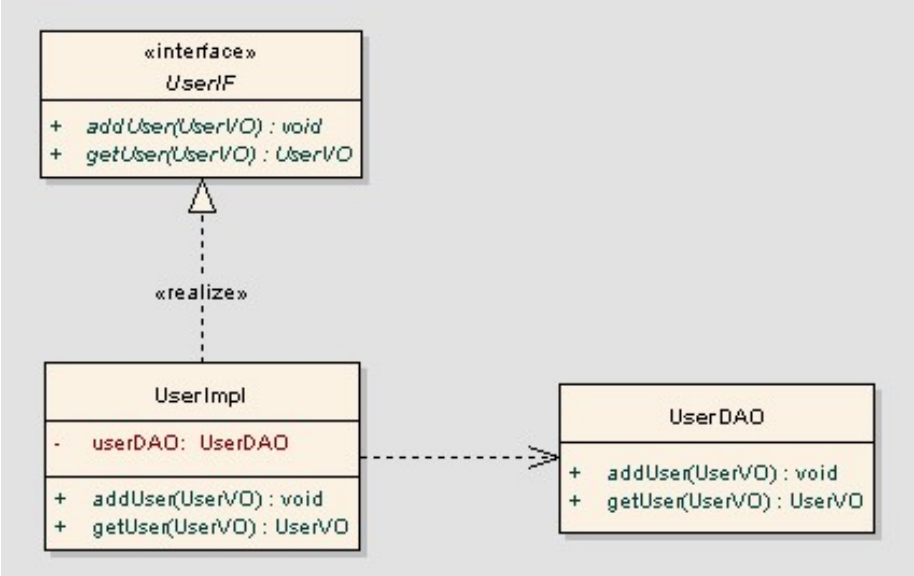
01 DI(Dependency Inversion) 개념

1) Dependency 관계

Dependency 관계란 Bean과 Bean의 결합관계다. 즉 하나의 Bean에서 다른 Bean의 변수나 메소드를 이용해야 한다면 이용하고자 하는 Bean에 대한 객체생성과 생성된 객체의 레퍼런스 정보가 필요하다.



위 그림에서 UserImpl 객체는 UserDAO 객체의 메소드를 이용해서 기능을 수행한다. 따라서 UserImpl 객체는 UserDAO 타입의 userDAO 변수를 가지고 있으며, userDAO 변수는 UserDAO 객체를 참조하고 있어야 하기 때문에 다음과 같은 객체생성 관련된 코드가 필요하다.

UserImpl.java

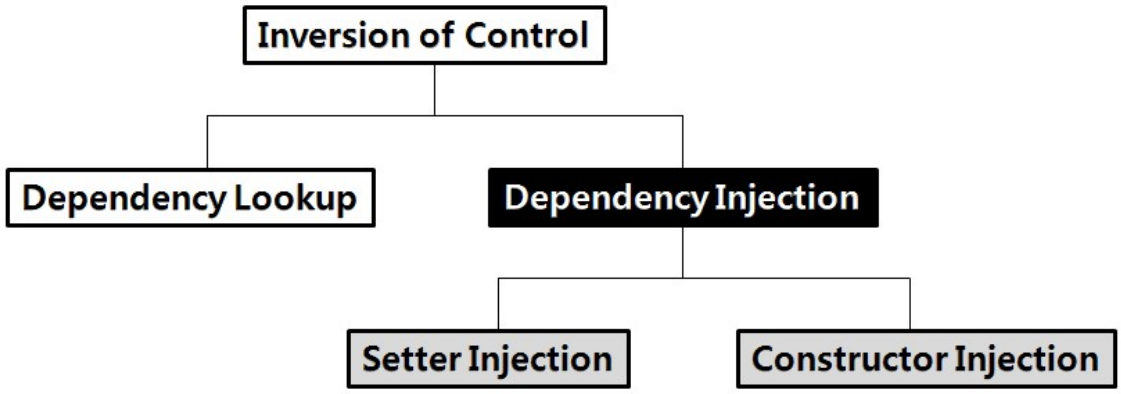
```
public class UserImpl implements UserIF {
        private UserDAO userDAO;

        public void addUser(UserVO vo) {
                userDAO = new UserDAO();
                userDAO.addUser(vo);
        }
}
```

하지만 Spring 프레임워크에서는 Bean의 라이프사이클을 프레임워크 차원에서 관리해줌으로 개발코드에 UserDAO 객체를 직접 생성하지 않아도 된다. 결국 Dependency 관리란 Spring 프레임워크에서 프레임워크가 관리하는 Bean을 다른 Bean에서 사용할 수 있도록 설정해주는 역할까지 대행해주기 때문에 어떤 Bean들의 의존관계를 프로그램 코드에 명시하지 않아도 되기 때문에 두 Bean간의 결합도가 최소화 되어 유지보수가 용이해진다.

