

Spring DI

Bok, Jong Soon
javaexpert@nate.com

<https://github.com/swacademy/Spring5>

IoC(Inversion of Control)

- 객체의 생성, 생명주기의 관리까지 모든 객체에 대한 제어권이 바뀌었다는 것을 의미
- Component 의존관계 결정(Component Dependency Resolution), 설정(Configuration) 및 Lifecycle를 해결하기 위한 Design Pattern
- 의존(Dependency)이란 변경에 의해 영향을 받는 관계라는 의미.
- 한 개의 Class의 내부 Code가 변경되었을 때 이와 관련된 다른 Class도 함께 변경해야 한다면 이를 변경에 따른 영향이 전파되는 관계로서 **의존**한다고 표현한다.
- 의존하는 대상이 있으면, 그 대상을 구하는 방법 필요.
- 가장 쉬운 방법은 의존 대상 객체를 직접 생성하는 것이다.
- 그래서 의존받는 Class를 생성하면 그 Class가 의존하고 있는 Class도 동시에 생성이 된다.
- 이렇게 Class 내부에서 직접 의존 객체를 생성하는 것은 쉽지만, 유지 보수 관점에서 보면 문제점이 유발될 수 있다.

IoC(Inversion of Control) Container

- Spring Framework도 객체에 대한 생성 및 생명주기를 관리할 수 있는 기능을 제공하고 있음.
- IoC Container 기능 제공.
- IoC Container는 객체의 생성을 책임지고, 의존성을 관리.
- POJO의 생성, 초기화, Service, 소멸에 대한 권한을 가진다.
- 개발자들이 직접 POJO를 생성할 수 있지만 Container에게 맡긴다.

IoC(Inversion of Control) 분류

■ DI : Dependency Injection

- Spring, PiconContainer
- Setter Injection, Constructor Injection, Method Injection
- 각 Class간의 의존관계를 빈 설정(Bean Definition) 정보를 바탕으로 Container가 자동으로 연결해주는 것

■ DL : Dependency Lookup

- EJB, Spring
- 의존성 검색 : 저장소에 저장되어 있는 Bean에 접근하기 위해 Container가 제공하는 API를 이용하여 Bean을 Lookup 하는 것

■ DL 사용시 Container 종속성이 증가하여, 주로 DI를 사용함.



Task 1. Non-DI Spring Project





Task 2. DI Demo in Spring



DI(Dependency Injection) 개념

- 각 Class간의 의존관계를 빈 설정(Bean Definition) 정보를 바탕으로 Container가 자동으로 연결해 주는 것을 말함.
- 개발자들은 단지 Bean 설정 File에서 의존관계가 필요하다는 정보를 추가하면 된다.
- 객체 Reference를 Container로부터 주입 받아서, 실행시에 동적으로 의존관계가 생성된다.
- Container가 흐름의 주체가 되어 Application Code에 의존관계를 주입해주는 것이다.
- 장점
 - Code 단순.
 - Component 간의 결합도가 제거됨.

DI(Dependency Injection) 유형

■ Setter Injection

- Setter method를 이용한 의존성 삽입
- 의존성을 입력 받는 Setter Method를 만들고, 이를 통해 의존성을 주입한다.

