

## 7. DI(Dependency Injection)

DI(Dependency Injection)란 “의존성 주입”으로 스프링 컨테이너가 지원하는 핵심 개념이며, DI는 객체 사이의 의존 관계를 객체 자신이 아닌 스프링 컨테이너가 수행한다. **웹 애플리케이션에서 구성 요소간의 종속성을 소스 코드에서 설정하지 않고, 외부의 설정 파일 등을 통해 주입하도록 하는 설계 패턴이다.** IoC 컨테이너는 어떤 클래스가 필요로 하는 인스턴스를 자동으로 생성/취득하여 연결시켜주는 역할을 한다. IoC 컨테이너가 인스턴스를 자동 생성하게 하려면 설정 파일에서 해당 클래스 정보와 설정 메타 정보를 설정해야만 한다. 스프링은 설정 파일이나 어노테이션으로 객체간의 의존 관계를 설정하고 클래스 간에 DI의 의존성 주입 방법을 사용한다.

### 7.1 스프링의 의존성 관리 방법

스프링 프레임워크의 가장 중요한 특징은 객체의 생성과 의존관계를 컨테이너가 자동으로 관리한다는 점이다. 이것이 바로 스프링 IoC(제어의 역행)의 핵심 원리이기도 하다. 스프링은 IoC(제어의 역행)를 다음 두 가지 형태로 지원한다.

- **Dependency Lookup**
- **Dependency Injection**

이 중에서 컨테이너가 애플리케이션 운용에 필요한 객체를 생성하고 클라이언트는 컨테이너가 생성한 객체를 검색하여 사용하는 방식을 Dependency Lookup이라고 한다. Dependency Lookup은 우리가 지금까지 컨테이너를 사용해왔던 방법이다. 하지만 Dependency Lookup은 실제 애플리케이션 개발 과정에서는 사용하지 않으며, 대부분 Dependency Injection을 사용하여 개발한다.

Dependency Injection은 객체 사이의 의존관계를 스프링 설정 파일에 등록된 정보를 바탕으로 컨테이너가 자동으로 처리해 준다. 따라서 의존성 설정을 바꾸고 싶을 때 프로그램 코드를 수정하지 않고 스프링 설정 파일 수정만으로 변경 사항을 적용할 수 있어서 유지보수가 향상된다.

Dependency Injection은 컨테이너가 직접 객체들 사이에 의존관계를 처리하는 것을 의미하며, 이것이 다시 Setter 메서드를 기반으로 하는 **세터 인젝션(Setter Injection)**과 생성자를 기반으로 하는 **생성자 인젝션(Constructor Injection)**으로 나뉜다.

설정 메타 정보의 주입 방법은 2가지가 있다.

생성자 기반의 주입	생성자를 통해 의존 관계를 설정한다. XML 설정 파일에서 <bean>의 하위 요소로 <constructor-arg>를 사용한다.
Setter 기반의 주입	setter 메서드로 클래스 사이의 의존관계를 설정한다. XML 설정 파일에서 <bean>의 하위 요소로 <property>를 사용한다

#### (1) 생성자 기반의 DI

생성자 기반의 DI는 의존 관계를 나타내는 다수의 인수로 생성자를 호출하는 컨테이너에서 클래스간의 의존관계를 설정하여 인스턴스를 참조할 수 있도록 한다. 생성자 기반의 DI는 <bean>의 하위 요소로 <constructor-arg> 속성이 있다.

속성	설명
index	Constructor의 몇 번째의 인수에 값을 전달할 것인지를 지정
type	Constructor의 어느 데이터형의 인수에 값을 전달할 것인지를 지정
ref	자식 요소 <ref bean = "bean 명" /> 대신 사용 가능

value	자식 요소 <value>값</value> 대신 사용 가능
-------	---------------------------------

- <constructor-arg> 요소의 ref 속성으로 빈 객체를 전달하는 DI의 예이다.

```
<bean id="bbsService" class="com.service.BbsService">
  <constructor-arg ref="bbsDAO" />
</bean>
```

- <constructor-arg> 요소의 type 속성으로 생성자 인수의 타입을 지정한다.

```
<bean id="calcBean" class="com.util.CalcBean">
  <constructor-arg type="int" value="100" />
</bean>
```

- <constructor-arg> 요소의 index 속성으로 생성자 인수를 명시적으로 지정한다.

```
<bean id="calcBean" class="com.util.CalcBean">
  <constructor-arg index="0" value="10" />
  <constructor-arg index="1" value="20" />
</bean>
```

## (2) Setter 기반의 DI

Setter 기반의 DI는 빈을 인스턴스화 하기 위해 인수가 없는 생성자나 인수가 없는 static 팩토리 메서드를 호출한 뒤에 빈의 setter 메서드를 호출하여 클래스간의 의존관계를 설정한다. Setter 기반의 DI는 <bean>의 하위 태그로 해당 클래스의 값을 설정하는 <property>와 하위 속성이 있다.

```
<property name="필드명" value="값" />
```

속성	설명
name	값의 의존 관계를 설정시킬 대상이 되는 필드명 지정
ref	객체 전달. <ref bean = "bean 명" /> 으로 표기.
value	값을 전달. <value>값</value> 으로 표기

- <property> 요소로 클래스 간의 의존관계 설정 – 설정하는 클래스는 <ref> 속성으로 표기한다.

```
<bean id="bbsDAO" class="com.dao.BbsDAO" />
<bean id="bbsService" class="com.service.BbsService" >
  <property name="dao"><ref bean="bbsDAO"></ref></property>
</bean>
<bean id="listController" class="com.controller.ListController" >
  <property name="service"><ref bean="bbsService"></ref></property>
</bean>
```

- <property> 요소의 value 속성

```
<bean id="dataSource" class="org.apache.~.dbcp.BasicDataSource" >
  <!--setDriverClassName(String) 메서드 호출 -->
  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
  <property name="url" value="jdbc:oracle:thin:@127.0.0.1:1521:XE" />
  <property name="username" value="hr" />
```

```
<property name="password" value="hr" />
</bean>
```

· 이너 빈(inner beans)

<property>나 <constructor-arg> 내부에 <bean>으로 정의한 빈을 이너 빈(inner bean) 또는 내부 빈이라고 부른다.

```
<bean id="outer" class="..." />
  <!-- 대상 빈에 대한 참조를 인라인으로 정의 -->
  <property id="target">
    <bean class="com.exam.Person" ><!-- 이너 빈 -->
      <property name="name" value="hong gil dong" />
      <property name="age" value="25" />
    </bean>
  </property>
</bean>
```

· 부모 빈 참조

동일한 빈 설정 정보가 중복될 때 부모 빈을 설정하고, 부모 빈 설정 정보를 참조하여 재사용할 수 있다. 부모 빈을 참조하는 빈에 지정된 프로퍼티의 값을 설정한다. 가장 일반적인 방법은 <ref>의 parent 속성으로 대상 빈을 지정한다.

```
<!-- 부모 컨텍스트 -->
<bean id="bbsService" class="com.service.BbsService" />
<!-- 자식 컨텍스트 -->
<bean id="bbsService" <!-- 부모 빈명과 동일 -->
  class="org.springframework.aop.framework.ProxyFactoryBean" >
  <property name="target"><ref parent="bbsService"></ref></property>
  <!-- 다른 설정과 의존성 추가 -->
</bean>
```

다음과 같이 dataSource의 설정 빈을 참조하여 다른 빈에 재사용할 수 있다.

```
<bean id="dataSource" class="org.apache.~.DriverManagerDataSource" >
  <property name="driverClassName" value="{jdbc.driverClass}" />
  <property name="url" value=" {jdbc.url}" />
  ...
</bean>
<bean id="transactionManager" class="org.~DataSourceTransactionManager" >
  <property name="dataSource" ref="dataSource" />
</bean>
<bean id="sqlSessionFactory" class="org.~SqlSessionFactoryBean" >
  <property name="dataSource" ref="dataSource" />
  ...
```

</bean>

· 컬렉션(Collection)

<property> 또는 <constructor-arg>의 하위 태그로 자바 컬렉션 타입인 <list>, <set>, <map>, <props> 태그로 프로퍼티와 인수를 설정할 수 있다.

태그	타입	설명
<list>	java.util.List, 배열	List 타입이나 배열에 값 목록을 전달할 때
<map>	java.util.Map	Map 타입에 <키, 값> 목록을 전달할 때
<set>	java.util.Set	Set 타입에 값 목록을 전달 할 때
<props>	java.util.Properties	Properties 타입에 <프로퍼티이름, 프로퍼티값> 목록을 전달할 때

컬렉션에는 <list>, <set>, <map>, <props>가 있으며, 컬렉션에 값을 설정하는 태그는 다음과 같다.

태그	설명	태그	설정
<ref>	<bean>으로 등록된 객체	<bean>	임의의 빈
<value>	기본값, 스트링 타입	<null>	널(null)

컬렉션을 설정하는 일반 형식은 다음과 같다.

```
<bean id="id명" class="클래스명..." >
  <property name="item명">
    <list> | <set> | <map> | <props>
      <ref bean="값1" > | <entry key="1" value-ref="값1" /> | <prop key="키명1">값1</prop>
      ...
      <ref bean="값n" > | <entry key="n" value-ref="값n" /> | <prop key="키명n">값n</prop>
    </list> | </set> | </map> | </props>
  </property>
</bean>
```

- <list> 태그는 List 타입이나 배열에 저장될 객체 목록을 <ref>, <value> 태그로 값을 설정한다.
- <set> 태그는 <list>와 같은 방법으로 값을 설정한다.
- <map> 태그는 Map 계열의 컬렉션 객체들을 <entry> 태그를 이용하여 value 값을 맵에 설정한다.
- <entry> 태그는 <키, 값>으로 표현하며, <key>/<value>, <key-ref>/<value-ref>, <key-type>/<value-type> 속성으로 설정한다.
- <props> 태그는 "java.util.Properties" 문자열 값을 설정하며, <props>의 <prop key="키명">에 키값을 설정한다.

(3) XML 네임스페이스를 이용한 프로퍼티 설정

XML 네임스페이스는 <property> 대신에 간단하게 설정하는 방법이다. XML 네임스페이스를 사용할 경우에는 xmlns:p="http://www.springframework.org/schema/p"가 <beans>에 선언되어야 한다.

