
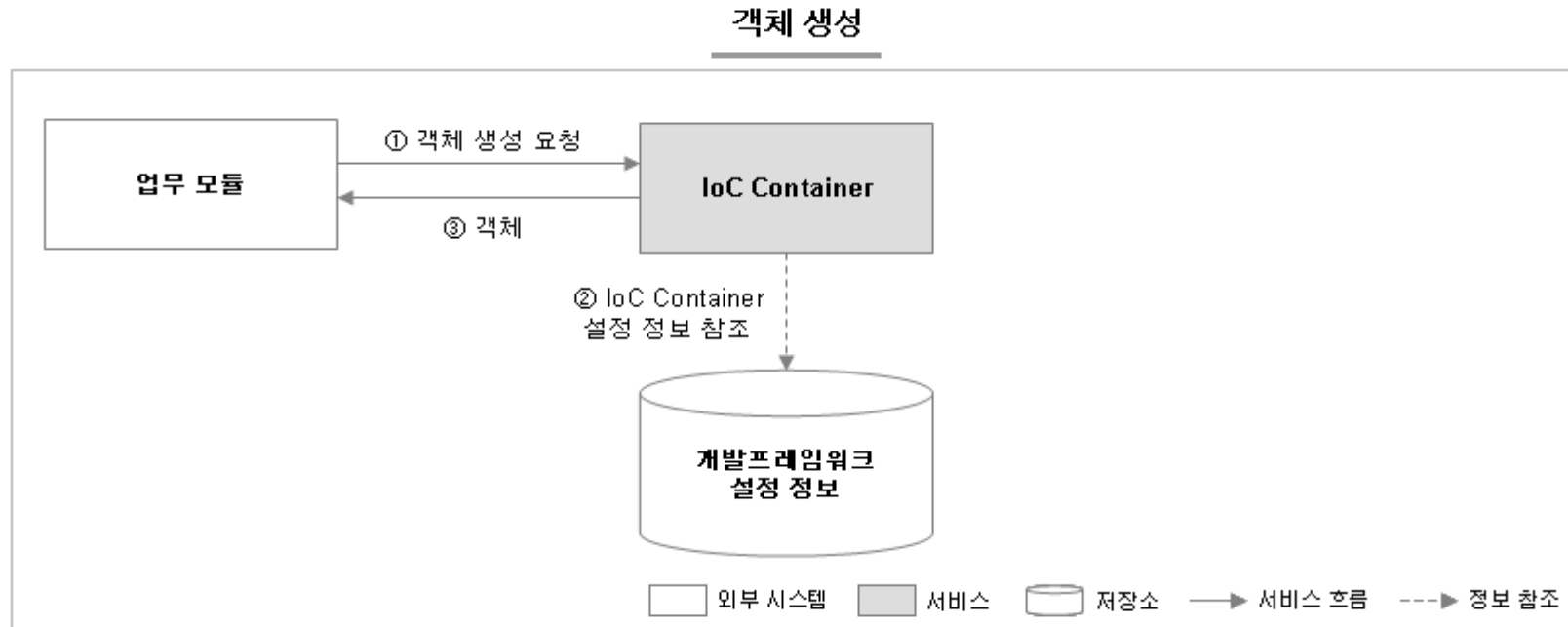


### □ 개요

- 프레임워크의 기본적인 기능인 **IoC(Inversion of Control) Container** 기능을 제공하는 서비스이다.
- 객체의 생성 시, 객체가 참조하고 있는 타 객체에 대한 의존성을 소스 코드 내부에서 하드 코딩하는 것이 아닌, 소스 코드 외부에서 설정하게 함으로써, 유연성 및 확장성을 향상시킨다.
- 주요 기능 : **Dependency Injection, Bean Lifecycle Management** 
- 오픈소스 : Spring Framework 3.0.5의 IoC Container를 수정없이 사용함.
- **의존성 관리의 중요성**

### □ 개요



- ① 업무 모듈은 IoC Container 서비스에 객체 생성을 요청한다.
- ② IoC Container는 표준 프레임워크 설정 정보에 객체 생성을 위한 종속성 정보 등과 같은 IoC Container 설정 정보를 참조한다.
- ③ IoC Container는 설정 정보에 따라 객체를 생성하여 업무 모듈에게 돌려준다.

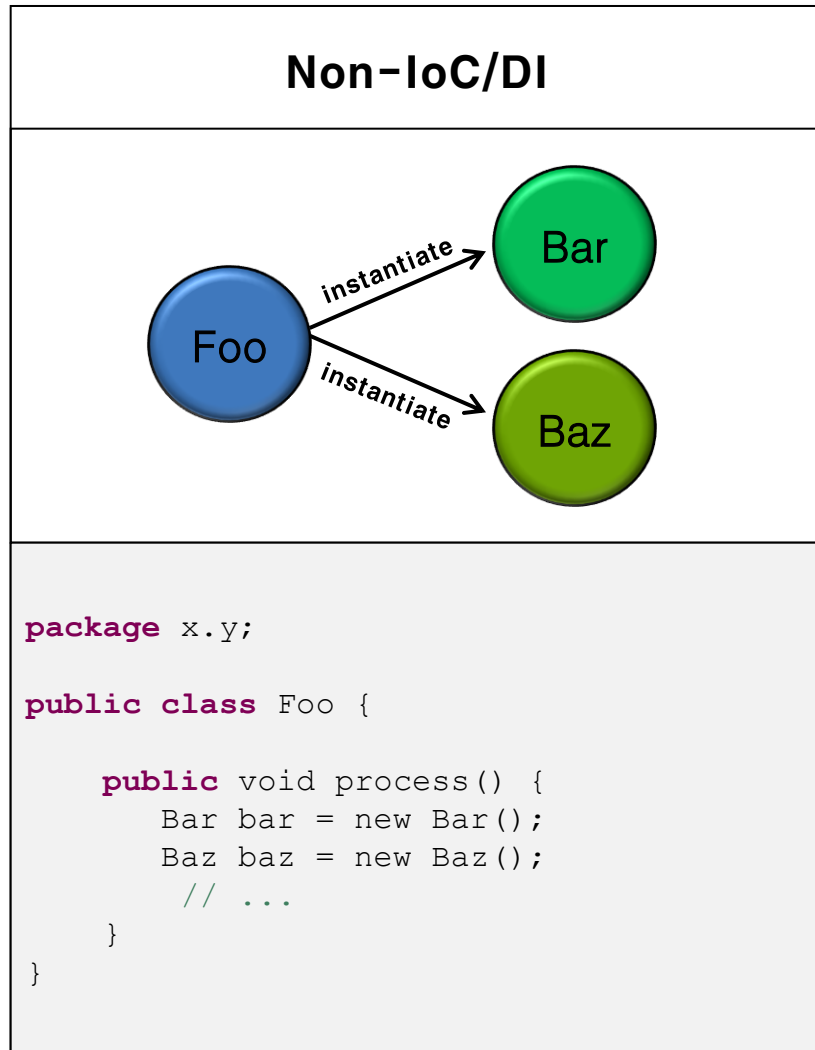
### ❑ IoC(Inversion of Control)이란?