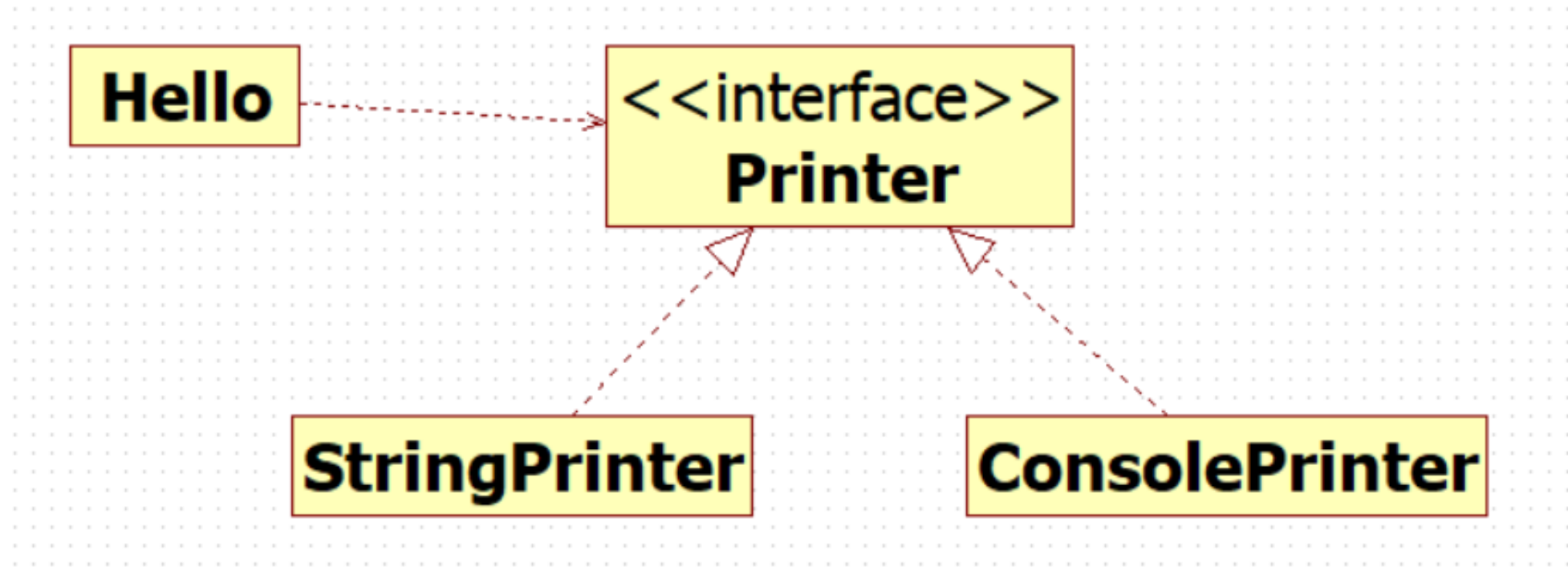
■ Constructor Injection

- 생성자를 이용한 의존성 삽입
- 필요한 의존성을 포함하는 Class의 생성자를 만들고 이를 통해 의존성을 주입한다.

■ Method Injection

- 일반 Method를 이용한 의존성 삽입
- 의존성을 입력받는 일반 method를 만들고 이를 통해 의존성을 주입한다.

DI(Dependency Injection)를 이용한 Class 호출방식



- **Hello** Class가 직접 **StringPrinter**나 **ConsolePrinter**를 찾아서 사용하는 것이 아니라 설정 File(Spring Bean Configuration File)에 설정하면 Container가 연결해준다.

Setter Injection

■ beans.xml

```
<bean id="hello" class="bean.Hello">
```

```
<!--bean은 Spring이 관리해주는 객체라는 뜻 -->
```

```
    <property name="name" value="Spring" />
```

```
    <property name="printer" ref="printer" />
```

```
</bean>
```

```
<bean id="printer" class="bean.StringPrinter" />
```

```
<bean id="consolePrinter" class="bean.ConsolePrinter" />
```

Setter Injection (Cont.)

■ Hello.java

```
package bean;

public class Hello{
    String name;
    Printer printer;
    public Hello(){}
    public void setName(String name){
        this.name = name;
    }
    public void setPrinter(Printer printer){
        this.printer = printer;
    }
}
```

Constructor Injection

■ beans.xml

```
<bean id="hello" class="bean.Hello">
```

```
<!--bean은 Spring이 관리해주는 객체라는 뜻 -->
```

```
    <constructor-arg index="0" value="Spring" />
```

```
    <constructor-arg index="1" ref="printer" />
```

```
</bean>
```

```
<bean id="printer" class="bean.StringPrinter" />
```

```
<bean id="consolePrinter" class="bean.ConsolePrinter" />
```

Constructor Injection (Cont.)