Dependency의 구현방법에는 두 가지가 있다.
첫째는 Dependency Lookup이다. 이 것은 컨테이너가 callback을 통해서 제공하는 Lookup Context를 이용해서 필요한 리소스나 오브젝트를 얻는 방식이다. EJB와 Apache Avalon의 구현방법이다.

둘째는 Dependency Injection이다. 비즈니스 오브젝트에 Lookup 코드를 사용하지 않고 컨테이너가 직접 의존구조를 오브젝트에 설정할 수 있도록 지정해주는 방법이다. 이 것은 다시 Setter Injection과 Constructor Injection으로 나뉜다.



2) Dependency Lookup

Spring의 Dependency Lookup을 살펴보기 전에 EJB 쪽에서 제공되는 Lookup 방식에 대해 살펴보도록 하자. EJB Bean은 EJB 컨테이너에 의해 관리된다. 즉 EJB Bean에 대한 생성 및 소멸을 서버가 관리해준다. Spring으로 이야기 하면 IoC기능을 제공해준다고 볼 수 있다.

이렇게 EJB 서버에 의해 관리되는 Bean을 이용하기 위해서는 아래와 같은 코드가 사용된다.

```
InitialContext cxt = new InitialContext();
HelloHome helloHome = (HelloHome) cxt.lookup("HelloEJB");
Hello hello = helloHome.create();
```

위 코드는 JNDI 프로그램 코드이다. JNDI는 Naming 서버 연동을 위한 자바 표준 스팩이며, EJB 같은 분산 환경 프로그램에서 주로 이용된다. 분산 환경에서의 객체 획득 시 Naming 서버를 이용하여 획득하게 된다. 즉 Naming 서버에 이름값만 넘겨주어 해당 이름으로 등록된 객체를 획득하는 것이다. 이 방식이 Lookup 방식이다. 즉 이름으로 사용하고자 하는 객체를 Lookup 하는 방식이다. 이렇게 하면 실제 EJB 객체를 가지는 서버와 이용하고자 하는 곳과 직접 결합되지 않아도 된다는 이점을 얻을 수 있다.

Spring 프레임워크와 EJB가 Dependency Lookup을 제공해주는 방식은 동일하다. 이용하고자 하는 곳에서 컨테이너에게 이름값으로 Lookup하여 컨테이너가 제공해주는 Bean을 얻어 사용하는 방식이다.

하지만 EJB에서는 Naming 서버가 반드시 있어야 한다. 즉, Naming 서버를 이용할 수 없는 환경에서는 사용이 불가능해진다. 하지만 Spring 프레임워크는 분산환경이 아니기 때문에 Spring의 Dependency Lookup을 이용하는데 Naming 서버를 필요로 하지 않으므로 EJB 보다 좀더 간편하게 이용할 수 있다.

Spring에서 Dependency Lookup을 이용하기 위해서는 해당 Bean이 XML 설정파일에 등록이 되어 있어야 하고 컨테이너 객체의 getBean()이라는 메소드를 이용한다. 즉 BeanFactory나 ApplicationContext에서 제공해주는 getBean() 메소드를 호출하면서 얻고자 하는 Bean의 이름값을 넘겨주면 된다.

```
UserService userService = (UserService)factory.getBean("userService");
```

위의 코드는 Spring Container가 관리하는 Bean 중에서 userService 라는 이름의 Bean을 획득하는 구문이다. 기본으로 Object 타입으로 넘기기 때문에 캐스팅 해서 사용하고 있다.

Spring의 Dependency Lookup이 EJB처럼 Naming 서버를 필요로 하지 않기 때문에 EJB보다 단순해진 측면은 있지만 여전히 Spring 종속적인 factory.getBean() 등의 코드가 쓰일 수 밖에 없기 때문에 Spring 내부 Bean이 다른 Spring Bean을 획득할 때는 사용하지 않는다.

Dependency Lookup은 오브젝트간에 결합도를 떨어뜨릴 수 있다는 측면에서 장점이 있기는 하다. 하지만 Dependency Lookup에 의해서 얻어진 오브젝트는 컨테이너 밖에서 실행할 수 없다든지, Bean의 변경 시 오브젝트 내에 변경에 대한 부분을 반영해야 하므로 일일이 수정해야 하며, 테스트하기 매우 어렵고, Strong typed가 아니므로 Object로 받아서 매번 Casting해야 한다는 문제점이 있다.