- IoC는 Inversion of Control의 약자로 한글로 “제어의 역전” 또는 “역제어”라고 부른다. 어떤 모듈이 제어를 가진다는 것은 “어떤 모듈을 사용할 것인지”, “모듈의 함수는 언제 호출할 것인지” 등을 스스로 결정한다는 것을 의미한다. 이러한 제어가 역전되었다는 것은, 어떤 모듈이 사용할 모듈을 스스로 결정하는 것이 아니라 다른 모듈에게 선택권을 넘겨준다는 것을 의미한다.

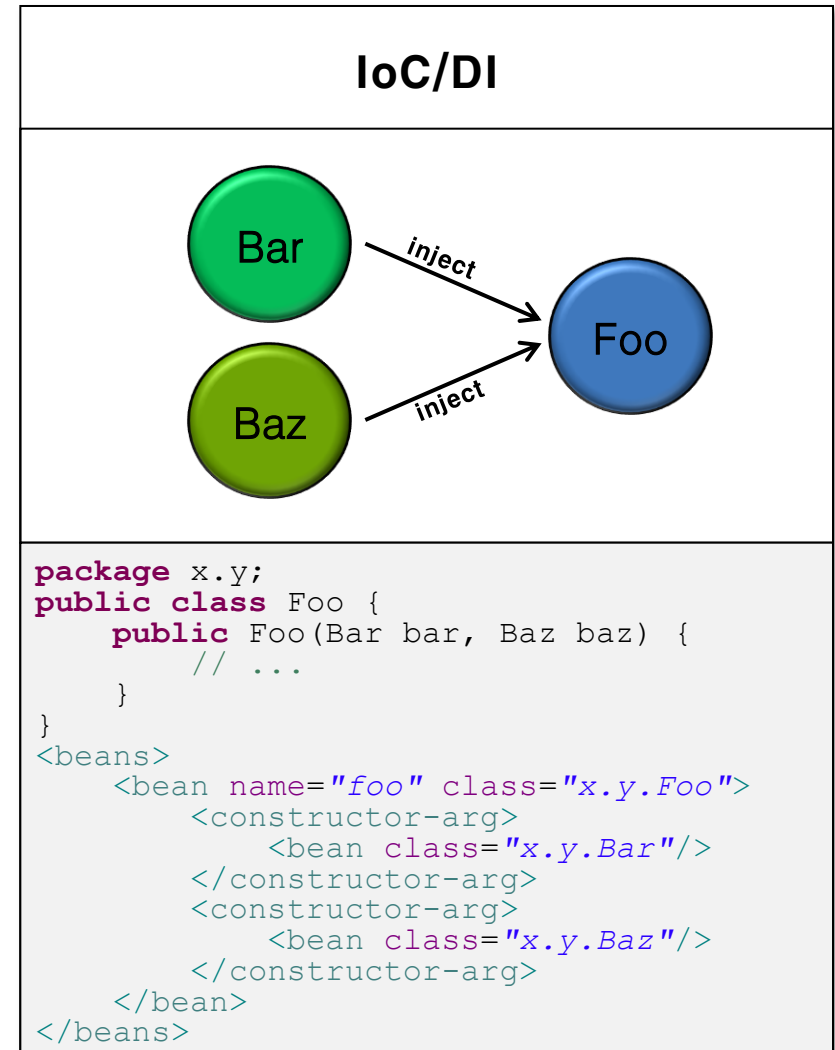
### ❑ DI(Dependency Injection)이란?

- **Dependency Injection**이란 모듈간의 의존성을 모듈의 외부(컨테이너)에서 주입시켜주는 기능으로 **Inversion of Control**의 한 종류이다.
- 런타임시 사용하게 될 의존대상과의 관계를 Spring Framework 이 총체적으로 결정하고 그 결정된 의존특징을 런타임시 부여한다.

### ❑ Non-IoC/DI vs IoC/DI



VS



### ❑ Container

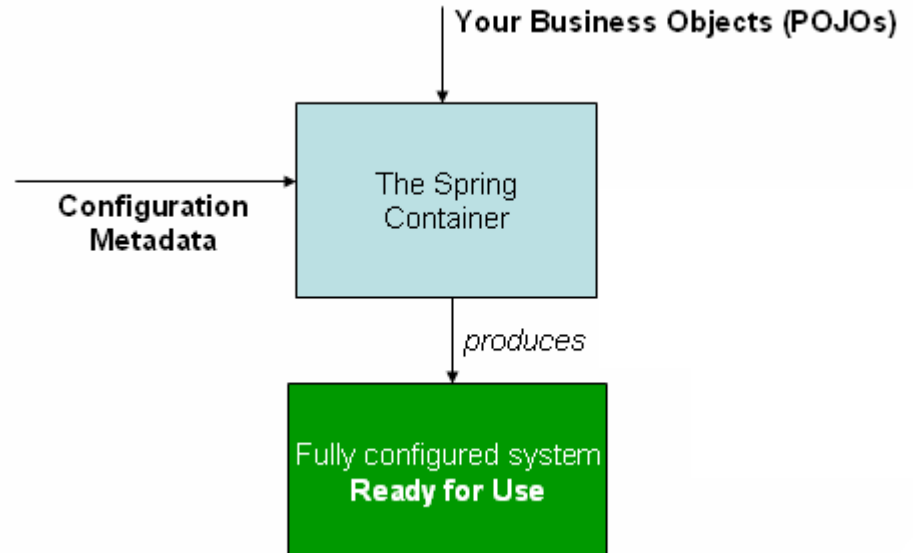
- Spring IoC Container는 객체를 생성하고, 객체 간의 의존성을 이어주는 역할을 한다.

### ❑ 설정 정보(Configuration Metadata)

- Spring IoC container가 “객체를 생성하고, 객체간의 의존성을 이어줄 수 있도록” 필요한 정보를 제공한다. 설정 정보는 기본적으로 XML 형태로 작성되며, 추가적으로 Java Annotation을 이용하여서도 설정이 가능하다.