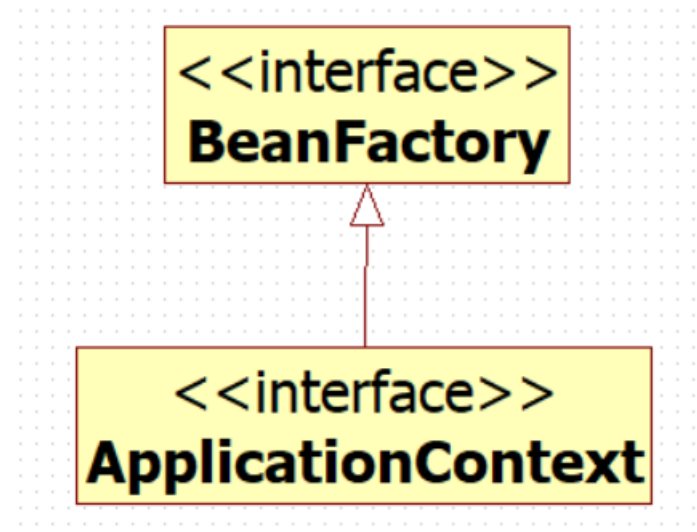
■ Hello.java

```
package bean;

public class Hello{
    String name;
    Printer printer;
    public Hello(){}
    public Hello(String name, Printer printer){
        this.name = name;
        this.printer = printer;
    }
}
```

Spring DI Container

- Spring DI Container가 관리하는 객체를 빈(Beans)이라고 하고, 이 Beans들을 관리한다는 의미로 Container를 빈 팩토리(BeanFactory)라고 부른다.
- 객체의 생성과 객체 사이의 런타임(run-time) 관계를 DI 관점에서 볼 때는 Container를 **BeanFactory**라고 한다.
- BeanFactory에 여러 가지 Container 기능을 추가하여 어플리케이션 컨텍스트(**ApplicationContext**)라고 부른다.



Spring DI Container (Cont.)