3) Dependency Injection

① Dependency Injection 이란?

Dependency Injection은 각 Bean 사이의 의존관계 설정을 XML 설정파일에 등록된 정보를 바탕으로 컨테이너가 자동적으로 연결해 주는 것이다. 즉 프로그램 코드에서 직접 필요한 Bean을 생성하지 않고 컨테이너가 자체적으로 해당 Bean에서 필요한 객체를 넘겨줘서 사용하는 방식이다.

Dependency Injection을 이용하면 프레임워크의 의존적인 코드가 작성되지 않기 때문에 프레임워크에 종속적인 코드를 작성하지 않아도 되는 이점이 있다. 또 Dependency 설정을 바꾸고 싶을 경우 프로그램 코드를 수정하지 않고 XML 설정파일 수정만으로 변경사항을 적용시킬 수 있음으로 유지보수성도 개선된다.

하지만 Dependency Injection은 Spring 내부에 등록된 Bean끼리 의존성 설정에서만 사용할 수 있으며 Spring 프레임워크의 관리를 받지 않는 외부 클래스에서는 사용할 수 없다.

② Constructor Injection

Spring에서는 Bean을 초기화 시키기 위해 default 생성자를 호출한다. 하지만 Constructor Injection을 사용하면 생성자의 매개변수로 Dependency 설정 내용을 넘겨주는 것임으로 Bean 클래스 작성시 매개변수를 가지는 생성자를 준비해야 한다.

UserVO.java

```
public class UserVO {
        private String userName;
        public UserVO(String userName) {
                this.userName = userName;
        }
}
```

이를 위한 XML 설정파일은 다음과 같다.

applicationContext.xml

```
<bean id="userVO" class="com.multicampus.biz.user.vo.UserVO">
        <constructor-arg value="홍길동" />
</bean>
```

생성자가 매개변수로 객체 타입을 받아들이는 경우에는 다음과 같이 설정하면 된다.

UserServiceImpl.java

```
public class UserServiceImpl implements UserService {
	private UserDAO userDAO;

	public UserServiceImpl(UserDAO userDAO) {
		this.userDAO = userDAO;
	}
}
```

UserServiceImpl 클래스를 작성하면서 UserDAO 객체를 매개변수로 가지는 생성자를 하나 준비했다. 이 생성자는 Spring 컨테이너에 의해서 호출될 것이며, 호출되는 시점에 UserDAO 객체를 매개변수로 넘겨준다. 이를 위한 XML 설정파일은 다음과 같다.

applicationContext.xml

```
<bean id="userService" class="com.multicampus.biz.user.impl.UserServiceImpl">
	<constructor-arg ref="userDAO" />
</bean>

<bean id="userDAO" class="com.multicampus.biz.user.impl.UserDAO" />
```

③ 매개변수 Type 매핑

Constructor Injection을 이용할 때 생성자에 하나의 값만 설정하는 게 아니라 여러 값을 한꺼번에 설정하고자 하는 경우가 있다. 이 경우는 아래처럼 <constructor-arg> 태그를 여러 개 사용하면 된다.

UserVO.java

```
public class UserVO {
	private String userName;
	private int age;
	public UserVO(String userName, int age) {
		this.userName = userName;
		this.age = age;
	}
}
```

이를 위한 XML 설정파일은 다음과 같다.

applicationContext.xml

```
<bean id="userVO" class="com.multicampus.biz.user.vo.UserVO">
        <constructor-arg value="홍길동" />
        <constructor-arg value="17" />
</bean>
```

위의 경우는 <constructor-arg> 태그를 두 개를 설정한 경우이다. 이럴 경우 해당 Bean에는 매개변수를 두 개 가지는 생성자가 정의되어 있으면 된다.

그런데 생성자가 오버로딩 되어있는 클래스의 경우, 어느 생성자를 호출해야 할지 불분명한 경우가 생길 수도 있다. 이를 위해 Spring에서는 매개변수의 type을 정확히 지정하거나 몇 번째 매개변수인지를 지정할 수 있는 방법을 제공해준다.

applicationContext.xml

```
<bean id="userVO" class="com.multicampus.biz.user.vo.UserVO">
        <constructor-arg type="java.lang.String" value="홍길동" />
        <constructor-arg type="int" value="17" />
</bean>
```

위의 코드를 보면 UserVO Bean에 Constructor Injection을 설정하는데 두 개의 매개변수를 설정하였다. type 속성을 설정하여 해당 데이터가 어떤 타입인지를 지정했다.