- p 네임스페이스는 <property> 요소 대신 bean 요소의 속성으로 표기한다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:p="http://www.springframework.org/schema/p"
```

```

xsi:schemaLocation="http://www.springframework.org/schema/beans
...
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    p:driverClassName="oracle.jdbc.driver.OracleDriver"
    p:url="jdbc:oracle:thin:@127.0.0.1:1521:XE"
    p:username="hr"
    p:password="1234" />
</beans>

```

- c 네임스페이스는 <constructor-arg> 대신 인라인 속성으로 설정한다.

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
...
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
<bean id="data" class="com.test.Data" >
    <constructor-arg ref="doc">
    <constructor-arg value="data12@naver.com">
</bean>
<!-- 'c-namespace' declaration -->
<bean id="data" class="com.test.Data" c:bar-ref="doc" c:email="data12@naver.com" />
</beans>

```

## 7.2 의존 관계 자동 설정

의존 관계 자동 설정은 자동 설정 모드를 통하여 빈을 설정하는 방법이다. 프로퍼티나 생성자 인수를 지정하는 일을 현저하게 줄일 수 있기 때문에 개발할 때 매우 유용한 방법이다. XML 기반의 설정 메타 데이터를 사용할 때 <bean>요소의 **autowire** 속성으로 "모드"를 지정한다.

(1) 자동 설정 모드는 다음과 같이 5가지가 존재한다.

속성	설명
no	기본값이며 자동 설정하지 않는다.
byName	프로퍼티 이름에 의한 자동 설정
byType	프로퍼티 타입을 이용한 자동 설정
constructor	생성자 파라메타 타입을 이용한 자동 설정
autodetect	constructor와 byType을 이용한 자동 설정

### ① byName

빈의 프로퍼티명과 일치하는 빈의 "name"이나 "id"가 있으면 자동 설정한다. 전달되는 빈 객체가 setXXX() 메서드로 받아야 한다. 프로퍼티명으로 전달하기 때문에 소스 코드의 프로퍼티명이 변경되면 설정 파일도 변경해야 한다.

## ② byType

빈의 프로퍼티 타입과 일치하는 빈의 "name"이나 "id"가 있으면 자동 설정한다.

## ③ constructor

constructor는 byType과 동일하게 타입을 이용하여 의존관계를 자동 설정한다.

## ④ autodetect

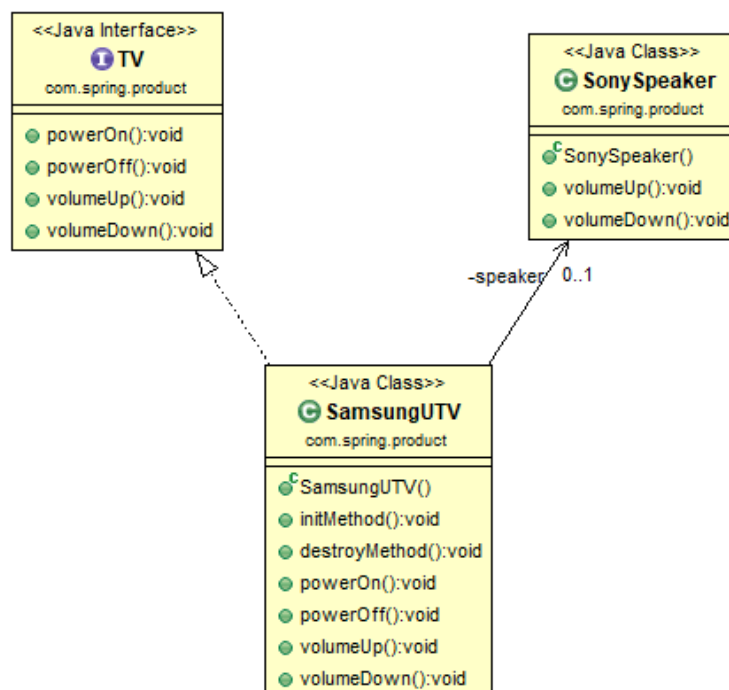
autodetect는 constructor 방식을 먼저 적용하고, constructor 방식을 적용할 수 없을 경우 byType 방식을 적용하여 의존관계를 자동 설정한다.

### (2) 자동 설정의 한계와 단점

- <property>와 <constructor-arg> 설정의 명시적 의존성을 항상 자동 설정을 재정의한다. 원시 타입, 문자열, 클래스 같은 간단한 프로퍼티들은 자동 설정할 수 없다.
- 자동 설정은 명시적인 설정보다 정확하지 않다.
- 설정 정보는 스프링 컨테이너에서 문서 생성 같은 도구에서는 사용할 수 없다.
- 컨테이너에서 다수의 빈 정의들은 자동 설정되는 setter 메서드나 생성자 인수로 지정된 타입과 일치하는 것을 찾는다. 단일 값을 갖는 의존성에 대해 모호할 경우 임의로 처리되지 않으며, 유일한 빈 정의를 찾지 못하면 예외가 발생한다.

## 7.3 의존성 관계

의존성(Dependency) 관계란 객체와 객체의 결합 관계이다. 즉 하나의 객체에서 다른 객체의 변수나 메서드를 이용해야 한다면 이용하려는 객체에 대한 객체 생성과 생성된 객체의 레퍼런스 정보가 필요하다.



위 그림에서 `SamsungUTV`는 `SonySpeaker`의 메서드를 이용해서 기능을 수행한다. 따라서 `SamsungUTV`는 `SonySpeaker`타입의 `speaker` 참조변수를 멤버변수로 가지고 있으며, `speaker` 참조변수는 `SonySpeaker` 객체를 참조하고 있어야 한다. 따라서 `SamsungUTV` 클래스는 어딘가에는 `SonySpeaker` 클래스에 대한 객체 생성 코드가 반드시 필요하다. 의존관계를 테스트하기 위해서 다음과 같이 `SonySpeaker` 클래스를 추가로 작성한다.

```

package com.spring.product;

public class SonySpeaker{
    public SonySpeaker() {
        System.out.println("==> SonySpeaker 객체 생성");
    }
    public void volumeUp() {
        System.out.println("SonySpeaker---소리 올린다.");
    }
    public void volumeDown() {
        System.out.println("SonySpeaker---소리 내린다.");
    }
}

```

그리고 SamsungUTV 클래스의 볼륨 조절 기능은 SonySpeaker를 이용하도록 수정한다.

```

package com.spring.product;

public class SamsungUTV implements TV {
    private SonySpeaker speaker;
    public SamsungUTV() {
        System.out.println("SamsungUTV 객체 생성");
    }
    public void initMethod() {
        System.out.println("객체 초기화 작업 처리.");
    }
    public void destroyMethod() {
        System.out.println("객체 삭제 전에 처리할 로직 처리...");
    }
    @Override
    public void powerOn() {
        System.out.println("SamsungUTV---전원을 켜다.");
    }
    @Override
    public void powerOff() {
        System.out.println("SamsungUTV---전원을 끈다.");
    }
    @Override
    public void volumeUp() {
        speaker = new SonySpeaker();
        speaker.volumeUp();
        //System.out.println("SamsungUTV---소리를 올린다.");
    }
}

```

```

@Override
public void volumeDown() {
    speaker = new SonySpeaker();
    speaker.volumeDown();
    //System.out.println("SamsungUTV---소리를 내린다.");
}
}

```

SamsungUTV 클래스 내의 volumeUp()과 volumeDown() 중 사용자는 두 개의 메서드를 호출 할 수도 있지만 하나의 메서드만 호출할 수 있기 때문에 이 예제에서는 두 메서드에 어떤 메서드를 SonySpeaker 객체 생성 코드를 모두 작성했다. 이제 TVUserEx 클래스를 다시 메서드를 호출하는 소스로 변경 한 후에 실행하면 다음과 같이 실행 결과를 확인할 수 있다.

TVUserEx.java

```

package com.spring.product;

import org.junit.Test;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

public class TVUserEx {
    @Test
    public void TvTestEx() {
        // 1. Spring 컨테이너를 구동한다.
        AbstractApplicationContext factory =
            new GenericXmlApplicationContext("applicationContext.xml");
        // 2. Spring 컨테이너로부터 필요한 객체를 요청(Lookup)한다.
        TV tv = (TV) factory.getBean("tv");
        tv.powerOn();
        tv.volumeUp();
        tv.volumeDown();
        tv.powerOff();

        // 2. Spring 컨테이너로부터 필요한 객체를 요청(Lookup)한다.
        // TV tv1 = (TV)factory.getBean("tv");
        // TV tv2 = (TV)factory.getBean("tv");
        // TV tv3 = (TV)factory.getBean("tv");

        // 3. Spring 컨테이너를 종료한다.
        factory.close();
    }
}

```



```
Markers Properties Servers Data Source Explorer Snippets Console Progress
-terminated- TVUserEx (Java Application) C:\SpringProject\workspace\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_15.0.2.v20210201-0955\jre\bin\java.exe (2021. 4. 15. 오후 2:48:52 - 오후 2:48:55)
INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource [applicationContext.xml]
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.GenericXmlApplicationContext@2f44
SamsungUTV 객체 생성
SamsungUTV---전원을 켜다.
==> SonySpeaker 객체 생성
SonySpeaker---소리 올린다.
==> SonySpeaker 객체 생성
SonySpeaker---소리 내린다.
SamsungUTV---전원을 끈다.
INFO : org.springframework.context.support.GenericXmlApplicationContext - Closing org.springframework.context.support.GenericXmlApplicationContext@2f49077
```

하지만 이 프로그램에는 두 가지 문제가 있다. 첫번째 SonySpeaker 객체가 쓸데없이 두개나 생성한 것이고, 두번째는 운영 과정에서 SonySpeaker의 성능이 떨어져서 SonySpeaker를 AppleSpeaker 같은 Speaker로 변경하고자 할 때 volumeUp(), volumeDown() 두 개의 메서드를 모두 수정해야 한다.

이러한 문제가 발생하는 이유는 의존 관계에 있는 Speaker 객체에 대한 생성 코드를 직접 SamsungUTV 소스에 명시했기 때문이다. 스프링은 이 문제를 의존성 주입(Dependency Injection)을 이용하여 해결한다.

## 7.4 생성자 인젝션 사용하기

스프링 컨테이너는 XML 설정 파일에 등록된 클래스를 찾아서 객체 생성할 때 기본적으로 매개변수가 없는 기본 생성자를 호출한다. 하지만 컨테이너가 기본 생성자 말고 매개변수를 가지는 다른 생성자를 호출하도록 설정할 수 있는데 이 기능을 이용하여 **생성자 인젝션(Constructor Injection)**을 처리한다. 생성자 인젝션을 테스트하기 위해서 SamsungUTV 클래스에 생성자를 추가한다.

```
package com.spring.product;

public class SamsungUTV implements TV {
    private SonySpeaker speaker;

    public SamsungUTV() {
        System.out.println("SamsungUTV 객체 생성");
    }

    public SamsungUTV(SonySpeaker speaker) {
        System.out.println("SamsungUTV SonySpeaker 객체 생성");
        this.speaker = speaker;
    }

    public void initMethod() {
        System.out.println("객체 초기화 작업 처리.");
    }

    public void destroyMethod() {
        System.out.println("객체 삭제 전에 처리할 로직 처리...");
    }

    @Override
    public void powerOn() {
        System.out.println("SamsungUTV---전원을 켜다.");
    }

    @Override
    public void powerOff() {
        System.out.println("SamsungUTV---전원을 끈다.");
    }
}
```

```

    }

    @Override
    public void volumeUp() {
        speaker.volumeUp();
    }

    @Override
    public void volumeDown() {
        speaker.volumeDown();
    }
}

```

SamsungUTV 클래스에 추가된 두 번째 생성자는 SonySpeaker 객체를 매개변수로 받아서 멤버변수로 선언된 speaker에 대입하도록 설정한다.