### ❑ Bean

- Spring IoC Container에 의해 생성되고 관리되는 객체를 의미한다.



### ❑ BeanFactory

- **BeanFactory 인터페이스는 Spring IoC Container의 기능을 정의하고 있는 기본 인터페이스**이다.
- Bean 생성 및 의존성 주입, 생명주기 관리 등의 기능을 제공한다.

### ❑ ApplicationContext

- BeanFactory 인터페이스를 상속받는 ApplicationContext는 BeanFactory가 제공하는 기능 외에 Spring AOP, 메시지 리소스 처리(국제화에 사용됨), 이벤트 처리 등의 기능을 제공한다.
- 모든 ApplicationContext 구현체는 BeanFactory의 기능을 모두 제공하므로, 특별한 경우를 제외하고는 ApplicationContext를 사용하는 것이 바람직하다.
- Spring Framework는 다수의 ApplicationContext 구현체를 제공한다. 다음은 ClassPathXmlApplicationContext를 생성하는 예제이다.

```
ApplicationContext context = new ClassPathXmlApplicationContext(  
    new String[] { "services.xml", "daos.xml" });  
Foo foo = (Foo) context.getBean("foo");  
  
// an Application is also a BeanFactory (via inheritance)  
BeanFactory factory = context;
```

※ Spring Container = Bean Factory = ApplicationContext = DI Container = IoC Container

### □ XML 설정 파일(1/2)

- XML 설정 파일은 <beans/> element를 root로 갖는다. 아래는 기본적인 XML 설정 파일의 모습이다.

```
<?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-3.0.xsd">

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

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

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

</beans>
```

### □ XML 설정 파일(2/2)

- XML 설정은 여러 개의 파일로 구성될 수 있으며, <import/> element를 사용하여 다른 XML 설정 파일을 import 할 수 있다.

```
<beans>

    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>

</beans>
```



### □ Bean 정의

- Bean 정의는 Bean을 객체화하고 의존성을 주입하는 등의 관리를 위한 정보를 담고 있다. XML 설정에서는 `<bean/>` element가 Bean 정의를 나타낸다. Bean 정의는 아래와 같은 속성을 가진다.



### □ Bean 이름

- 모든 Bean은 하나의 id를 가지며, 하나 이상의 name을 가질 수 있다. id는 container 안에서 고유해야 한다.

```
<bean id="exampleBean" class="example.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

- Bean은 <alias/> element를 이용하여 추가적인 name을 가질 수 있다.

```
<alias name="fromName" alias="toName"/>
```

### □ Bean Class

- 모든 Bean은 객체화를 위한 Java class가 필요하다. (예외적으로 상속의 경우 class가 없어도 된다.)

```
<bean id="exampleBean" class="example.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

### □ Bean 객체화(1/2)

- 일반적으로 Bean 객체화는 Java 언어의 'new' 연산자를 사용한다. 이 경우 별도의 설정은 필요없다.
- 'new' 연산자가 아닌 static factory 메소드를 사용하여 Bean을 객체화할 수 있다. 이 경우 Constructor Injection 방식의 의존성 주입 설정을 따른다.

```
<bean id="exampleBean"
      class="examples.ExampleBean"
      factory-method="createInstance"/>
```

- 자신의 static factory 메소드가 아닌 별도의 Factory 클래스의 static 메소드를 사용하여 Bean을 객체화할 수 있다. 이 경우 역시 Constructor Injection 방식의 의존성 주입 설정을 따른다.

```
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="serviceLocator" class="com.foo.DefaultServiceLocator">
  <!-- inject any dependencies required by this locator bean -->
</bean>

<!-- the bean to be created via the factory bean -->
<bean id="exampleBean"
      factory-bean="serviceLocator"
      factory-method="createInstance"/>
```

```
ExampleBean exampleBean = new ExampleBean();
ExampleBean exampleBean = ExampleBean.createInstance();
ExampleBean exampleBean = DefaultServiceLocator.createInstance();
```

### □ Bean 객체화(2/2)

- <bean/> element의 'lazy-init' attribute를 사용하여 Bean 객체화 시기를 설정할 수 있다.
  - 일반적으로 Bean 객체화는 BeanFactory가 객체화되는 시점에 수행된다. 만약, 'lazy-init' attribute 값이 'true'인 경우, 설정된 Bean의 객체가 실제로 필요하다고 요청한 시점에 객체화가 수행된다.
  - 'lazy-init' attribute가 설정되어 있지 않으면 기본값을 사용한다. Spring Framework의 기본값은 'false'이다.

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

- <beans/> element의 'default-lazy-init' attribute를 사용하여 XML 설정 파일 내의 모든 Bean 정의에 대한 lazy-init attribute의 기본값을 설정할 수 있다.

```
<beans default-lazy-init="true">
    <!-- no beans will be pre-instantiated... -->
</beans>
```

### □ 의존성 주입(1/16)

- 의존성 주입에는 **Constructor Injection**과 **Setter Injection**의 두가지 방식이 있다.
- Constructor Injection(1/3)
  - Constructor Injection은 **argument**를 갖는 생성자를 사용하여 의존성을 주입하는 방식이다. **<constructor-arg/>** element를 사용한다. 생성자의 argument와 **<constructor-arg/>** element는 class가 같은 것끼리 매핑한다.

```
package x.y;

public class Foo {

    public Foo(Bar bar, Baz baz) {
        // ...
    }
}
```

```
<beans>
    <bean name="foo" class="x.y.Foo">
        <constructor-arg>
            <bean class="x.y.Bar"/>
        </constructor-arg>
        <constructor-arg>
            <bean class="x.y.Baz"/>
        </constructor-arg>
    </bean>
</beans>
```

### □ 의존성 주입(2/16)

#### – Constructor Injection(2/3)

- 만약 생성자가 같은 class의 argument를 가졌거나 primitive type인 경우 argument와 <constructor-arg/> element간의 매핑이 불가능하다. 이 경우, **Type**을 지정하거나 순서를 지정할 수 있다.

```
package examples;

public class ExampleBean {

    // No. of years to the calculate the Ultimate Answer
    private int years;

    // The Answer to Life, the Universe, and Everything
    private String ultimateAnswer;

    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

### □ 의존성 주입(3/16)

#### – Constructor Injection(3/3)

- Type 지정

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <constructor-arg type="int" value="7500000"/>  
    <constructor-arg type="java.lang.String" value="42"/>  
</bean>
```

- 순서 지정

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <constructor-arg index="0" value="7500000"/>  
    <constructor-arg index="1" value="42"/>  
</bean>
```

### □ 의존성 주입(4/16)

#### – Setter Injection(1/2)

- Setter Injection은 **argument가 없는 기본 생성자를 사용하여 객체를 생성한 후, setter 메소드를 사용하여 의존성을 주입하는 방식으로, <property/> element를 사용한다.**
- Class에 attribute(또는 setter 메소드 명)과 <property/> element의 'name' attribute를 사용하여 매핑한다.