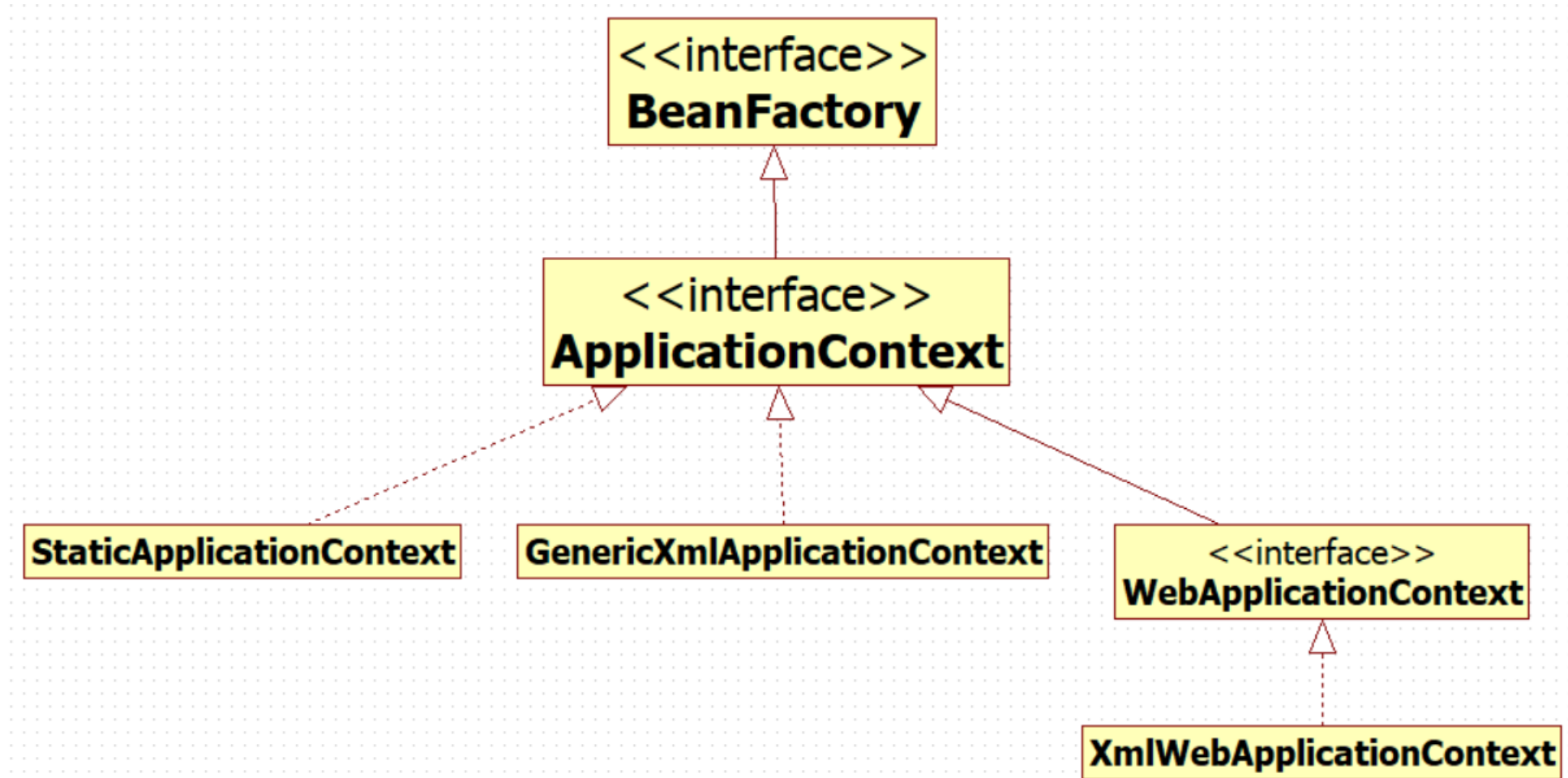
■ BeanFactory

- Bean을 등록, 생성, 조회, 반환 관리함
- 보통은 **BeanFactory**를 바로 사용하지 않고, 이를 확장한 **ApplicationContext**를 사용함
- **getBean()** method가 정의되어 있음.

■ ApplicationContext

- Bean을 등록, 생성, 조회, 반환 관리하는 기능은 **BeanFactory**와 같음.
- Spring의 각종 부가 Service를 추가로 제공함.
- Spring이 제공하는 **ApplicationContext** 구현 class가 여러가지 종류가 있음.

Spring DI Container (Cont.)



Spring DI Terminology

■ Bean

- Spring이 IoC 방식으로 관리하는 객체라는 뜻
- Spring이 직접 생성과 제어를 담당하는 객체를 Bean이라고 부른다.

■ BeanFactory

- Spring의 IoC를 담당하는 핵심 Container
- Bean을 등록, 생성, 조회, 반환하는 기능을 담당.
- **BeanFactory**를 바로 사용하지 않고 이를 확장한 **ApplicationContext**를 주로 이용

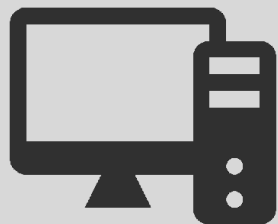
Spring DI Terminology (Cont.)

■ **ApplicationContext**

- **BeanFactory**를 확장한 IoC Container
- Bean을 등록하고 관리하는 기능은 **BeanFactory**와 동일하지만 Spring이 제공하는 각종 부가 service를 추가로 제공
- Spring에서는 **ApplicationContext**를 **BeanFactory**보다 더 많이 사용

■ Configuration metadata

- **ApplicationContext** 또는 **BeanFactory**가 IoC를 적용하기 위해 사용하는 meta정보
- 설정 meta정보는 IoC Container에 의해 관리되는 Bean 객체를 생성하고 구성할 때 사용됨.