이를 위해 XML 설정 파일은 다음과 같다.

```

<?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.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <bean id="tv" class="com.spring.product.SamsungUTV">
        <constructor-arg ref="sony"></constructor-arg>
    </bean>
    <bean id="sony" class="com.spring.product.SonySpeaker" />
</beans>

```

생성자 인젝션을 위해서는 SamsungUTV 클래스 <bean> 등록 설정에서 시작 태그와 종료 태그 사이에 <constructor-arg> 엘리먼트를 추가하면 된다.

그리고 생성자 인자로 전달할 객체의 아이디를 <constructor-arg> 엘리먼트에 ref 속성으로 참조한다.

실행 결과는 다음과 같이 확인할 수 있으며, SamsungUTV 클래스의 객체가 생성될 때, 기본 생성자가 아닌 두번째 생성자가 사용됐다는 것과 스프링 설정파일에 SonySpeaker가 SamsungUTV 클래스 아래에 등록되었는데도 먼저 생성되고 있다는 것에 대해 확인하기 바란다.

```

INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading...
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.GenericXmlApplicationContext@2f4907!
==> SonySpeaker 객체 생성
SamsungTV SonySpeaker 객체 생성
SamsungUTV---전원을 켜다.
SonySpeaker---소리 울린다.
SonySpeaker---소리 내린다.
SamsungUTV---전원을 끈다.
INFO : org.springframework.context.support.GenericXmlApplicationContext - Closing org.springframework.context.support.GenericXmlApplicationContext@2f4907!

```

스프링 컨테이너는 기본적으로 <bean> 등록된 순서대로 객체를 생성하며, 모든 객체는 기본 생성자 호출을 원칙으로 한다. 그런데 생성자 인젝션으로 의존성 주입될 SonySpeaker가 먼저 객체 생성되었으며, SonySpeaker객체를 매개변수로 받아들이는 생성자 호출하여 객체를 생성하였다. 결국 SamsungUTV는 SonySpeaker 객체를 참조할 수 있으므로 문제없이 볼륨 조절을 처리할 수 있게 되었다.

(1) 다중 변수 매핑

생성자 인젝션에서 초기화해야 할 멤버변수가 여러 개이면, 여러 개의 값을 한꺼번에 전달해야 한다. 이 때는 다음처럼 생성자를 적절하게 추가하면 된다.

```
package com.spring.product;

public class SamsungUTV implements TV {
    private SonySpeaker speaker;
    private int price;

    public SamsungUTV() {
        System.out.println("SamsungUTV 객체 생성");
    }
    public SamsungUTV(SonySpeaker speaker) {
        System.out.println("SamsungTV SonySpeaker 객체 생성");
        this.speaker = speaker;
    }
    public SamsungUTV(SonySpeaker speaker, int price) {
        System.out.println("SamsungTV 객체 생성과 가격");
        this.speaker = speaker;
        this.price = price;
    }
    public void initMethod() {
        System.out.println("객체 초기화 작업 처리.");
    }
    public void destroyMethod() {
        System.out.println("객체 삭제 전에 처리할 로직 처리...");
    }
    @Override
    public void powerOn() {
        System.out.println("SamsungUTV---전원을 켜다. (가격 : " + price + ")");
    }
    @Override
    public void powerOff() {
        System.out.println("SamsungUTV---전원을 끈다.");
    }
    @Override
    public void volumeUp() {
        speaker.volumeUp();
    }
    @Override
    public void volumeDown() {
        speaker.volumeDown();
    }
}
```

```
}
}
```

그리고 스프링 설정 파일에 <constructor-arg> 엘리먼트를 매개변수의 개수만큼 추가해야 한다.

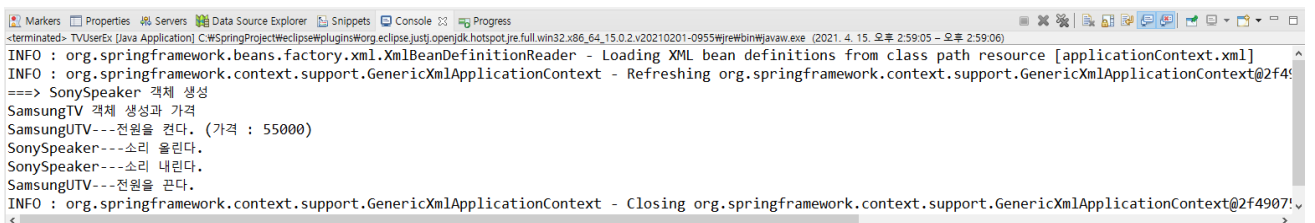
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <bean id="tv" class="com.spring.product.SamsungUTV">
        <constructor-arg ref="sony"></constructor-arg>
        <constructor-arg value="55000"></constructor-arg>
    </bean>

    <bean id="sony" class="com.spring.product.SonySpeaker" />
</beans>
```

<constructor-arg> 엘리먼트에는 ref와 value 속성을 사용하여 생성자 매개변수로 전달할 값을 지정할 수 있다. 이때 인자로 전달될 데이터가 <bean>으로 등록된 다른 객체일 때는 ref 속성을 이용하여 해당 객체의 아이디나 이름을 참조하지만, 고정된 문자열이나 정수 값은 기본형 데이터 일때는 value 속성을 사용한다. 따라서 SonySpeaker 객체를 생성자 인자로 전달할 때는 ref 속성을 이용하고, 55000이라는 정수 값을 전달할 때는 value 속성을 사용하였다.

이렇게 설정한 후에 TVUserEx 프로그램을 실행하면 다음과 같은 결과가 출력된다.



```
<terminated> TVUserEx [Java Application] C:\SpringProject\workspace\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_15.0.2.v20210201-0955\jre\bin\javaw.exe (2021. 4. 15. 오후 2:59:05 - 오후 2:59:06)
INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource [applicationContext.xml]
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.GenericXmlApplicationContext@2f4907!
==> SonySpeaker 객체 생성
SamsungTV 객체 생성과 가격
SamsungTV---전원을 켜다. (가격 : 55000)
SonySpeaker---소리 올린다.
SonySpeaker---소리 내린다.
SamsungTV---전원을 끈다.
INFO : org.springframework.context.support.GenericXmlApplicationContext - Closing org.springframework.context.support.GenericXmlApplicationContext@2f4907!
<
```

그런데 필드가 여러 개 정의되어있을 때 index 속성을 이용하여 어떤 값이 어느 필드에 매개변수로 매핑되는지 지정할 수 있다. index는 0부터 시작한다.

```
...
<constructor-arg index="0" ref="sony"></constructor-arg>
<constructor-arg index="1" value="55000"></constructor-arg>
...
```

## (2) 의존관계 변경

지금까지는 SamsungUTV 객체가 SonySpeaker를 이용하여 동작했지만 유지보수 과정에서 다른 스피커로 교체하는 상황도 발생할 것이다. 의존성 주입은 이런 상황을 매우 효과적으로 처리해 준다.

실습을 위해 모든 스피커 관련 클래스는 인터페이스를 구현한 구현체로 작성한다. 그래서 Speaker라는 이름으로 인터페이스를 추가한다.

```
package com.spring.product;

public interface Speaker {
    void volumeUp();
    void volumeDown();
}
```

그리고 Speaker 인터페이스를 구현한 또 다른 스피커인 AppleSpeaker를 추가로 구현한다.

```
package com.spring.product;

public class AppleSpeaker implements Speaker {
    public AppleSpeaker() {
        System.out.println("==> AppleSpeaker 객체 생성");
    }
    @Override
    public void volumeUp() {
        System.out.println("AppleSpeaker---소리 올린다.");
    }
    @Override
    public void volumeDown() {
        System.out.println("AppleSpeaker---소리 내린다.");
    }
}
```

그리고 SonySpeaker 클래스 역시 Speaker 인터페이스를 구현할 수 있도록 수정해야 한다.

```
public class SonySpeaker implements Speaker{
    ...
}
```

그리고 나서 SamsungUTV 클래스의 멤버변수와 매개변수 타입을 모두 SonySpeaker에서 Speaker로 수정하면 된다.

```
package com.spring.product;

public class SamsungUTV implements TV {
    private Speaker speaker;
    private int price;

    public SamsungUTV() {
        System.out.println("SamsungUTV 객체 생성");
    }
    public SamsungUTV(Speaker speaker) {
        System.out.println("SamsungTV Speaker 객체 생성");
        this.speaker = speaker;
    }
}
```

```

public SamsungUTV(Speaker speaker, int price) {
    System.out.println("SamsungTV 객체 생성과 가격");
    this.speaker = speaker;
    this.price = price;
}

public void initMethod() {
    System.out.println("객체 초기화 작업 처리.");
}

public void destroyMethod() {
    System.out.println("객체 삭제 전에 처리할 로직 처리...");
}

@Override
public void powerOn() {
    //System.out.println("SamsungUTV---전원을 켜다.");
    System.out.println("SamsungUTV---전원을 켜다. (가격 : " + price + ")");
}

@Override
public void powerOff() {
    System.out.println("SamsungUTV---전원을 끈다.");
}

@Override
public void volumeUp() {
    //speaker = new SonySpeaker();
    speaker.volumeUp();
    //System.out.println("SamsungUTV---소리를 올린다");
}

@Override
public void volumeDown() {
    //speaker = new SonySpeaker();
    speaker.volumeDown();
    //System.out.println("SamsungUTV---소리를 내린다.");
}
}

```

마지막으로 AppleSpeaker도 스프링 설정 파일에 <bean> 등록하고, <constructor-arg> 엘리먼트의 속성값을 apple로 지정하면 SamsungUTV가 AppleSpeaker를 이용하여 볼륨을 조절한다.

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"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans

```

```

http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <bean id="tv" class="com.spring.product.SamsungUTV">
        <constructor-arg ref="apple"></constructor-arg>
        <constructor-arg value="55000"></constructor-arg>
    </bean>

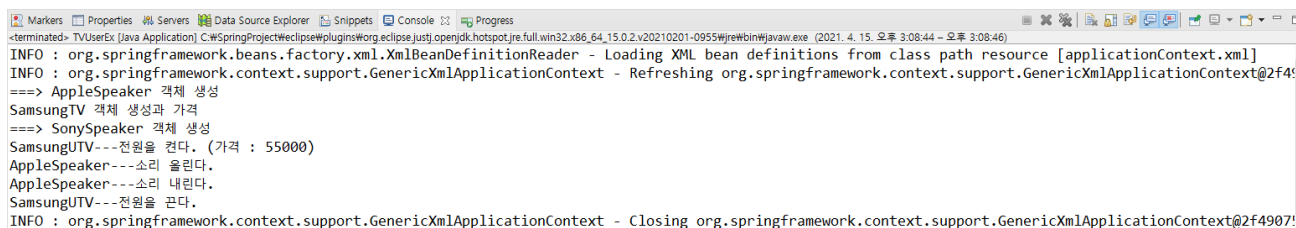
    <bean id="sony" class="com.spring.product.SonySpeaker" />
    <bean id="apple" class="com.spring.product.AppleSpeaker" />

</beans>

```

TVUser 프로그램을 다시 실행하면 다음과 같이 실행되는 Speaker가 AppleSpeaker로 변경된 것을 확인할 수 있다.