```
public class ExampleBean {  
  
    private AnotherBean beanOne;  
    private YetAnotherBean beanTwo;  
    private int i;  
  
    public void setBeanOne(AnotherBean beanOne) {  
        this.beanOne = beanOne;  
    }  
  
    public void setBeanTwo(YetAnotherBean beanTwo) {  
        this.beanTwo = beanTwo;  
    }  
  
    public void setIntegerProperty(int i) {  
        this.i = i;  
    }  
}
```



### □ 의존성 주입(5/16)

#### – Setter Injection(2/2)

```
<bean id="exampleBean" class="examples.ExampleBean">
  <!-- setter injection using the nested <ref/> element -->
  <property name="beanOne"><ref bean="anotherExampleBean"/></property>

  <!-- setter injection using the neater 'ref' attribute -->
  <property name="beanTwo" ref="yetAnotherBean"/>
  <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

- 매핑 규칙은 <property/> element의 'name' attribute의 첫문자를 알파벳 대문자로 변경하고 그 앞에 'set'을 붙인 setter 메소드를 호출한다.

```
<bean id="exampleBean"
      class="examples.ExampleBean">
  <property name="beanOne">
    <ref bean="anotherExampleBean"/>
  </property>
</bean>
```

```
public class ExampleBean {
  public void setBeanOne(
    AnotherBean beanOne)
  {
    this.beanOne = beanOne;
  }
}
```

### □ 의존성 주입(6/16)

#### – 의존성 상세 설정(1/11)

- <constructor-arg/> element 과 <property/> element는 ‘명확한 값’, ‘다른 Bean에 대한 참조’, ‘Inner Bean’, ‘Collection’, ‘Null’ 등의 값을 가질 수 있다.
- 명확한 값

Java Primitive Type, String 등의 명확한 값을 나타낸다. 사람이 인식 가능한 문자열 형태를 값으로 갖는 <value/> element를 사용한다. Spring IoC container가 String 값을 해당하는 type으로 변환하여 주입해준다.

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  <!-- results in a setDriverClassName(String) call -->
  <property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property name="url">
    <value>jdbc:mysql://localhost:3306/mydb</value>
  </property>
  <property name="username">
    <value>root</value>
  </property>
  <property name="password">
    <value>masterkaoli</value>
  </property>
</bean>
```

### □ 의존성 주입(7/16)

#### – 의존성 상세 설정(2/11)

- 다른 Bean에 대한 참조

<ref/> element를 사용하여 다른 Bean 객체를 참조할 수 있다. 참조할 객체를 지정하는 방식은 'container', 'local', 'parent' 등이 있다.

1. container : 가장 일반적인 방식으로 같은 container 또는 부모 container에서 객체를 찾는다.

```
<ref bean="someBean"/>
```

2. local : 같은 XML 설정 파일 내에 정의된 Bean 객체를 찾는다.

```
<ref local="someBean"/>
```

3. parent : 부모 XML 설정 파일 내에 정의된 Bean 객체를 찾는다.

```
<!-- in the parent context -->
<bean id="accountService" class="com.foo.SimpleAccountService">
  <!-- insert dependencies as required as here -->
</bean>
```

```
<!-- in the child (descendant) context -->
<bean id="accountService" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target">
    <ref parent="accountService"/>
  </property>
</bean>
```

### □ 의존성 주입(8/16)

#### – 의존성 상세 설정(3/11)

- Inner Bean

<property/> 또는 <constructor-arg/> element 안에 있는 <bean/> element를 inner bean이라고 한다. Inner bean의 'scope' flag 와 'id', 'name'은 무시된다. Inner bean의 scope은 항상 prototype이다. 따라서 inner bean을 다른 bean에 주입하는 것은 불가능하다.

```
<bean id="outer" class="...">
  <!-- instead of using a reference to a target bean, simply define the target bean
        inline -->
  <property name="target">
    <bean class="com.example.Person"> <!-- this is the inner bean -->
      <property name="name" value="Fiona Apple"/>
      <property name="age" value="25"/>
    </bean>
  </property>
</bean>
```

### □ 의존성 주입(9/16)

#### – 의존성 상세 설정(4/11)

##### • Collection(1/2)

Java Collection 타입인 List, Set, Map, Properties를 표현하기 위해 <list/>, <set/>, <map/>, <props/> element가 사용된다. map의 key와 value, set의 value의 값은 아래 element 중 하나가 될 수 있다.

bean   ref   idref   list   set   map   props   value   null
--

```
<bean id="moreComplexObject" class="example.ComplexObject">
  <!-- results in a setAdminEmails(java.util.Properties) call -->
  <property name="adminEmails">
    <props>
      <prop key="administrator">administrator@example.org</prop>
      <prop key="support">support@example.org</prop>
      <prop key="development">development@example.org</prop>
    </props>
  </property>
  <!-- results in a setSomeList(java.util.List) call -->
  <property name="someList">
    <list>
      <value>a list element followed by a reference</value>
      <ref bean="myDataSource" />
    </list>
  </property>
  ...
</bean>
```

### □ 의존성 주입(10/16)

#### – 의존성 상세 설정(5/11)

- Collection(2/2)

```
...

<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
  <map>
    <entry>
      <key><value>an entry</value></key>
      <value>just some string</value>
    </entry>
    <entry>
      <key><value>a ref</value></key>
      <ref bean="myDataSource" />
    </entry>
  </map>
</property>
<!-- results in a setSomeSet(java.util.Set) call -->
<property name="someSet">
  <set>
    <value>just some string</value>
    <ref bean="myDataSource" />
  </set>
</property>
</bean>
```

### □ 의존성 주입(11/16)

#### – 의존성 상세 설정(6/11)

- Null

Java의 null 값을 사용하기 위해서 `<null/>` element를 사용한다. Spring IoC container는 value 값이 설정되어 있지 않은 경우 빈문자열(“”)로 인식한다.