Constructor Injection을 이용하면서 index 속성을 이용하여 해당 값이 몇 번째 매개변수인지를 정확하게 지정할 수 있다. 이때 index 는 0부터 시작한다.

applicationContext.xml

```
<bean id="userVO" class="com.multicampus.biz.user.vo.UserVO">
        <constructor-arg index="0" value="홍길동" />
        <constructor-arg index="1" value="17" />
</bean>
```

④ Setter Injection

Constructor Injection은 생성자를 이용하여 Dependency를 설정하는 방법이다. 이에 비해 Setter Injection은 Bean이 가지고 있는 Setter 메소드를 호출하여 Dependency를 설정하는 방법이다.

UserServiceImpl.java

```
public class UserServiceImpl implements UserService {
        private UserDAO userDAO;

        public void setUserDAO(UserDAO userDAO) {
                this.userDAO = userDAO;
        }
}
```

UserServiceImpl 클래스에는 UserDAO 타입의 userDAO 변수가 있고, 이 변수에 UserDAO 객체를 Dependency Injection 하기 위해 setUserDAO() 메소드가 정의되어있다.

setUserDAO() 메소드는 Spring 프레임워크에 의해 호출되며 호출되는 시점은 Spring에 의해 Bean 객체가 생성된 직후에 호출된다. 즉 어떠한 라이프사이클 관련 메소드 보다 먼저 setUserDAO() 메소드가 호출된다.

위의 Setter Injection을 이용하기 위한 XML 설정파일은 다음과 같이 정의되어 있어야 한다.

applicationContext.xml

```
<bean id="userService" class="com.multicampus.biz.user.impl.UserServiceImpl">
        <property name="userDAO" ref="userDAO" />
</bean>

<bean id="userDAO" class="com.multicampus.biz.user.impl.UserDAO" />
```

Setter Injection을 위해서는 <property> 태그를 사용해야 하며 <property> 태그의 name 값이 호출하고자하는 메소드 명이 된다. 즉 name이 userDAO 라고 설정되어 있으면 호출되는 메소드는 setUserDAO()가 된다. 변수 앞에 set을 붙이고 첫 글자를 대문자로 바꾼 것이 호출할 메소드 이름이 된다.

Constructor Injection과 마찬가지로 Setter 메소드를 호출하면서 다른 Bean을 넘기기 위해서는 <ref> 태그를 사용하고 기본형 데이터를 넘기기 위해서는 <value> 태그를 사용한다.
ref와 value는 별도의 태그를 사용해도 되고 bean 의 속성으로 설정하는 것도 가능하다.

⑤ Setter Injection과 Constructor Injection 동시 사용

Dependency를 이용할 때 Constructor Injection 과 Setter Injection 을 동시에 한 Bean에 설정하여 사용하는 것도 가능하다.

UserServiceImpl.java

```
public class UserServiceImpl implements UserService {
        private UserDAO userDAO;
        private int userCount;
        private String beanName;

        public UserServiceImpl(int userCount, String beanName){
                this.userCount = userCount;
                this.beanName = beanName;
        }
        public void setUserDAO(UserDAO userDAO) {
                this.userDAO = userDAO;
        }
}
```

위 클래스에 대한 XML 설정파일은 다음과 같다.

applicationContext.xml

```
<bean id="userService" class="com.multicampus.biz.user.impl.UserServiceImpl">
        <constructor-arg>
                <value>12<value>
        </constructor-arg>
        <constructor-arg>
                <value>userService</value>
        </constructor-arg>
        <property name="userDAO" ref="userDAO" />
</bean>
```

userService Bean 설정에 <constructor-arg> 태그와 <property> 태그가 같이 사용되고 있다. 이 경우 UserServiceImpl 클래스의 매개변수를 2개 가지고 있는 생성자를 호출하면서 Bean을 초기화 시키고 Bean 초기화 직후 setUserDAO() 메소드를 호출해준다.

⑥ Setter Injection vs Constructor Injection

DI의 경우에는 또 두 가지로 나뉘어 지는데 어떤 것을 사용하더라도 상관은 없다. 선택사항 이기는 하지만 생성자를 이용할 것인가, Setter 메소드를 이용할 것인가의 차이점이 있다.

Injection 유형	설명
Setter	사용하려는 Bean이 실제 기능 구현 시 사용될 수도 있고, 사용되지 않을 수도 있을 때 이 방식을 사용하는 것이 좋다.
Constructor	사용하려는 외부 Bean이 자신의 내부에서 반드시 사용되어야 하고, 이것을 사용해야 지만 자신이 객체화되어 올바른 기능을 수행할 수 있는 경우에 사용한다.

실제 대부분의 경우 Setter Injection이 사용되고 있으며 이는 Constructor Injection을 사용하는 것 자체가 객체지향 적으로 부담이 되기 때문이다.