결국 스프링 설정 파일만 적절히 관리하면 동작하는 TV도 변경할 수 있고, TV가 사용하는 스피커도 변경할 수 있다. 여기서 핵심은 이 과정에서 어떤 자바 코드도 변경하지 않는다는 것이다.



```

INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource [applicationContext.xml]
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.GenericXmlApplicationContext@2f4907
==> AppleSpeaker 객체 생성
SamsungTV 객체 생성과 가격
==> SonySpeaker 객체 생성
SamsungUTV---전원을 켜다. (가격 : 55000)
AppleSpeaker---소리 올린다.
AppleSpeaker---소리 내린다.
SamsungUTV---전원을 끈다.
INFO : org.springframework.context.support.GenericXmlApplicationContext - Closing org.springframework.context.support.GenericXmlApplicationContext@2f4907

```

## 7.5 Setter 인젝션 이용하기

생성자 인젝션은 생성자를 이용하여 의존성을 처리한다. 하지만 Setter 인젝션은 이름에서 알 수 있듯이 Setter 메서드를 호출하여 의존성 주입을 처리하는 방법이다. 두 가지 방법 모두 멤버변수를 원하는 값으로 설정하는 것을 목적으로 하고 있고, 결과가 같으므로 둘 중 어떤 방법을 쓰든 상관없다. 다만 대부분 Setter 인젝션을 사용하며, Setter 메서드가 제공되지 않는 클래스에 대해서만 생성자 인젝션을 사용한다.

### (1) Setter 인젝션 기본

Setter 인젝션을 테스트하기 위해 SamsungUTV 클래스에 Setter 메서드를 추가한다.

```

package com.spring.product;

public class SamsungUTV implements TV {
    private Speaker speaker;
    private int price;

    public SamsungUTV() {
        System.out.println("SamsungUTV 객체 생성");
    }

    public SamsungUTV(Speaker speaker) {

```

```

        System.out.println("SamsungTV SonySpeaker 객체 생성");
        this.speaker = speaker;
    }
    public SamsungUTV(Speaker speaker, int price) {
        System.out.println("SamsungTV 객체 생성과 가격");
        this.speaker = speaker;
        this.price = price;
    }
    public void setSpeaker(Speaker speaker) {
        System.out.println("==> setSpeaker() 호출");
        this.speaker = speaker;
    }
    public void setPrice(int price) {
        System.out.println("==> setPrice() 호출");
        this.price = price;
    }
    public void initMethod() {
        System.out.println("객체 초기화 작업 처리.");
    }
    public void destroyMethod() {
        System.out.println("객체 삭제 전에 처리할 로직 처리...");
    }
    @Override
    public void powerOn() {
        //System.out.println("SamsungUTV---전원을 켜다.");
        System.out.println("SamsungUTV---전원을 켜다. (가격 : " + price + ")");
    }
    @Override
    public void powerOff() {
        System.out.println("SamsungUTV---전원을 끈다.");
    }
    @Override
    public void volumeUp() {
        //speaker = new SonySpeaker();
        speaker.volumeUp();
        //System.out.println("SamsungUTV---소리를 올린다");
    }
    @Override
    public void volumeDown() {
        //speaker = new SonySpeaker();
        speaker.volumeDown();
        //System.out.println("SamsungUTV---소리를 내린다.");
    }

```



```
}
}
```

Setter 메서드는 스프링 컨테이너가 자동으로 호출하며, 호출하는 시점은 <bean> 객체 생성 직후이다. 따라서 Setter 인젝션이 동작하려면 Setter 메서드뿐만 아니라 **기본 생성자도 반드시 필요하다.**

Setter 인젝션을 이용하려면 스프링 설정 파일에 <constructor-arg> 엘리먼트 대신 <property> 엘리먼트를 사용해야 한다.

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"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <bean id="tv" class="com.spring.product.SamsungUTV">
        <property name="speaker" ref="apple"></property>
        <property name="price" value="55000"></property>
    </bean>

    <bean id="sony" class="com.spring.product.SonySpeaker" />
    <bean id="apple" class="com.spring.product.AppleSpeaker" />
</beans>
```

Setter 인젝션을 이용하려면 <property> 엘리먼트를 사용해야 하며 name 속성값이 호출하고자 하는 메소드 이름이다. 즉 name 속성값이 "speaker"라고 설정되어 있으면 호출되는 메소드는 setSpeaker()이다.

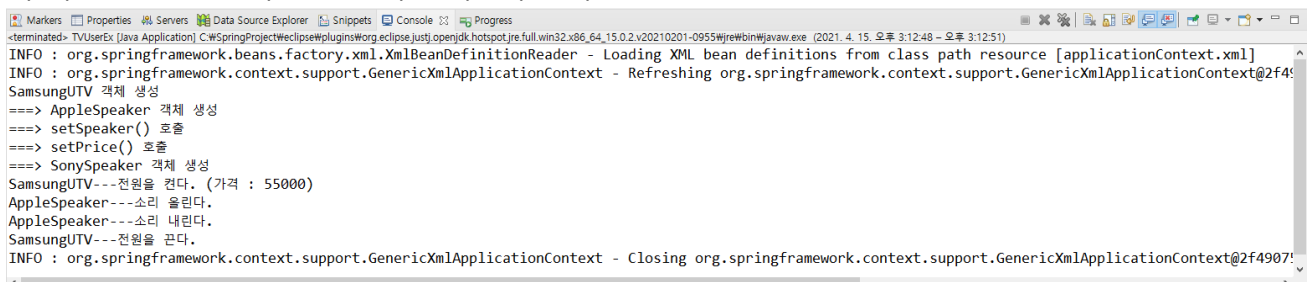
변수 이름에서 첫 글자를 대문자로 바꾸고 앞에 "set"을 붙인 것이 호출할 메서드 이름이다.

Setter 메서드와 <property> 엘리먼트의 name 속성의 관계에 대한 예제를 살펴보자

Setter 메서드 이름	name 속성값
setSpeaker()	name="speaker"
setAddressList()	name="addressList"
setBoardDAO()	name="boardDAO"

생성자 인젝션과 마찬가지로 Setter 메서드를 호출하면서 다른 <bean> 객체를 인자로 넘기려면 ref 속성을 사용하고, 기본형 데이터를 넘기려면 value 속성을 사용한다.

위 작성한 프로그램의 실행 결과는 다음과 같다.



```

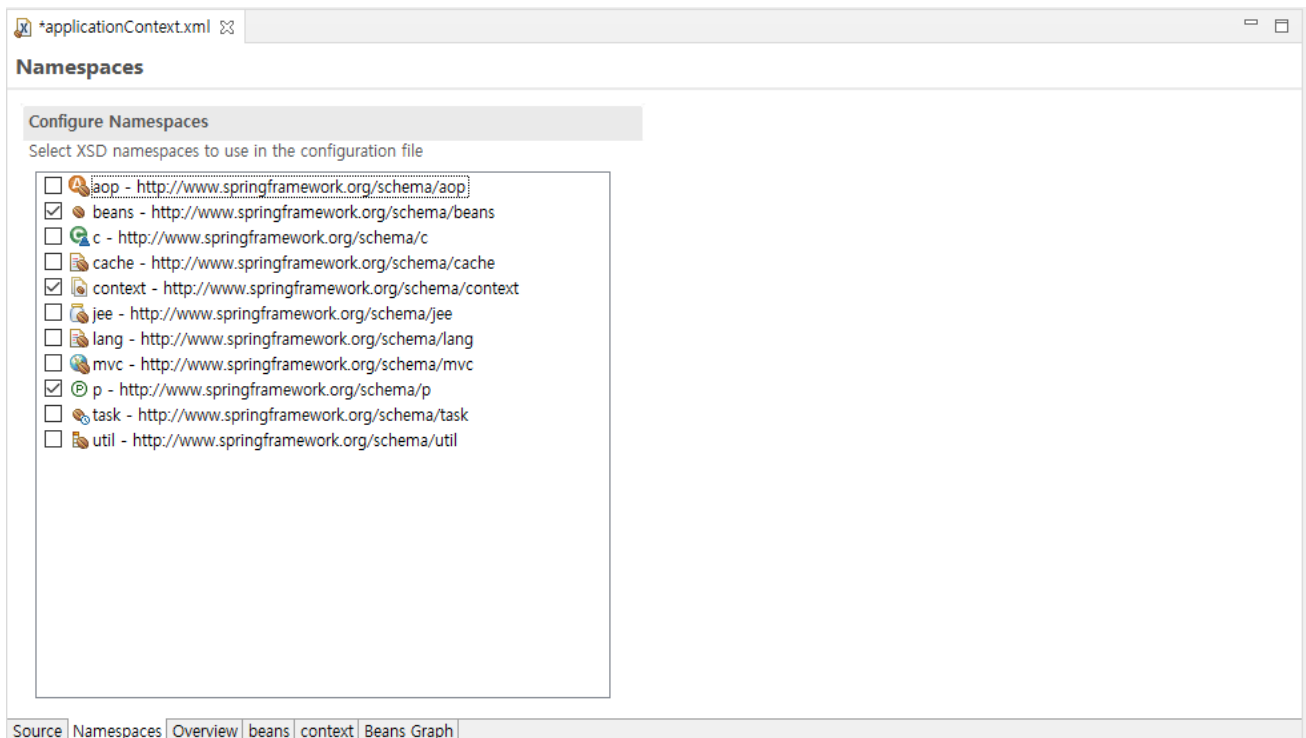
<terminated> TVUserEx [Java Application] C:\SpringProject\weclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_15.0.2.v20210201-0955\jre\bin\javaw.exe (2021. 4. 15. 오후 3:12:48 ~ 오후 3:12:51)
INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource [applicationContext.xml]
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.GenericXmlApplicationContext@2f4907...
SamsungUTV 객체 생성
====> AppleSpeaker 객체 생성
====> setSpeaker() 호출
====> setPrice() 호출
====> SonySpeaker 객체 생성
SamsungUTV---전원을 켜다. (가격 : 55000)
AppleSpeaker---소리 올린다.
AppleSpeaker---소리 내린다.
SamsungUTV---전원을 끈다.
INFO : org.springframework.context.support.GenericXmlApplicationContext - Closing org.springframework.context.support.GenericXmlApplicationContext@2f4907...

```

## (2) p 네임스페이스 사용하기

Setter 인젝션을 설정할 때, 'p 네임스페이스'를 이용하면 좀 더 효율적으로 의존성 주입을 처리할 수 있다.

**p 네임스페이스**는 네임스페이스에 대한 별도의 schemaLocation이 없다. 따라서 네임스페이스만 적절히 선언하고 사용할 수 있다.



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"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">
...
</beans>
```

p 네임스페이스를 선언했으면 다음과 같이 참조형 변수에 참조할 객체를 할당할 수 있다.

p:변수명-ref="참조할 객체의 이름이나 아이디"

기본형이나 문자형 변수에 직접 값을 설정할 때는 다음과 같이 사용한다.

p:변수명="설정할값"

다음은 p 네임스페이스를 이용하여 SamsungUTV와 SonySpeaker의 의존성 주입을 설정한 것이다.