```
<bean class="ExampleBean">
  <property name="email"><value/></property>
</bean>
```

위 ExampleBean의 email 값은 “”이다. 아래는 email의 값이 null인 예제이다.

```
<bean class="ExampleBean">
  <property name="email"><null/></property>
</bean>
```

### □ 의존성 주입(12/16)

#### – 의존성 상세 설정(7/11)

- 간편한 표기 1

<property/>, <constructor-arg/>, <entry/> element의 <value/> element는 'value' attribute로 대체될 수 있다.

```
<property name="myProperty">
  <value>hello</value>
</property>
```

=

```
<property name="myProperty" value="hello"/>
```

```
<constructor-arg>
  <value>hello</value>
</constructor-arg>
```

=

```
<constructor-arg value="hello"/>
```

```
<entry key="myKey">
  <value>hello</value>
</entry>
```

=

```
<entry key="myKey" value="hello"/>
```



### □ 의존성 주입(13/16)

#### – 의존성 상세 설정(8/11)

- 간편한 표기 2

<property/>, <constructor-arg/> element의 <ref/> element는 'ref' attribute로 대체될 수 있다.

```
<property name="myProperty">
  <ref bean="myBean">
</property>
```

=

```
<property name="myProperty" ref="myBean"/>
```

```
<constructor-arg>
  <ref bean="myBean">
</constructor-arg>
```

=

```
<constructor-arg ref="myBean"/>
```

- 간편한 표기 3

<entry/> element의 'key', 'ref' element 는 'key-ref', 'value-ref' attribute로 대체될 수 있다.

```
<entry>
  <key>
    <ref bean="myKeyBean" />
  </key>
  <ref bean="myValueBean" />
</entry>
```

=

```
<entry key-ref="myKeyBean"
      value-ref="myValueBean"/>
```

### □ 의존성 주입(14/16)

#### – 의존성 상세 설정(9/11)

- p-namespace(1/2)

<property/> element 대신 'p-namespace'를 사용하여 XML 설정을 작성할 수 있다. 아래 classic bean과 p-namespace bean은 동일한 Bean 설정이다.

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

  <bean name="classic" class="com.example.ExampleBean">
    <property name="email" value="foo@bar.com"/>
  </bean>

  <bean name="p-namespace" class="com.example.ExampleBean"
    p:email="foo@bar.com"/>
</beans>
```

### □ 의존성 주입(15/16)

#### – 의존성 상세 설정(10/11)

- p-namespace(2/2)

Attribute 이름 끝에 '-ref'를 붙이면 참조로 인식한다.

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

    <bean name="john-classic" class="com.example.Person">
        <property name="name" value="John Doe"/>
        <property name="spouse" ref="jane"/>
    </bean>

    <bean name="john-modern"
        class="com.example.Person"
        p:name="John Doe"
        p:spouse-ref="jane"/>

    <bean name="jane" class="com.example.Person">
        <property name="name" value="Jane Doe"/>
    </bean>
</beans>
```

### □ 의존성 주입(16/16)

#### – 의존성 상세 설정(11/11)

- Compound property name

복합 형식의 property 이름도 사용할 수 있다.

```
<bean id="foo" class="foo.Bar">  
    <property name="fred.bob.sammy" value="123" />  
</bean>
```

### □ Bean Scope(1/4)

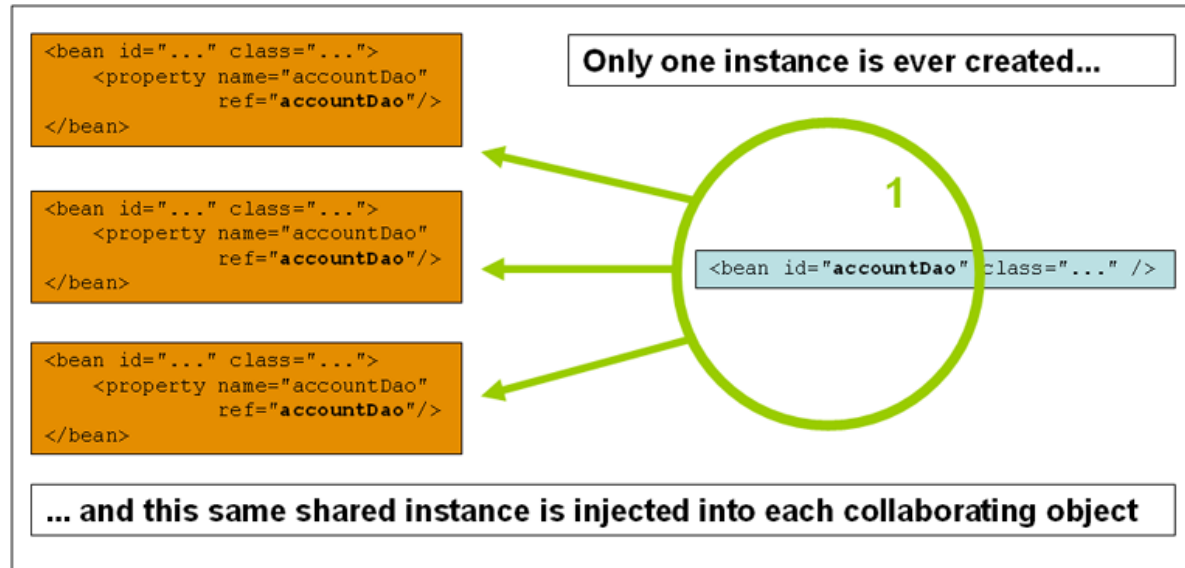
- Bean Scope은 객체가 유효한 범위로 아래 5가지의 scope이 있다.
- Spring Bean의 기본 Scope는 **singleton** 이다.

Scope	설 명
singleton	하나의 Bean 정의에 대해서 Spring IoC Container 내에 단 하나의 객체만 존재한다.
prototype	하나의 Bean 정의에 대해서 다수의 객체가 존재할 수 있다.
request	하나의 Bean 정의에 대해서 하나의 HTTP request의 생명주기 안에 단 하나의 객체만 존재한다; 즉, 각각의 HTTP request는 자신만의 객체를 가진다. Web-aware Spring ApplicationContext 안에서만 유효하다.
session	하나의 Bean 정의에 대해서 하나의 HTTP Session의 생명주기 안에 단 하나의 객체만 존재한다. Web-aware Spring ApplicationContext 안에서만 유효하다.
global session	하나의 Bean 정의에 대해서 하나의 global HTTP Session의 생명주기 안에 단 하나의 객체만 존재한다. 일반적으로 portlet context 안에서 유효하다. Web-aware Spring ApplicationContext 안에서만 유효하다.

### ❑ Bean Scope(2/4)

#### – Singleton Scope

- Bean이 singleton인 경우, 단 하나의 객체만 공유된다.



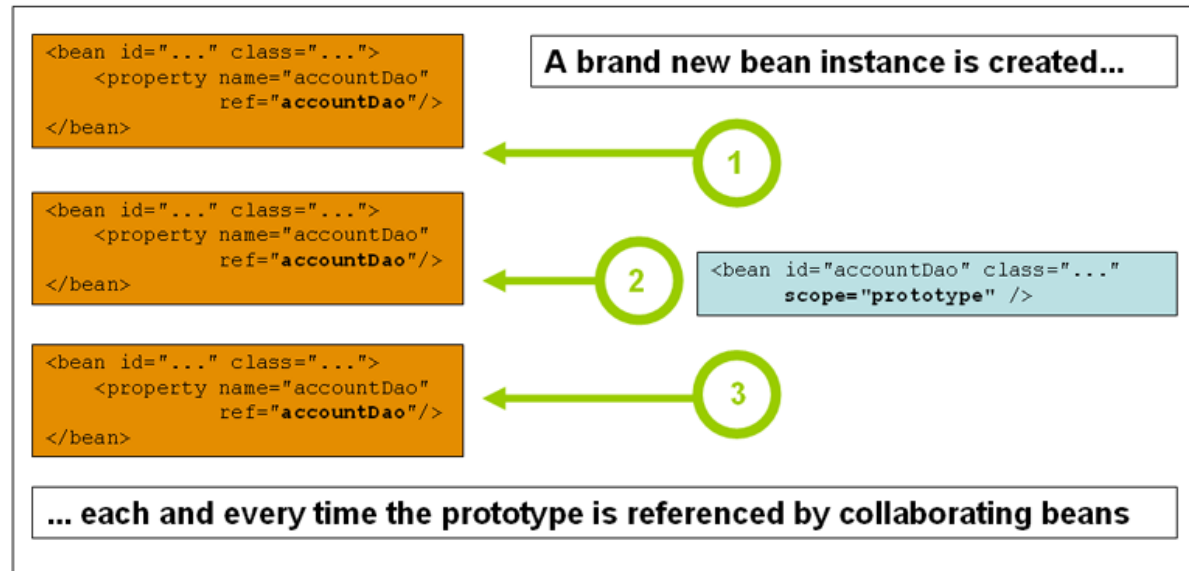
- Spring의 기본 scope은 'singleton'이다. 설정하는 방법은 아래와 같다.