4) 집합객체 설정

① 집합객체

생성자나 Setter 메소드가 실행될 때 List나, Map과 같은 Collection 타입으로 설정하고자 할 경우가 있을 것이다. Spring에서는 이를 위해 Collection Mapping을 지원하고 있다.

Element	Java Type
<list>	java.util.List, 배열Type
<set>	java.util.Set
<map>	java.util.Map
<props>	java.util.Properties

Spring에서 집합객체의 설정을 위해 <list>, <set>, <map>, <props> 태그를 지원한다.

② List 타입 매핑

java.util.List 타입의 Collection 객체는 <list> 태그를 사용하여 설정한다.

```
public class CollectionBean {
        private List<String> addressList;
        public void setAddressList(List<String> addressList){
                this.addressList = addressList;
        }
}
```

```
<bean id="collectionBean" class="com.multicampus.ioc.injection.CollectionBean">
        <property name="addressList">
                <list>
                        <value>서울시 강남구 역삼동</value>
                        <value>서울시 성동구 성수동</value>
                </list>
        </property>
</bean>
```

위의 설정은 CollectionBean 클래스가 setAddressList() 메소드를 가지고 있어야 한다는 것을 의미한다. 이 setAddressList() 메소드는 List를 매개변수로 받아들이도록 만들어져 있어야 한다. 그럴 경우 두 개의 문자열 객체를 가지고 있는 List 객체가 매개변수로 전달된다.

③ Set 타입 매핑

중복 값을 허용하지 않는 집합 객체를 표현할 때는 java.util.Set 이라는 Collection 타입을 사용한다. Collection 객체는 <set> 태그를 사용하여 설정할 수 있다.

```
public class CollectionBean {
        private Set<String> addressList;
        public void setAddressList(Set<String> addressList){
                this.addressList = addressList;
        }
}
```

```
<bean id="collectionBean" class="com.multicampus.ioc.injection.CollectionBean">
        <property name="addressList">
                <set value-type="java.lang.String">
                        <value>서울시 강남구 역삼동</value>
                        <value>서울시 성동구 성수동</value>
                </set>
        </property>
</bean>
```

위의 예는 setAddressList() 라는 메소드가 Set Collection 객체를 매개변수로 받아들여야 한다는 것을 의미한다. 여기에서 <value>는 중복된 값이 아니어야 할 것이다. 그리고 <value> 태그 대신에 <ref bean="address">와 같이 별도로 선언된 객체를 참조할 수도 있다.

④ Map 타입 매핑

key=value 형태의 집합 객체를 표현할 때는 java.util.Map 이라는 Collection 타입을 사용한다. Collection 객체는 <map> 태그를 사용하여 설정할 수 있다.

```
public class HandlerMapping {
        private Map<String, Controller> mappings;
        public void setMappings(Map<String, Controller> mappings){
                this.mappings = mappings;
        }
}
```

```
<bean id="handlerMapping" class="com.multicampus.ioc.injection.HandlerMapping">
        <property name="mappings">
                <map>
                        <entry>
                                <key><value>login</value></key>
                                <ref bean="loginController"/>
                        </entry>
                        <entry>
                                <key><value>logout</value></key>
                                <ref bean="logoutController"/>
                        </entry>
                </map>
        </property>
</bean>

<bean id="loginController" class="com.multicampus.system.LoginController" />
<bean id="logoutController" class="com.multicampus.system.LogoutController /">
```

위의 예는 setMappings() 메소드가 Map 인터페이스 타입의 객체를 매개변수로 받아들여야 한다는 것을 의미한다. <entry> 객체의 key 속성값은 Map 객체의 key값이 되며 <value> 객체의 내용은 Map객체의 value로 지정된다.

⑤ Properties 타입 매핑

key=value 형태의 집합 객체를 표현할 때는 java.util.Properties 라는 Collection 타입을 사용한다. Collection 객체는 <map> 태그를 사용하여 설정할 수 있다.

```
public class HandlerMapping {
        private Properties mappings;
        public void setMappings(Properties mappings){
                this.mappings = mappings;
        }
}
```

```
<bean id="handlerMapping" class="com.multicampus.ioc.injection.HandlerMapping">
        <property name="mappings">
                <props>
                        <prop key="/login.do">loginController</prop>
                        <prop key="/logout.do">logoutController</prop>
                </ props >
        </property>
</bean>

<bean id="loginController" class="com.multicampus.system.LoginController" />
<bean id="logoutController" class="com.multicampus.system.LogoutController /">
```

위의 예는 setMappings() 메소드가 java.util.Properties 타입의 객체를 매개변수로 전달받는다는 것을 의미한다.