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"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <bean id="tv" class="com.spring.product.SamsungUTV"
          p:speaker-ref="sony" p:price="55000" />
    <bean id="sony" class="com.spring.product.SonySpeaker" />
    <bean id="apple" class="com.spring.product.AppleSpeaker" />
</beans>

```

## 7.6 컬렉션(Collection) 객체 설정

### (1) List 타입 매핑

프로그램을 개발하다 보면 배열이나 List 같은 컬렉션 객체를 이용하여 데이터 집합을 사용해야 하는 경우가 있다. 이때 컬렉션 객체를 의존성 주입하면 되는데, 스프링에서는 이를 위해 컬렉션 매핑과 관련된 엘리먼트를 지원한다.

컬렉션 유형	엘리먼트
java.util.List, 배열	<list>
java.util.Set	<set>
java.util.Map	<map>
java.util.Properties	<props>

배열 객체나 java.util.List 타입의 컬렉션 객체는 <list> 태그를 사용하여 설정한다. 먼저 List 컬렉션을 멤버 변수로 가지는 CollectionBean 클래스를 다음과 같이 작성한다. **com.spring.injection** 패키지를 만든다.

src/main/java

```

package com.spring.injection;

import java.util.List;

public class CollectionBean {
    private List<String> addressList;

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

    public List<String> getAddressList() {
        return addressList;
    }
}

```

작성된 CollectionBean 클래스를 스프링 설정 파일에 다음과 같이 <bean> 등록한다.

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"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <!--
    <bean id="tv" class="com.spring.product.SamsungUTV" p:speaker-ref="sony" p:price="55000" />
    <bean id="sony" class="com.spring.product.SonySpeaker" />
    <bean id="apple" class="com.spring.product.AppleSpeaker" />
    -->
    <bean id="collectionBean" class="com.spring.injection.CollectionBean">
        <property name="addressList">
            <list>
                <value>서울시 강남구 역삼동</value>
                <value>서울시 성동구 무학로2길</value>
            </list>
        </property>
    </bean>
</beans>
```

위의 설정은 두 개의 문자열 주소가 저장된 List 객체를 CollectionBean 객체의 setAddressList() 메서드를 호출할 때 인자로 전달하여 addressList 멤버변수를 값 설정이다.

이제 간단한 클라이언트 프로그램을 작성하여 List 컬렉션이 정상적일 의존성 주입되었는지 확인해본다.

src/test/java에 생성한다.

```
package com.spring.injection;

import java.util.List;
import org.junit.Test;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

public class CollectionBeanClient {
    @Test
    public void TestEx() {
```

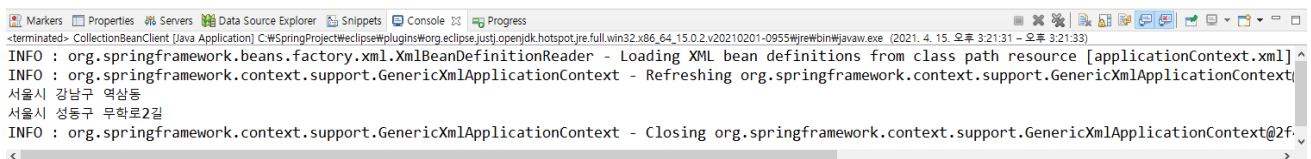
```

    AbstractApplicationContext factory =
        new GenericXmlApplicationContext("applicationContext.xml");

    CollectionBean bean = (CollectionBean) factory.getBean("collectionBean");
    List<String> addressList = bean.getAddressList();
    for (String address : addressList) {
        System.out.println(address.toString());
    }
    factory.close();
}
}

```

작성된 모든 파일을 저장하고 실행하면 다음과 같은 결과가 출력된다.



```

INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource [applicationContext.xml]
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.GenericXmlApplicationContext
서울시 강남구 역삼동
서울시 성동구 무학로2길
INFO : org.springframework.context.support.GenericXmlApplicationContext - Closing org.springframework.context.support.GenericXmlApplicationContext@2f

```

## (2) Set 타입 매핑

중복 값을 허용하지 않는 집합 객체를 사용할 때는 java.util.Set이라는 컬렉션을 사용한다. 컬렉션 객체는 <set> 태그를 사용하여 설정할 수 있다. CollectionBean.java 수정

```

package com.spring.injection;

import java.util.Set;

public class CollectionBean {
    private Set<String> addressList;

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

    public Set<String> getAddressList() {
        return addressList;
    }
}

```

CollectionBeanClient 클래스에서 다음과 같이 수정한다.

```

public class CollectionBeanClient {
    @Test
    public void TvTestEx() {
        ...
        CollectionBean bean = (CollectionBean) factory.getBean("collectionBean");
        //List<String> addressList = bean.getAddressList();
        Set<String> addressList = bean.getAddressList();
        ...
    }
}

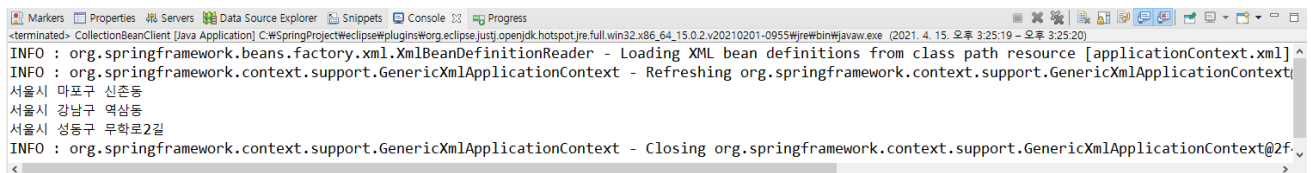
```

```
}
}
```

스프링 설정 파일에 <list> 엘리먼트를 <set> 엘리먼트로 수정하자.

applicationContext.xml

```
...
<bean id="collectionBean" class="com.spring.injection.CollectionBean">
    <property name="addressList">
        <set value-type="java.lang.String">
            <value>서울시 마포구 신촌동</value>
            <value>서울시 강남구 역삼동</value>
            <value>서울시 강남구 역삼동</value>
            <value>서울시 성동구 무학로2길</value>
        </set>
    </property>
</bean>
```



```
<terminated> CollectionBeanClient [Java Application] C:\SpringProject\weclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_15.0.2.v20210201-0955\jre\bin\javaw.exe (2021.4.15. 오후 3:25:19 - 오후 3:25:20)
INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource [applicationContext.xml]
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.GenericXmlApplicationContext
서울시 마포구 신촌동
서울시 강남구 역삼동
서울시 강남구 역삼동
서울시 성동구 무학로2길
INFO : org.springframework.context.support.GenericXmlApplicationContext - Closing org.springframework.context.support.GenericXmlApplicationContext@2f...
```

위의 예는 setAddressList() 메소드를 호출할 때, 문자열 타입의 데이터 여러 개의 저장한 Set 컬렉션을 인자로 전달하겠다는 설정이다. 그런데 위 설정을 보면 “서울시 강남구 역삼동”이라는 주소가 두 번 등록된 것을 확인할 수 있다. 그러나 Set 컬렉션은 같은 데이터를 중복해서 저장하지 않으므로 실제 실행해보면 “서울시 강남구 역삼동”이라는 주소가 한번 출력된 것을 확인할 수 있을 것이다.

### (3) Map 타입 매핑

특정 key로 데이터를 등록하고 사용할 때는 java.util.Map 컬렉션을 사용하며, <map> 태그를 사용하여 설정할 수 있다.

```
package com.spring.injection;

import java.util.Map;

public class CollectionBean {
    private Map<String, String> addressList;

    public void setAddressList(Map<String, String> addressList) {
        this.addressList = addressList;
    }

    public Map<String, String> getAddressList() {
        return addressList;
    }
}
```

CollectionBeanClient 클래스에서 다음과 같이 수정한다.

```
package com.spring.injection;

import java.util.Map;
import org.junit.Test;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

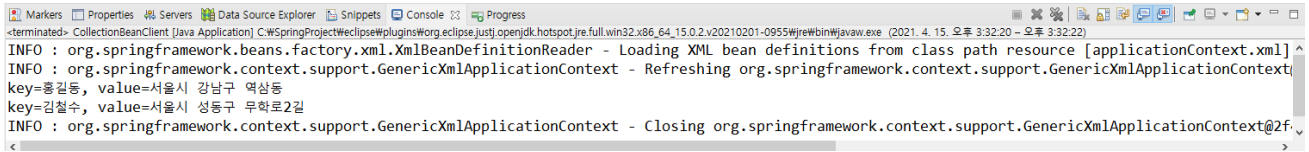
public class CollectionBeanClient {
    @Test
    public void TestEx() {
        AbstractApplicationContext factory =
            new GenericXmlApplicationContext("applicationContext.xml");
        CollectionBean bean = (CollectionBean) factory.getBean("collectionBean");
        Map<String, String> addressList = bean.getAddressList();
        // 모든 항목을 방문한다.
        for (Map.Entry<String, String> addList : addressList.entrySet()) {
            String key = addList.getKey();
            String value = addList.getValue();
            System.out.println("key=" + key + ", value=" + value);
        }
        factory.close();
    }
}
```

스프링 설정 파일에 <set> 엘리먼트를 <map> 엘리먼트로 수정하자.

applicationContext.xml

```
...
<bean id="collectionBean" class="com.spring.injection.CollectionBean">
    <property name="addressList">
        <map>
            <entry>
                <key><value>홍길동</value></key>
                <value>서울시 강남구 역삼동</value>
            </entry>
            <entry>
                <key><value>김철수</value></key>
                <value>서울시 성동구 무학로2길</value>
            </entry>
        </map>
    </property>
</bean>
...
```

위의 예는 setAddressList() 메소드를 호출할 때, Map 타입의 객체를 인자로 전달하는 설정이다. 이때 <entry> 엘리먼트에서 사용된 <key> 엘리먼트는 Map 객체의 key 값을 설정할 때 사용하며, <value> 엘리먼트는 Map 객체의 value를 설정할 때 사용한다.



