3));
```

```
        p.setDepartment(d);
```

```
        sqlMapClient.update("updatePersonDept", p);
```

```
        sqlMapClient.commitTransaction();
```

```
    } finally {
```

```
        sqlMapClient.endTransaction();
```

```
    }
```

```
}
```

Transaction 시작

Commit

Transaction 종료

Spring과 iBATIS 연동

Spring과 iBATIS (1/2)

- iBATIS 클라이언트 템플릿 설정
 - SqlMapClientFactoryBean 설정
 - SqlMapClient 객체를 생성하는 Factory 객체
 - SqlMapClientTemplate 설정
 - iBATIS의 SqlMapClient 객체의 역할을 하는 객체

```
<bean id="sqlMapClient"  
  class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">  
  <property name="dataSource" ref="dataSource"/>  
  <property name="configLocation" value="sqlMapConfig.xml"/>  
</bean>
```

```
<bean id="sqlMapClientTemplate"  
  class="org.springframework.orm.ibatis.SqlMapClientTemplate">  
  <property name="sqlMapClient" ref="sqlMapClient"/>  
</bean>
```

iBATIS 설정 파일

Spring과 iBATIS (2/2)

- iBATIS 클라이언트 템플릿 설정
 - DAO에서 템플릿 사용 – SqlMapClientTemplate를 주입 받는다.

```
public class IbatisUserDao implements UserDao {
    public IbatisUserDao() {}
    private SqlMapClientTemplate sqlMapClientTemplate;
    public void setSqlMapClientTemplate(SqlMapClientTemplate sqlMapClientTemplate) {
        this.sqlMapClientTemplate = sqlMapClientTemplate;
    }
    @Override
    public User selectMemberById(String id) {
        return (MemberDTO)sqlMapClientTemplate.queryForObject("SELECT_MEMBER_BY_ID", id);
    }
    @Override
    public void insertMember(MemberDTO mto) {
        sqlMapClientTemplate.insert("INSERT_MEMBER", mto);
    }
}
```

```
<bean id="dao" class="dao. IbatisUserDao">
    <property name="sqlMapClientTemplate" ref="sqlMapClientTemplate"/>
</bean>
```