5) 의존 관계 자동 설정

① autowire 속성

Spring 프레임워크에서는 XML 기반의 Spring 설정파일을 간단하고 쉽게 작성할 수 있도록 Bean 설정을 자동으로 처리할 수 있는 방법을 제공해준다. 즉 XML 설정파일에 일일이 Dependency 관계를 설정하지 않고 Spring 프레임워크에서 자체 판단해서 Bean 객체를 생성하고 Dependency Injection을 제공해주는 방식이다.

기존에 Bean 들의 Dependency 관계 설정은 다음과 같이 처리된다.

applicationContext.xml

```
<bean id="userService" class="com.multicampus.biz.user.impl.UserServiceImpl">
        <property name="userDAO" ref="userDAO" />
</bean>

<bean id="userDAO" class="com.multicampus.biz.user.impl.UserDAO" />
```

이 설정을 autowire 속성을 이용해서 설정하면 다음과 같다.

applicationContext.xml

```
<bean id="userService" class="com.multicampus.biz.user.impl.UserServiceImpl"
autowire="byName" />

<bean id="userDAO" class="com.multicampus.biz.user.impl.UserDAO" />
```

UserServiceImpl Bean에 UserDAO Bean을 Setter Injection으로 설정하고자 한다. 하지만 일반적인 경우라면 <property> 태그를 이용하여 설정하였을 것이지만 위의 설정을 보면 Setter Injection에 대한 설정은 보이지 않는다. 대신 autowire 속성이 설정되고 byName 이라고 지정되어 있다.

위와 같이만 설정해도 UserServiceImpl 객체에 userDAO Bean을 설정해주는데 이런 기능을 의존관계 자동 설정이라고 한다. 즉 XML 설정파일에 Bean들에 대한 Dependency를 설정하지 않아도 프레임워크에서 자동으로 처리해주는 것이다.

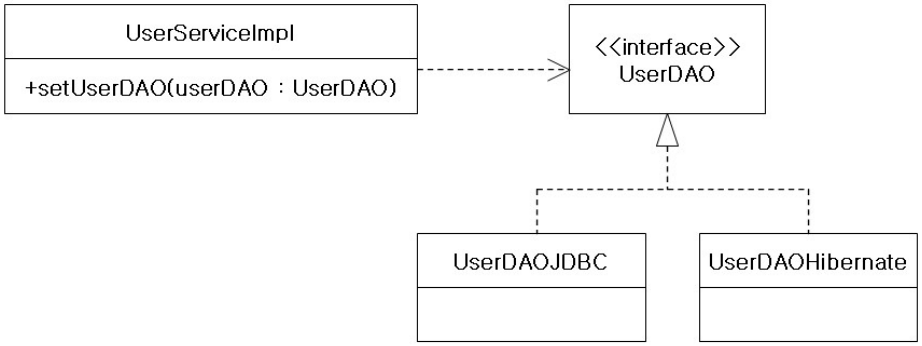
autowire 속성은 byName 이외에 다음과 같은 값들을 가질 수 있다.

속 성	설 명
no	autowire 기능을 사용하지 않겠다는 설정이며 기본값이다.
byName	Bean에 설정된 property명과 동일한 이름의 bean을 찾아 설정한다.
byType	해당 property 타입의 Bean을 찾아 설정한다. 만약 없다면 설정은 안되며 두 개 이상 존재하면 에러가 발생한다.
constructor	byType과 동일하지만 생성자를 이용하는 경우이다.
autodetect	constructor와 byType이 동시에 적용된다.

② byType 사용

byType은 프로퍼티의 타입과 동일한 타입을 갖는 Bean 객체를 검색해서 프로퍼티 값으로 설정한다. 그런데 이 byType 설정에 대해서는 주의할 부분이 있는데 동일한 타입의 Bean이 두 개 이상 존재할 때, 예외가 발생되기 때문에 주의해야 한다.

다음과 같은 구조의 Bean 설정을 생각해보자.



UserServiceImpl은 UserDAO Type의 변수를 멤버로 갖는다. 이럴 경우 byType을 이용한 XML 설정파일은 다음과 같이 작성될 것이다.

applicationContext.xml

```
<bean id="userService" class="com.multicampus.biz.user.impl.UserServiceImpl"
autowire="byType" />

<bean id="jdbc" class="com.multicampus.biz.user.impl.UserDAOJDBC" />
<bean id="hibernate" class="com.multicampus.biz.user.impl.UserDAOHibernate" />
```

위와 같이 설정하면 Spring Container는 UserServiceImpl 클래스에 있는 userDAO 프로퍼티에 어떤 객체를 Injection할지 판단할 수 없다. 따라서 위의 경우에는 예외가 발생된다. 이렇게 byType 방식을 이용하여 Bean 대한 의존성을 설정할 경우, Property 타입과 동일한 타입의 Bean 객체를 하나만 설정해야 한다는 것을 의미한다.