#### (4) Properties 타입 매핑

key=value 형태의 데이터를 저장하고 사용할 때는 java.util.Properties라는 컬렉션을 사용하며, <props> 엘리먼트를 사용하여 설정한다.

```
package com.spring.injection;

import java.util.Properties;

public class CollectionBean {
    private Properties addressList;

    public void setAddressList(Properties addressList) {
        this.addressList = addressList;
    }

    public Properties getAddressList() {
        return addressList;
    }
}
```

CollectionBeanClient 클래스에서 다음과 같이 수정한다.

```
package com.spring.injection;

import java.util.Properties;
import org.junit.Test;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

public class CollectionBeanClient {
    @Test
    public void TestEx() {
        AbstractApplicationContext factory =
            new GenericXmlApplicationContext("applicationContext.xml");
        CollectionBean bean = (CollectionBean) factory.getBean("collectionBean");
        Properties addressList = bean.getAddressList();
        System.out.println(addressList.getProperty("홍길동"));
        System.out.println(addressList.getProperty("김철수"));
    }
}
```



```

        factory.close();
    }
}

```

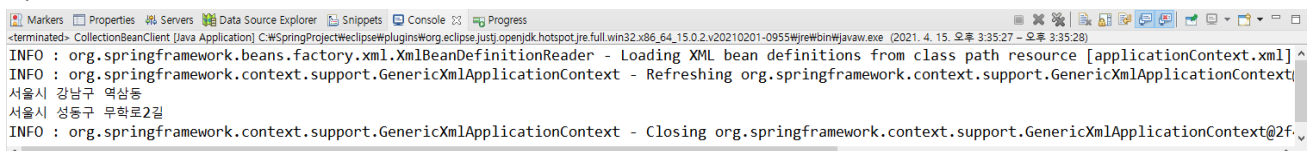
스프링 설정 파일에 <map> 엘리먼트를 <props> 엘리먼트로 수정하자.  
applicationContext.xml

```

...
<bean id="collectionBean" class="com.spring.injection.CollectionBean">
    <property name="addressList">
        <props>
            <prop key="홍길동">서울시 강남구 역삼동</prop>
            <prop key="김철수">서울시 성동구 무학로2길</prop>
        </props>
    </property>
</bean>
...

```

위의 예는 setAddressList() 메소드를 호출할 때, java.util.Properties 타입의 객체를 인자로 전달하는 설정이다.



## 7.7 클래스 자동 검색과 컴포넌트

클래스패스(classpath)에 위치한 클래스들을 검색하여 어노테이션이 붙은 클래스를 빈으로 자동 설정하는 기능이 제공된다. 스프링 2.0의 @Repository, 스프링 2.5 @Component, @Service, @Controller 어노테이션으로 소스코드의 클래스 선언부에 명시한다.

(1) @Component와 stereotype(스테레오타입) 어노테이션

@Repository, @Component, @Service, @Controller 어노테이션은 스프링이 관리하는 모든 컴포넌트에 대한 제너릭 스테레오타입이다. 이 어노테이션들은 애플리케이션의 프리젠테이션 계층, 서비스 계층, 퍼시스턴스 계층에서 사용된다.

@Component : 일반적인 컴포넌트를 설정하는 기본 스테레오 타입.

@Repository : 퍼시스턴스 계층의 DAO 컴포넌트

@Service : 서비스 계층의 서비스 컴포넌트

@Controller : 프레젠테이션 계층의 컨트롤러 컴포넌트

(2) 클래스 자동 검색과 빈 설정

스프링은 스테레오타입의 클래스들을 검색하고 대응되는 ApplicationContext의 빈을 자동으로 설정한다. 클래스들을 자동 검색하고 대응하는 빈을 설정하려면 XML에 <context:component-scan>요소를 포함시켜야 한다. base-package 요소는 클래스에 대한 공통 부모 패키지이다.

```
<context:component-scan base-package="패키지명" />
```

다음은 명시한 기본 패키지명에서 클래스 자동 검색과 빈을 설정하는 예이다.

```
<context:component-scan base-package="com.**.controller" />
<context:component-scan base-package="com.**.service" />
<context:component-scan base-package="com.**.dao" />
```

◆ ant 형식의 경로 패턴에 사용되는 3가지 대체 문자

- ① ? : ? 위치의 1개 문자 대체
- ② \* : \* 위치의 모든 문자 대체
- ③ \*\* : \*\* 위치의 모든 패키지(또는 디렉토리) 대체

(3) <context:compont-scan>의 하위 요소

@Component, @Repository, @Service, @Controller와 @Component가 붙은 애너테이션의 클래스들만 후보 컴포넌트로 탐색된다. <component-scan>의 하위요소로 <context:include-filter>나 <context:exclude-filter>를 추가한다.

- include-filter : 자동 스캔 대상에 포함시킬 클래스
- exclude-filter : 자동 스캔 대상에서 제외시킬 클래스

각 필터 요소들은 type과 expression 속성을 필요로 한다.

include-filter/exclude-filter에서 type속성의 필터 타입

```
<context:include-filter type="" expression="" />
<context:exclude-filter type="" expression="" />
```

필터 타입	설 명
annotation	대상 컴포넌트에서 타입 레벨로 표현되는 애너테이션
assignable	대상 컴포넌트가 extend/implement로 할당 가능한 클래스나 인터페이스
aspectj	대상 컴포넌트들과 일치되는 AspectJ 타입 표현식
regex	대상 컴포넌트 클래스명과 일치되는 정규 표현식
custom	org.springframework.core.type.TypeFilter 인터페이스를 구현한 커스텀 구현체

•○ "com" 하위 패키지에서 자동 검색될 때 "org.springframework.stereotype.Controller"의 애너테이션 타입을 포함시키는 <context:component-scan>의 설정은 다음과 같다.

```
<context:component-scan base-package="com"
    <context:include-filter type="annotation"
        Expression="org.springframework.stereotype.Controller" />
</context:component-scan>
```

## 8. 스프링 어노테이션 기반의 설정

어노테이션 기반의 설정은 XML로 컴포넌트를 설정하지 않고 소스 코드의 클래스나 메소드에 어노테이션으로 선언하는 방법이다. 스프링의 2.0의 @Required, 스프링 2.5에서 @Autowired, @PostConstruct, @PreDestroy, 스프링 3.0에서 @Inject, @Named 애너테이션을 사용한다. 애너테이션 주입은 XML 주입 이전에 실행되기 때문에 두 가지 방법을 사용할 경우 프로퍼티들은 XML 설정으로 치환된다.

애너테이션	설 명
@Required	프로퍼티 설정 메소드에 @Required 애너테이션 기술
@Autowired	의존 관계 자동 설정. 설정자, 필드, 메서드에 적용 가능. 빈 객체의 타입으로 자동 주입
@Qualifier	타입이 동일한 빈 객체의 특정 빈 설정
@Inject	@Autowired 애너테이션 대신 사용. required 속성 없음
@Named	@Component 애너테이션 대신 사용

애너테이션 기반의 설정 방법을 이용할 경우 스프링 설정 파일의 <beans> 내부에 <context:annotation-config>로 빈 객체를 설정해야 인식된다.

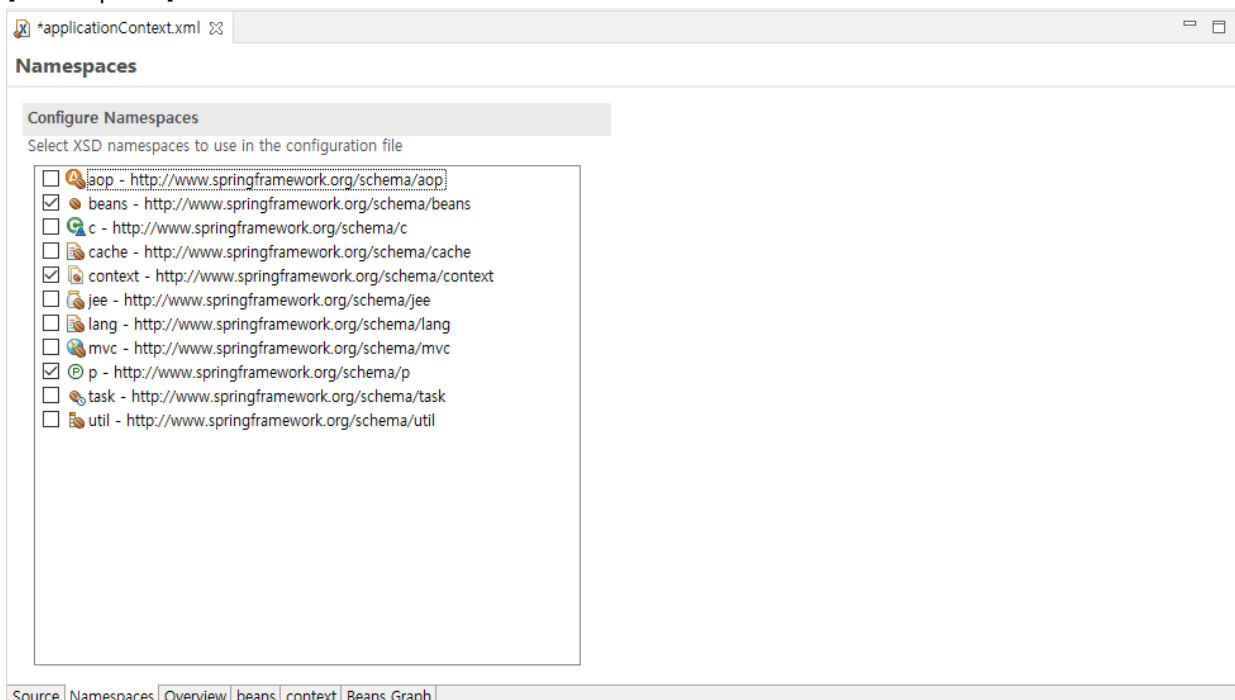
```
<context:annotation-config />
```

### 8.1 어노테이션 설정 기초

대부분 프레임워크가 그렇듯이 스프링 프레임워크 역시 XML 설정이 매우 중요하다. 그만큼 XML 파일의 과도한 설정에 대한 부담도 크며, 이로 인해 프레임워크 사용을 꺼리기도 한다. 따라서 대부분 프레임워크는 어노테이션을 이용한 설정을 지원하고 있다.

#### (1) Context 네임스페이스 추가

어노테이션 설정을 추가하려면 다음과 같이 스프링 설정 파일의 루트 엘리먼트인 <beans>에 Context 관련 네임스페이스와 스키마 문서의 위치를 등록해야 한다. 이는 p 네임스페이스를 추가했을 때처럼 [Namespaces] 탭을 선택하고 'context' 항목만 체크하면 간단하게 추가할 수 있다.