```
<bean id="accountService" class="com.foo.DefaultAccountService"/>  
  
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>  
  
<bean id="accountService" class="com.foo.DefaultAccountService" singleton="true"/>
```

### □ Bean Scope(3/4)

#### – Prototype Scope

- Singleton이 아닌 prototype scope의 형태로 정의된 **bean**은 필요한 매 순간 새로운 **bean** 객체가 생성된다.



- 설정하는 방법은 아래와 같다.

```
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

<bean id="accountService" class="com.foo.DefaultAccountService" singleton="false"/>
```

### □ Bean Scope(4/4)

#### – 기타 Scope

- request, session, global session은 Web 기반 어플리케이션에서만 유효한 scope이다.

- Request Scope

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

위 정의에 따라, Spring container는 모든 HTTP request에 대해서 'loginAction' bean 정의에 대한 LoginAction 객체를 생성할 것이다. 즉, 'loginAction' bean은 HTTP request 수준에 한정된다(scoped). 요청에 대한 처리가 완료되었을 때, 한정된(scoped) bean도 폐기된다.

- Session Scope

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

위 정의에 따라, Spring container는 하나의 HTTP Session 일생동안 'userPreferences' bean 정의에 대한 UserPreferences 객체를 생성할 것이다. 즉, 'userPreferences' bean은 HTTP Session 수준에 한정된다(scoped). HTTP Session이 폐기될 때, 한정된(scoped) bean도 폐기된다.

- Global Session Scope

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>
```

global session scope은 HTTP Session scope과 비슷하지만 단지 portlet-based web 어플리케이션에서만 사용할 수 있다.

Portlet 명세(specifications)는 global Session을 하나의 portlet web 어플리케이션을 구성하는 여러 portlet들 모두가 공유하는 것으로 정의하고 있다. global session scope으로 설정된 bean은 global portlet Session의 일생에 한정된다.





### □ Bean 성질 변화(1/4)

#### – Lifecycle Callback

- Spring IoC Container는 **Bean의 각 생명주기에 호출되도록 설정된 메소드를 호출해준다.**
- Initialization callback

org.springframework.beans.factory.InitializingBean interface를 구현하면 bean에 필요한 모든 property를 설정한 후, 초기화 작업을 수행한다. InitializingBean interface는 다음 메소드를 명시하고 있다.

```
void afterPropertiesSet() throws Exception;
```

일반적으로, InitializingBean interface의 사용을 권장하지 않는다. 왜냐하면 code가 불필요하게 Spring과 결합되기(couple) 때문이다. 대안으로, bean 정의는 초기화 메소드를 지정할 수 있다. XML 기반 설정의 경우, 'init-method' attribute를 사용한다.

```
public class ExampleBean {  
  
    public void init() {  
        // do some initialization work  
    }  
}
```

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

### □ Bean 성질 변화(2/4)

#### – Lifecycle Callback

- Destruction callback

org.springframework.beans.factory.DisposableBean interface를 구현하면, container가 파괴될 때 bean이 callback을 받을 수 있다. DisposableBean interface는 다음 메소드를 명시하고 있다.

```
void destroy() throws Exception;
```

일반적으로, DisposableBean interface의 사용을 권장하지 않는다. 왜냐하면 code가 불필요하게 Spring과 결합되기(couple) 때문이다. 대안으로, bean 정의는 초기화 메소드를 지정할 수 있다. XML 기반 설정의 경우, 'destroy-method' attribute를 사용한다.

```
public class ExampleBean {  
  
    public void cleanup() {  
        // do some destruction work (like releasing pooled connections)  
    }  
}
```

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

### □ Bean 성질 변화(3/4)

#### – Lifecycle Callback

- 기본 Instantiation & Destruction callback

Spring IoC container 레벨에서 기본 Instantiation & Destruction callback 메소드를 지정할 수 있다. <beans/> element의 'default-init-method', 'default-destroy-method' attribute를 사용한다.

```
<beans default-init-method="init" default-destroy-method="destroy">

    <bean id="blogService" class="com.foo.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
    </bean>

</beans>
```

### □ Bean 성질 변화(4/4)

#### – Knowing who you are

- BeanFactoryAware

org.springframework.beans.factory.BeanFactoryAware interface를 구현한 class는 자신을 생성한 BeanFactory를 참조할 수 있다.