③ constructor 사용

constructor는 byType 방식과 동일하게 타입을 이용해서 Dependency를 설정한다. 단, 프로퍼티가 아닌 생성자의 매개변수 타입을 사용한다는 점이 다르다.

```
public class UserServiceImpl implements UserService {
        private UserDAO userDAO;
        public UserServiceImpl(UserDAO userDAO){
                this.userDAO = userDAO;
        }
}
```

```
<bean id="userService" class="com.multicampus.ioc.user.impl.UserServiceImpl"
autowire="constructor" />

<bean id="jdbc" class="com.multicampus.ioc.user.impl.UserDAOJDBC />
```

이렇게 설정하면 UserServiceImpl 클래스의 생성자가 호출될 때, 매개변수로 UserDAO 타입의 객체인 UserDAOJDBC 객체가 전달된다. constructor 방식을 사용하는 것 역시 byType을 사용하는 것과 동일하게 매개변수에 해당하는 동일 타입의 Bean이 여러 개 등록되는 경우에 예외가 발생된다.

④ autodetect 사용

autodetect를 사용하는 경우에는 constructor 방식을 먼저 적용하고, constructor 방식을 사용할 수 없는 경우에는 byType을 적용하여 Dependency 설정을 처리한다.

02 Annotation 기반 설정

1) Annotation 설정

① Namespace 선언

Spring 프레임워크를 이용하면 대부분의 프레임워크들이 그렇듯이 XML 설정부분이 중요하다. XML 선언만으로 Bean 의 라이프사이클이 관리가 되고 Dependency 가 관리된다는 것은 매우 유용한 기능이다. 하지만 XML 파일의 과도한 설정에 대한 부담감 역시 크다.

이러한 이유에서 대부분의 프레임워크에서 Annotation 을 이용한 설정을 지원해 주고 있으며 Spring 프레임워크에서도 이를 지원해 주기 위해서 여러 가지 Annotation 을 제공해 주고 있다.

Annotation을 사용하여 Bean을 정의하면 XML 설정 파일에 따로 Bean 정의를 명시하지 않아도 Spring 컨테이너가 Bean을 인식하고 관리할 수 있다. 단, Annotation 설정을 추가하기 위해서는 다음과 같이 XML 설정파일의 루트 엘리먼트인 <beans>에 Context 관련 네임스페이스 선언과 스키마 문서의 위치를 등록해야 한다.

applicationContext.xml

```
<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/contxt/springcontext-3.0.xsd">
</beans>
```

② Composnent 스캔

XML 설정파일에 Context 관련 네임스페이스를 등록했으면 이제부터 “context”라는 접두사를 이용하여 Context 관련 태그들을 사용할 수 있다. XML 설정 파일에 Annotation이 설정된 클래스들을 Bean으로 등록하기 위해서는 <context:component-scan />이라는 태그를 정의해야 한다. 이 설정을 추가하면 Spring 컨테이너는 클래스패스 상에 존재하는 클래스들을 스캔 하여 Annotation이 정의된 클래스들 Bean으로 인식하고 자동으로 등록한다.

applicationContext.xml

```
<beans …생략…>
        <context:component-scan base-package="com.multicampus.biz" />
</beans>
```

③ <bean> 과 관련된 Annotation

클래스를 XML 파일에 <bean> 태그로 등록하여 Spring Framework에 의해 관리되는 클래스로 등록하는 것을 대체하기 위한 Annotation은 @Service, @Component다. 클래스 선언 부분에 @Service나 @Component Annotation을 설정해줌으로써 Spring 프레임워크에서는 해당 클래스를 빈으로 관리해준다. @Repository 역시 동일한 역할을 해주지만 DAO 같은 데이터 접근 로직을 처리해주는 클래스를 위한 Annotation으로 Persistence Layer에서 발생한 Exception에 대한 Transaciton 처리부분이 추가된 Annotation이다.