## (2) 컴포넌트 스캔(component-scan) 설정

스프링 설정 파일에 애플리케이션에서 사용할 객체들을 <bean> 등록하지 않고 자동으로 생성하려면 <context:component-scan />이라는 엘리먼트를 정의해야 한다. 이 설정을 추가하면 스프링 컨테이너는 클래스 패스에 있는 클래스들을 스캔하여 @Component가 설정된 클래스들을 자동으로 객체 생성한다.

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
    <context:component-scan base-package="com.spring" />
</beans>
```

<context:component-scan /> 설정을 제외한 나머지 <bean> 설정은 모두 삭제하거나 주석으로 처리하면 된다.

여기서 중요한 것은 <context:component-scan /> 엘리먼트의 base-package 속성인데, 만약 속성값을 "com.spring" 형태로 지정하면 com.spring 패키지로 시작하는 모든 하위 패키지를 스캔 대상에 포함한다. 따라서 다음과 같은 모든 패키지의 클래스들이 스캔 대상이 된다.

```
com.spring
com.spring.exam01
com.spring.injection
com.spring.product
com.spring.product.board
com.spring.product.board.impl
```

## (3) @Component

<context:component-scan />을 설정했으면 이제 스프링 설정 파일에 클래스들을 일일이 <bean> 엘리먼트로 등록할 필요가 없다. @Component만 클래스 선언부 위에 설정하면 끝난다. 예를 들어 LgTV 클래스에 대한 <bean> 등록을 XML 설정과 어노테이션 설정으로 처리하면 다음과 같다.

applicationContext.xml 설정

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
```

```

http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <context:component-scan base-package="com.spring" />

    <bean class="com.spring.product.LgUTV"></bean>

</beans>

```

#### Annotation 설정

```

package com.spring.product;

import org.springframework.stereotype.Component;

@Component
public class LgUTV implements TV {
    public LgUTV() {
        System.out.println("LgUTV 객체 생성");
    }
    ...
}

```

여기에서 두 설정 모두 해당 클래스에 기본 생성자가 있어야만 컨테이너가 객체를 생성할 수 있다. 이렇게 설정했다면 클래스의 객체가 메모리에 생성되는 것은 문제없다. 그러나 클라이언트 프로그램에서 LgTV 객체를 요청할 수 없다. 클라이언트가 스프링 컨테이너가 생성한 객체를 요청하려면, 요청할 때 사용할 아이디나 이름이 반드시 설정되어 있어야 한다.

```

package com.spring.product;

import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;
import org.junit.Test;

public class TVUserEx {
    @Test
    public void TvTestEx() {
        // 1. Spring 컨테이너를 구동한다.
        AbstractApplicationContext factory = new
        GenericXmlApplicationContext("applicationContext.xml");
        // 2. Spring 컨테이너로부터 필요한 객체를 요청(Lookup)한다.
        TV tv = (TV) factory.getBean("tv");
        tv.powerOn();
        tv.volumeUp();
        tv.volumeDown();
        tv.powerOff();
    }
}

```

```

        // 3. Spring 컨테이너를 종료한다.
        factory.close();
    }
}

```

따라서 클라이언트의 요청을 위해서라도 다음과 같이 아이디 설정(id="tv")이 필요하다.

applicationContext.xml 설정

```

<?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"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <context:component-scan base-package="com.spring" />

    <bean id="tv" class="com.spring.product.LgUTV"></bean>
</beans>

```

클래스 선언 부분에 @Component를 설정해줌으로써 스프링 컨테이너는 해당 클래스를 bean으로 생성하고 관리한다. 수정한 파일들을 모두 저장하고 TVUserEx 프로그램을 실행하면 다음과 같은 실행 결과를 확인할 수 있다.

```

<terminated> TVUserEx (Java Application) C:\SpringProject\weclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_15.0.2.v20210201-0955\jre\bin\javaw.exe (2021. 4. 15. 오후 3:45:53 - 오후 3:45:56)
INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource [applicationContext.xml]
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.GenericXmlApplicationContext
INFO : org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor - JSR-330 'javax.inject.Inject' annotation found and support
LgUTV 객체 생성
LgUTV 객체 생성
LgUTV---전원 켜다.
LgUTV---소리를 올린다
LgUTV---소리를 내린다.
LgUTV---전원 끈다.
INFO : org.springframework.context.support.GenericXmlApplicationContext - Closing org.springframework.context.support.GenericXmlApplicationContext@2f

```

applicationContext.xml 설정에서 id를 생략할 경우

```

...
    <bean class="com.spring.product.LgUTV"></bean>
...

```

Annotation에 설정해 준다.

```

package com.spring.product;
import org.springframework.stereotype.Component;

@Component("tv")
public class LgUTV implements TV {

```

```

public LgUTV() {
    System.out.println("LgUTV 객체 생성됨");
}
...
}

```

### id나 name 속성 미지정 시 이름 규칙

스프링 컨테이너가 클래스 객체를 생성할 때 id나 name 속성을 지정하지 않는다면, 컨테이너가 자동으로 이름을 설정해준다. 이때 이름 규칙은 클래스 이름의 첫글자를 소문자로 변경하기만 하면 된다. 따라서 id나 name 속성이 설정되지 않은 경우, LgTV 객체를 요청하려면 lgTV라는 이름을 사용하면 된다.

## 8.2 의존성 주입 설정

### (1) 의존성 주입 어노테이션

스프링에서 의존성 주입을 지원하는 어노테이션으로는 @Autowired, @Inject, @Qualifier, @Resource가 있다.

어노테이션	설명
@Autowired	주로 변수 위에 설정하여 해당 타입의 객체를 찾아서 자동으로 할당한다. org.springframework.beans.factory.annotation.Autowired;
@Qualifier	특정 객체의 이름을 이용하여 의존성 주입할 때 사용한다. org.springframework.beans.factory.annotation.Qualifier;

### (2) @Autowired

@Autowired는 생성자나 메서드, 멤버변수 위에 모두 사용할 수 있다. 어디에 사용하든 결과가 같아서 상관없지만, 대부분 멤버변수 위에 선언하여 사용한다. 스프링 컨테이너는 멤버 변수 위에 붙은 @Autowired를 확인하는 순간 해당 변수의 타입을 체크한다. 그리고 그 타입의 객체가 메모리에 존재하는지를 확인한 후에, 그 객체를 변수에 주입한다. 그런데 만약 @Autowired 가 붙은 객체가 메모리에 없다면 컨테이너가 NoSuchBeanDefinitionException을 발생시킨다. NoSuchBeanDefinitionException 메시지는 @Autowired 대상 객체가 메모리에 존재하지 않는다는 의미이다.

@Autowired 기능을 테스트 하기 위해서 LgTV 클래스에 다음과 같이 멤버변수를 추가하고 의존관계를 설정한다.

```

package com.spring.product;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component("tv")
public class LgTV implements TV {
    @Autowired
    private Speaker speaker;

    public LgTV() {
        System.out.println("==> LgTV 객체 생성됨");
    }
}

```

```

    }
    public void powerOn() {
        System.out.println("LgTV---전원 켜다.");
    }
    public void powerOff() {
        System.out.println("LgTV---전원 끈다.");
    }
    public void volumeUp() {
        //System.out.println("LgTV---소리를 올린다.");
        speaker.volumeUp();
    }
    public void volumeDown() {
        //System.out.println("LgTV---소리를 내린다.");
        speaker.volumeDown();
    }
}

```

위처럼 설정하면 LgTV 클래스에는 의존성 주입에 사용했던 Setter 메서드나 생성자는 필요없다. 그리고 스프링 설정 파일 역시 <context:component-scan /> 외에는 아무런 설정도 하지 않는다. 그러나 SonySpeaker객체가 메모리에 없으면 에러가 발생하므로 반드시 SonySpeaker 객체가 메모리에 생성되어 있어야 한다. SonySpeaker객체를 생성하려면 다음과 같이 **두 가지 방법 중 하나**를 처리해야 한다.

applicationContext.xml 설정 수정

```

...
<bean id="tv" class="com.spring.product.LgUTV"> </bean>
<bean id="sony" class="com.spring.product.SonySpeaker"> </bean>
...

```

Annotation 설정

```

package com.spring.product;

import org.springframework.stereotype.Component;

@Component("sony")
public class SonySpeaker implements Speaker{
    public SonySpeaker() {
        System.out.println("==> SonySpeaker 객체 생성");
    }
    ...
}

```

어떤 방법을 사용해도 상관없다. 의존성 주입 대상이 되는 SonySpeaker 객체가 메모리에 생성만 되면 @Autowired에 의해서 컨테이너가 SonySpeaker객체를 speaker 멤버변수에 자동으로 할당하기 때문이다.



```

INFO : org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor - JSR-330 'javax.inject.Inject' annotation found and support
LgUTV 객체 생성
==> SonySpeaker 객체 생성
LgUTV 객체 생성
LgUTV---전원 켜다.
SonySpeaker---소리 올린다.
SonySpeaker---소리 내린다.
LgUTV---전원 끈다.
INFO : org.springframework.context.support.GenericXmlApplicationContext - Closing org.springframework.context.support.GenericXmlApplicationContext@2f

```

### (3) @Qualifier

의존성 주입 대상이 되는 Speaker 타입의 객체가 두 개 이상일 때 에러가 발생한다. 만약 SonySpeaker 와 AppleSpeaker 객체가 모두 메모리에 생성되어 있는 상황이라면 컨테이너는 어떤 객체를 할당할지 스스로 판단할 수 없어서 에러가 발생한다. 이런 상황을 테스트하기 위해서 AppleSpeaker 클래스에도 @Component를 선언한다.

```

package com.spring.product;

import org.springframework.stereotype.Component;

@Component("apple")
public class AppleSpeaker implements Speaker {
    public AppleSpeaker() {
        System.out.println("==> AppleSpeaker 객체 생성");
    }
    ...
}

```

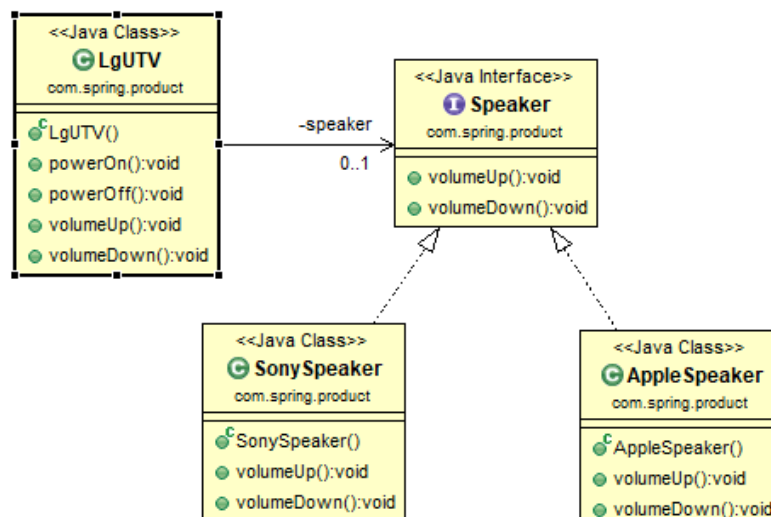
수정된 파일들을 모두 저장하고 TVUser 프로그램을 다시 실행하면 콘솔에 다음과 같은 예외 메시지가 출력된다.

```

생략...; nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException: No
qualifying bean of type [com.spring.product.Speaker] is defined: expected single matching bean but found
2: apple,sony

```

이는 아래 클래스 다이어그램에서 보듯이 @Autowired 대상이 되는 Speaker 타입의 객체가 SonySpeaker, AppleSpeaker로 두 개이고 둘 다 메모리에 생성되어 있으므로 어떤 객체를 의존성 주입할 지 모르기 때문에 발생한 예외이다.