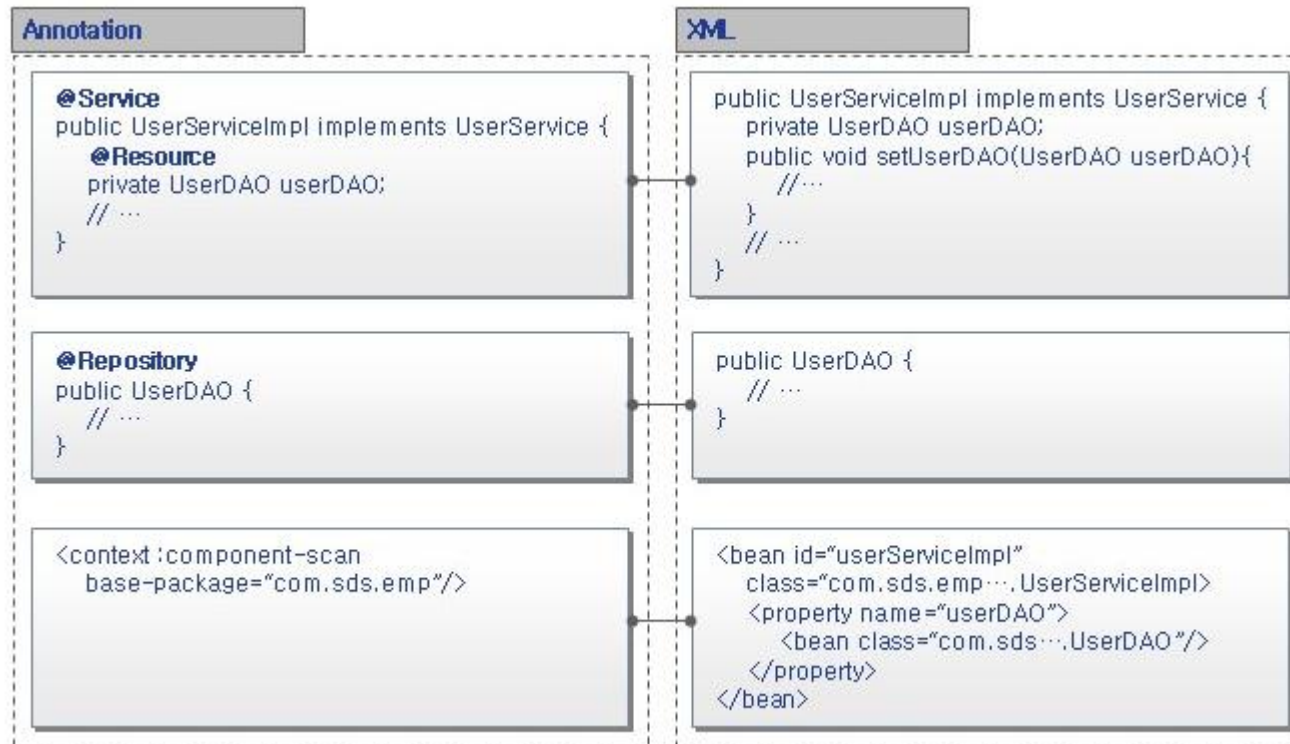
```
public interface BeanFactoryAware {  
  
    void setBeanFactory(BeanFactory beanFactory) throws BeansException;  
  
}
```

BeanFactoryAware interface를 사용하면, 자신을 생성한 BeanFactory를 알 수 있고, 프로그램적으로 다른 bean을 검색할 수 있다. 하지만 이 방법은 Spring과의 결합을 발생시키고, Inversion of Control 스타일에도 맞지 않으므로 피하는 것이 좋다. 대안으로는 org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean을 사용할 수 있다.

## LAB 201-IoC 실습(1)

### □ Annotation

- XML 설정 파일을 사용하는 대신 자바 어노테이션을 사용할 수 있음 (자바 5 이상)
- annotation의 사용으로 설정파일을 간결화하고, View 페이지와 객체 또는 메소드의 매핑을 명확하게 할 수 있다



### □ Autowiring

- Spring IoC container는 서로 관련된 Bean 객체를 자동으로 엮어줄 수 있다.  
**자동엮기(autowiring)는 각각의 bean 단위로 설정되며, 자동엮기 기능을 사용하면 <property/>나 <constructor-arg/>를 지정할 필요가 없으므로, 타이핑일 줄일 수 있다.**
- 자동엮기에는 5가지 모드가 있으며, XML 기반 설정에서는 <bean/> element의 'autowire' attribute를 사용하여 설정할 수 있다.

Scope	설명
no	자동엮기를 사용하지 않는다. Bean에 대한 참조는 <ref/> element를 사용하여 지정해야만 한다. 이 모드가 기본(default)이다.
byName	Property의 이름으로 자동엮기를 수행한다. Property의 이름과 같은 이름을 가진 bean을 찾아서 엮어준다.
byType	Property의 타입으로 자동엮기를 수행한다. Property의 타입과 같은 타입을 가진 bean을 찾아서 엮어준다. 만약 같은 타입을 가진 bean이 container에 둘 이상 존재할 경우 exception이 발생한다. 만약 같은 타입을 가진 bean이 존재하지 않는 경우, 아무 일도 발생하지 않는다. 즉, property에는 설정되지 않는다.
constructor	byType과 유사하지만, 생성자 argument에만 적용된다. 만약 같은 타입의 bean이 존재하지 않거나 둘 이상 존재할 경우, exception이 발생한다.
autodetect	Bean class의 성질에 따라 constructor와 byType 모드 중 하나를 선택한다. 만약 default 생성자가 존재하면, byType 모드가 적용된다.

- <bean/> element의 'autowire-candidate' attribute 값을 'false'로 설정함으로써, 대상 bean이 다른 bean과 자동으로 엮이는 것을 방지한다.