다음은 UserServiceImpl 클래스를 Bean으로 등록하는 일반적인 XML 선언과 Annotation을 사용한 예이다.

```
<bean id="userService" class="com.multicampus.biz.user.UserServiceImpl" />
```

위에서 선언된 Bean 설정과 동일한 설정을 Annotation 기반으로 다음과 같이 할 수 있다.

```
@Service("userService")
public class UserServiceImpl implements UserService {
}
```

이 경우에 해당되는 Bean을 찾기 위해서는 속성으로 정의한 name을 활용해야 한다.

```
String resource = "spring/ioc/applicaitonContext.xml";
private ApplicationContext cxt = new ClassPathXmlApplicationContext(resource);
UserService userService = (UserService)ctx.getBean("userService");
```

2) Dependency Injection 설정

① Dependency Injection 관련 Annotation

Spring에서 의존성 주입을 지원하는 Annotation으로는 @Autowired, @Qualifier, @Resource가 있다.

Annotation	설명
@Autowired	Spring 프레임워크에서 지원하는 Dependency 정의 용도의 Annotation으로, Spring 프레임워크에 종속적이긴 하지만 정밀한 Dependency Injection이 필요한 경우에 유용하다.
@Qualifier	동일한 타입의 Bean 객체가 여러 개 있는 경우에 특정 Bean 객체를 선택적으로 주입할 때 사용한다.
@Resource	JSR-250 표준 Annotation으로 Spring Framework 2.5 이후 버전부터 지원하는 Annotation으로서 특정 프레임워크에 종속되지 않는 Annotation이다.

② @Autowired Annotation

@Autowired는 Spring2.5 버전 이후에 추가된 기능으로서 타입을 기반으로 의존관계에 있는 객체를 주입하게 된다.

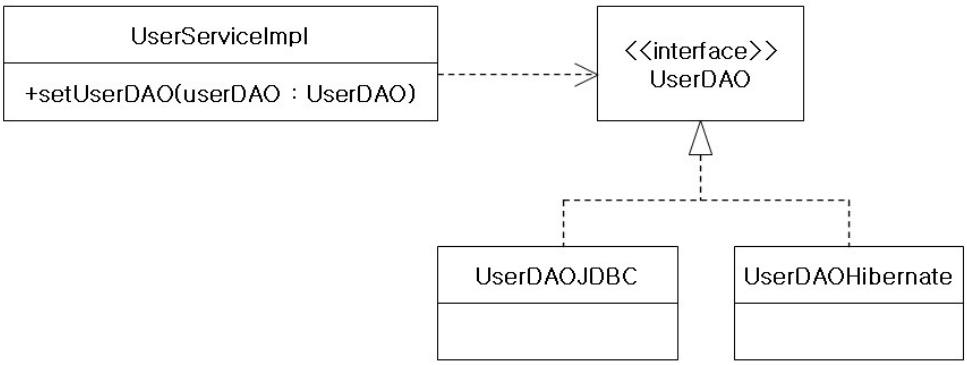
UserServiceImpl 객체와 UserDAO 객체가 의존관계에 있을 경우, 다음과 같이 설정한다.

UserServiceImpl.java

```
@Service("userService")
public class UserServiceImpl implements UserService {
        @Autowired
        private UserDAO userDAO;
}
```

위와 같이 설정하면 자바 클래스에는 Setter 메소드나 생성자는 필요 없다. 그리고 XML 설정파일 역시 아무런 설정을 필요로 하지 않는다.

그러나 @Autowired Annotation을 활용한 Dependency Injection은 의존관계에 있는 객체의 타입을 기준으로 의존성을 주입하기 때문에 UserDAO 인터페이스를 구현한 클래스가 두 개 이상인 경우에는 어떤 객체를 의존성 주입해야 할지 판단할 수 없으므로 에러를 발생시키게 된다.



이때 이런 문제를 해결하기 위해서 사용하는 Annotation이 @Qualifier Annotation이다.

UserServiceImpl.java

```
@Service("userService")
public class UserServiceImpl implements UserService {
        @Autowired
        @Qualifier("userDAOJDBC")
        private UserDAO userDAO;
}
```

@Qualifier를 사용하여 의존성을 주입할 경우, 의존성 주입할 Bean 객체의 이름을 지정해야 한다. 위 설정은 UserServiceImpl 객체가 “userDAOJDBC” 라는 이름의 Bean 객체를 의존성주입하는 설정이다.

③ @ Resource Annotation

@Resource annotation은 Bean name을 지정하여 Dependency Injection을 하고자 하는 경우에 사용한다. @Resource는 name이라는 속성을 가지고 있어서, Spring 컨테이너가 @Resource로 정의된 요소에 의존성 주입하기 위한 Bean을 검색할 때, name 속성에 지정한 이름을 검색할 Bean Name으로 사용한다.

```
@Service("userService")
public class UserServiceImpl implements UserService {
    @Resource(name="userDAO")
    private UserDAO userDAO;
}
```