이런 문제를 해결하기 위해서 스프링은 **@Qualifier** 어노테이션을 제공한다. @Autowired와 @Qualifier를 다음과 같이 변수 위에 사용하면 된다.

```
package com.spring.product;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component("tv")
public class LgUTV implements TV {
    @Autowired
    @Qualifier("apple")
    private Speaker speaker;

    public LgUTV() {
        System.out.println("LgUTV 객체 생성됨");
    }
    ...
}
```

@Qualifier 어노테이션을 이용하면 의존성 주입될 객체의 아이디나 이름을 지정할 수 있는데, 이때 Speaker 객체의 이름(sony, apple) 중 하나를 지정하면 간단하게 처리할 수 있다. 정상적으로 실행된다.

#### (4) 어노테이션과 XML 설정 병행하여 사용하기

스프링으로 의존성 주입을 처리할 때 XML 설정과 어노테이션 설정은 장단점이 서로 상충한다. 앞에서 살펴본 대로 XML 방식은 자바 소스를 수정하지 않고 XML 설정파일의 설정만 변경하면 실행되는 Speaker를 교체할 수 있어서 유지보수가 편하다. 하지만 XML 설정에 대한 부담 역시 존재한다. 그리고 자바 소스에 의존관계와 관련된 어떤 메타데이터도 없으므로 XML 설정을 해석해야만 무슨 객체가 의존성 주입되는지를 확인할 수 있다.

반면에 어노테이션 기반 설정은 XML 설정에 대한 부담도 없고 의존관계에 대한 정보가 자바 소스에 들어 있어서 사용하기는 편하다. 하지만 의존성 주입할 객체의 이름이 자바 소스에 명시되어야 하므로 자바 소스를 수정하지 않고 Speaker를 교체할 수 없다는 문제가 생긴다. 이런 문제를 서로의 장점을 조합하는 것으로 해결할 수 있는데 다음 실습으로 확인한다.

우선 LgUTV 클래스의 speaker 변수를 원래대로 @Autowired 어노테이션만 설정한다.

```
package com.spring.product;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component("tv")
public class LgUTV implements TV {
```

**@Autowired**

**private Speaker speaker;**

```
public LgUTV() {  
    System.out.println("LgUTV 객체 생성됨");  
}
```

...

}

당연히 위 설정만으로는 에러가 발생한다. Speaker 타입의 객체가 메모리에 두개 있어서이다. 하지만 기존의 SonySpeaker, AppleSpeaker 클래스에 설정했던 **@Component**를 제거하여 객체가 자동으로 생성되는 것을 차단한다.

```
package com.spring.product;  
public class SonySpeaker implements Speaker{  
    public SonySpeaker() {  
        System.out.println("==> SonySpeaker 객체 생성");  
    }  
    ...  
}
```

```
package com.spring.product;  
//@Component("apple")  
public class AppleSpeaker implements Speaker {  
    public AppleSpeaker() {  
        System.out.println("==> AppleSpeaker 객체 생성");  
    }  
    ...  
}
```

그리고 나서 둘 중 하나만 스프링 설정 파일에 <bean> 등록하여 처리하면 된다.

applicationContext.xml

```
...  
    <context:component-scan base-package="com.spring.product" />  
    <bean class="com.spring.product.SonySpeaker" />  
</beans>
```

위 설정대로라면 LgTV에 설정한 @Autowired에 의해서 SonySpeaker 객체가 의존성 주입된다. TVUserEx 실행한다.

이후에 AppleSpeaker로 교체할 때는 SonySpeaker를 AppleSpeaker로 수정하면 된다.

applicationContext.xml

```
...  
    <context:component-scan base-package="com.spring.product" />  
    <bean class="com.spring.product.AppleSpeaker" />  
</beans>
```

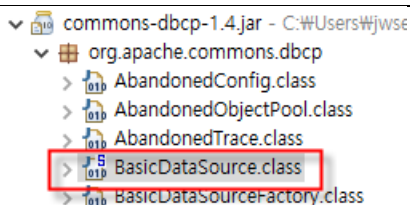
결국 클라이언트가 요청할 LgTV는 @Component 어노테이션으로 처리하고, 의존성 주입 역시 @Autowired로 처리한다. 다만 변경될 Speaker만 스프링 설정 파일에 <bean> 등록함으로써 자바 코드 수정없이 XML 수정만으로 Speaker를 교체할 수 있다.

TVUserEx 실행한다.

그래서 직접 개발한 클래스는 어노테이션을 사용할 수도 있고, XML 설정을 할 수도 있다. 하지만 라이브러리 형태로 제공되는 클래스는 반드시 XML 설정을 통해서만 사용할 수 있다. 따라서 아파치에서 제공하는 BasicDataSource 클래스를 사용하여 DB 연동을 처리한다면 'commons-dbc-1.4.jar' 파일에 있는 BasicDataSource 클래스에 관련된 어노테이션을 추가할 수는 없다.

pom.xml

```
<!-- DBCP -->
<dependency>
    <groupId>commons-dbc</groupId>
    <artifactId>commons-dbc</artifactId>
    <version>1.4</version>
</dependency>
```



BasicDataSource를 스프링 컨테이너가 생성하도록 하려면 다음과 같이 반드시 설정 파일에 <bean> 등록을 해야만 한다.

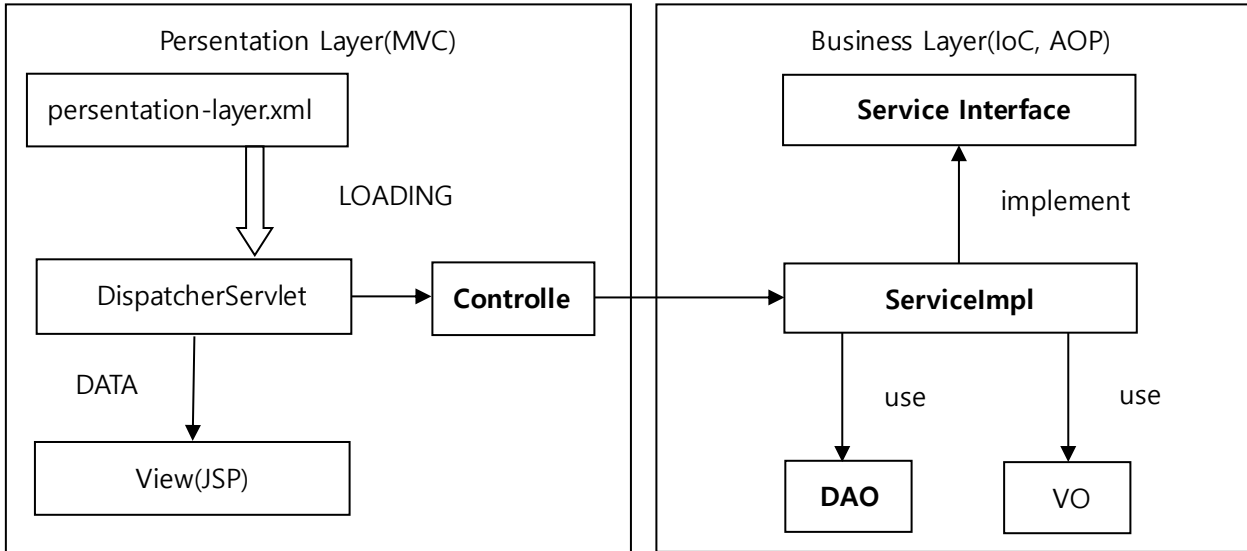
applicationContext.xml

```
<context:component-scan base-package="com.spring.product" />
<!-- DataSource 설정 -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
    <property name="url" value="jdbc:oracle:thin:@127.0.0.1:1521:XE" />
    <property name="username" value="hr" />
    <property name="password" value="1234" />
</bean>
```

### 8.3 추가 어노테이션

다음 그림은 시스템의 전체 구조를 두 개의 로직으로 표현한 것이다. 먼저 프레젠테이션 로직은 사용자와 커뮤니케이션을 담당하고, 비즈니스 로직은 사용자의 요청에 대한 비즈니스 로직 처리를 담당한다.

프레젠테이션과 비즈니스 레이어로 구성된 시스템 구조



이 구조에서 가장 핵심 요소는 Controller, ServiceImpl, DAO 클래스이다. Controller 클래스는 사용자의 요청을 제어하며, ServiceImpl 클래스는 실질적인 비즈니스 로직을 처리한다. DAO 클래스는 데이터베이스 연동을 담당한다.

앞에서 @Component를 이용하여 스프링 컨테이너가 해당 클래스 객체를 생성하도록 설정할 수 있었다. 그런데 시스템을 구성하는 이 모든 클래스에 @Component를 할당하면 어떤 클래스가 어떤 역할을 수행하는지 파악하기 어렵다. 스프링 프레임워크에서는 이런 클래스들을 분류하기 위해서 @Component를 상속하여 다음과 같은 세 개의 어노테이션을 추가로 제공한다.

어노테이션	클래스명	의미
@Service	XXXServiceImpl	비즈니스 로직을 처리하는 Service 클래스
@Repository	XXXDAO	데이터베이스 연동을 처리하는 DAO 클래스
@Controller	XXXController	사용자 요청을 제어하는 Controller 클래스

이처럼 어노테이션을 나눈 이유는 단순히 해당 클래스를 분류하기 위해서만은 아니다. @Controller는 해당 객체를 MVC(Model-View-Controller) 패턴에서 컨트롤러 객체로 인식하도록 해주며, @Repository는 DB 연동 과정에서 발생하는 예외를 변환해주는 특별한 기능이 추가되어 있다. 이와 관련된 내용은 해당 과지에서 다루고 클래스들을 분류하는 의미만 이해하고 넘어간다.