### □ Annotation 기반 설정(1/12)

- Spring은 Java Annotation을 사용하여 Bean 정의를 설정할 수 있다. 이 기능을 사용하기 위해서는 다음 namespace와 element를 추가해야 한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:annotation-config/>

</beans>
```



### □ Annotation 기반 설정(2/12)

#### – @Required

- @Required annotation은 setter 메소드에 적용된다. @Required annotation이 설정된 property는 <property/>, <constructor-arg/> element를 통해서 명시적으로 값이 설정되거나, autowiring에 의해서 값이 설정되어야 한다.

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Required  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
  
    // ...  
}
```

### ❑ Annotation 기반 설정(3/12)

#### – @Autowired(1/3)

- @Autowired annotation은 자동으로 역을 property를 지정하기 위해 사용한다. setter 메소드, 일반적인 메소드, 생성자, field 등에 적용된다.
- Setter 메소드

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Autowired  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
  
    // ...  
}
```

### ❑ Annotation 기반 설정(4/12)

#### – @Autowired(2/3)

- 일반적인 메소드

```
public class MovieRecommender {  
  
    private MovieCatalog movieCatalog;  
  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    public void prepare(MovieCatalog movieCatalog, CustomerPreferenceDao  
customerPreferenceDao) {  
        this.movieCatalog = movieCatalog;  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
  
    // ...  
}
```

### □ Annotation 기반 설정(5/12)

#### – @Autowired(3/3)

- 생성자 및 field

```
public class MovieRecommender {  
  
    @Autowired  
    private MovieCatalog movieCatalog;  
  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
  
    // ...  
}
```

### □ Annotation 기반 설정(6/12)

#### – @Qualifier(1/3)

- @Autowired annotation만을 사용하는 경우, 같은 Type의 Bean이 둘 이상 존재할 때 문제가 발생한다. 이를 방지하기 위해서 @Qualifier annotation을 사용하여 찾을 Bean의 대상 집합을 좁힐 수 있다. @Qualifier annotation은 field 뿐 아니라 생성자 또는 메소드의 parameter에도 사용할 수 있다.
- Field

```
public class MovieRecommender {  
  
    @Autowired  
    @Qualifier("main")  
    private MovieCatalog movieCatalog;  
  
    // ...  
}
```

### □ Annotation 기반 설정(7/12)

- @Qualifier(2/3)
  - 메소드 Parameter

```
public class MovieRecommender {  
  
    private MovieCatalog movieCatalog;  
  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    public void prepare(@Qualifier("main") MovieCatalog movieCatalog,  
                       CustomerPreferenceDao customerPreferenceDao) {  
        this.movieCatalog = movieCatalog;  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
  
    // ...  
}
```

### □ Annotation 기반 설정(7/12)

#### – @Qualifier(3/3)

- @Qualifier annotation의 값으로 사용되는 qualifier는 <bean/> element의 <qualifier/> element로 설정한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="main"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="action"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>
</beans>
```

### □ Annotation 기반 설정(8/12)

#### – @Resource

- @Resource annotation의 name 값으로 대상 bean을 찾을 수 있다. @Resource annotation은 field 또는 메소드에 사용할 수 있다.

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Resource(name="myMovieFinder")  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

- @Resource annotation에 name 값이 없을 경우, field 명 또는 메소드 명을 이용하여 대상 bean을 찾는다.

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Resource  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```



### □ Annotation 기반 설정(9/12)

#### – @PostConstruct & @PreDestroy

- @PostConstruct와 @PreDestroy는 각각 Instantiation callback, Destruction callback 메소드를 지정하기 위해 사용한다.

```
public class CachingMovieLister {

    @PostConstruct
    public void populateMovieCache() {
        // populates the movie cache upon initialization...
    }

    @PreDestroy
    public void clearMovieCache() {
        // clears the movie cache upon destruction...
    }

}
```

### □ Annotation 기반 설정(10/12)

#### – Auto-detecting components(1/3)

- Spring은 @Repository, @Service, @Controller annotation을 사용하여, 각각 Persistence, Service, Presentation 레이어의 컴포넌트로 지정하여 특별한 관리 기능을 제공하고 있다. @Repository, @Service, @Controller는 @Component annotation을 상속받고 있다. Spring IoC Container는 @Component annotation(또는 자손)으로 지정된 class를 XML Bean 정의 없이 자동으로 찾을 수 있다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:component-scan base-package="org.example"/>

</beans>
```

<context:component-scan/> element의 'base-package' attribute는 컴포넌트를 찾을 기본 package이다. '.'를 사용하여 다수의 기본 package를 지정할 수 있다.

### □ Annotation 기반 설정(11/12)

#### – Auto-detecting components(2/3)

- 이름 설정

@Component, @Repository, @Service, @Controller annotation의 name 값으로 bean의 이름을 지정할 수 있다. 아래 예제의 Bean 이름은 “myMovieLister”이다.

```
@Service("myMovieLister")
public class SimpleMovieLister {
    // ...
}
```

만약 name 값을 지정하지 않으면, class 이름의 첫문자를 소문자로 변환하여 Bean 이름을 자동으로 생성한다. 아래 예제의 Bean 이름은 “movieFinderImpl”이다.

```
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

### □ Annotation 기반 설정(12/12)

#### – Auto-detecting components(3/3)

- Scope 설정

@Scope annotation을 사용하여, 자동으로 찾은 Bean의 scope를 설정할 수 있다.

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

- Qualifier 설정

@Qualifier annotation을 사용하여, 자동으로 찾은 Bean의 qualifier를 설정할 수 있다.

```
@Component
@Qualifier("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
```

### ❑ ApplicationContext for web application(1/2)

- Spring은 Web Application에서 ApplicationContext를 쉽게 사용할 수 있도록 각종 class들을 제공하고 있다.
- Servlet 2.4 이상
  - **Servlet 2.4 specification**부터 **Listener**를 사용할 수 있다. **Listener**를 사용하여 **ApplicationContext**를 설정하기 위해서 **web.xml** 파일에 다음 설정을 추가한다.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>
</context-param>

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

**contextParam**의 '**contextConfigLocation**' 값은 **Spring XML** 설정 파일의 위치를 나타낸다.

### ❑ ApplicationContext for web application(2/2)

#### – Servlet 2.3 이하

- Servlet 2.3이하에서는 Listener를 사용할 수 없으므로, Servlet를 사용한다. web.xml 파일에 다음 설정을 추가한다.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>
</context-param>

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

## LAB 201-IoC 실습(2)

### ❑ The Spring Framework - Reference Documentation / Chapter 3. The IoC container

- <http://static.springframework.org/spring/docs/2.5.x/reference/beans.html>
- <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html>

### ❑ Inversion of Control

- <http://martinfowler.com/bliki/InversionOfControl.html>

### ❑ Inversion of Control Containers and the Dependency Injection pattern

- <http://martinfowler.com/articles/injection.html>