THE LAB

Task 3. 간단한 DI Project



jUnit의 개요와 특징

- TDD의 창시자인 Kent Beck과 Design Pattern 책의 저자인 Erich Gamma가 작성
- 단정(Assert) method로 Test Case의 수행 결과를 판별
 - **assertEquals**(예상 값, 실제 값)
- jUnit4부터는 test를 지원하는 annotation 제공, **@Test**, **@Before**, **@After**
- 각 **@Test** method가 호출할 때마다 새로운 instance를 생성하여 독립적인 test가 이루어지도록 한다.

jUnit Library 설치

- <http://mvnrepository.com>
- jUnit으로 검색
- jUnit 4.12 version을 pom.xml에 추가

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

- pom.xml > right-click > Run As > Maven Install

jUnit에서 Test를 지원하는 Annotation

■ @Test

- 이것이 선언된 method는 test를 수행하는 method가 된다.
- jUnit은 각각의 test가 서로 영향을 주지 않고 독립적으로 실행됨을 원칙으로 함으로 **@Test** 마다 객체를 생성한다.

■ @Ignore

- 이것이 선언된 method는 test를 실행하지 않게 한다.

■ @Before

- 이것이 선언된 method는 **@Test**가 실행되기 전에 반드시 실행된다.
- **@Test** method에서 공통으로 사용하는 code를 **@Before** method에 선언하여 사용하면 된다.

jUnit에서 Test를 지원하는 Annotation (Cont.)

■ **@After**

- 이것이 선언된 method는 **@Test** method가 실행된 후 실행된다.

■ **@BeforeClass**

- 이 annotation은 **@Test** method보다 먼저 한번만 수행되어야 할 경우에 사용하면 된다.

■ **@AfterClass**

- 이 annotation은 **@Test** method보다 나중에 한번만 수행되어야 할 경우에 사용하면 된다.

Test 결과를 확인하는 단정(Assert) Method 종류

■ `assertEquals(a, b)`

- 객체 a와 b가 일치함을 확인

■ `assertArrayEquals(a, b)`

- 배열 a, b가 일치함을 확인

■ `assertSame(a, b)`

- 객체 a, b가 같은 객체임을 확인
- `assertEquals()` method는 값이 같은지를 확인하는 것이고, `assertSame()` method는 두 객체의 reference가 같은지를 확인한다.(==연산자)

■ `assertTrue(a)`

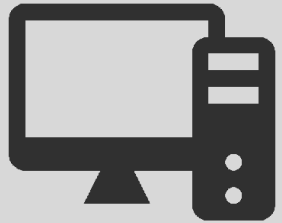
- 조건 a가 참인가를 확인

■ `assertNotNull(a)`

- 객체 a가 null이 아님을 확인한다.

■ 이외에도 다양한 assert method가 존재함

■ <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>



THE LAB

Task 4. jUnit을 사용한 DI Test Class 작성하기

Spring TestContext Framework

■ @RunWith (SpringJUnit4ClassRunner.class)

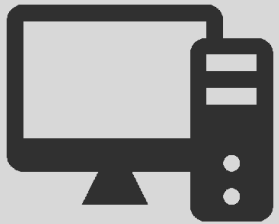
- JUnit Framework의 test 실행방법을 확장할 때 사용하는 Annotation
- **SpringJUnit4ClassRunner**라는 class를 지정해주면 JUnit이 Test를 진행하는 중에 **ApplicationContext**를 만들고 관리하는 작업을 진행해 준다.
- Annotation은 각각의 test 별로 객체가 생성되더라도 Singleton의 **ApplicationContext**를 보장한다.

■ @ContextConfiguration

- Spring bean 설정 file의 위치를 지정할 때 사용되는 Annotation

■ @Autowired

- Spring DI에서 사용되는 특별한 Annotation
- 해당 변수에 자동으로 빈(Bean)을 매핑해준다.
- Spring bean 설정 file을 읽기 위해 굳이 **GenericXmlApplicationContext**를 사용할 필요가 없다.



THE LAB

Task 5. Spring TestContext Framework



Dependency Injection(의존주입) 방법의 종류

- XML file을 이용한 DI 설정 방법
 - setter 이용하기
 - 생성자 이용하기
- Java Annotation 이용한 DI 설정 방법
- Java Annotation과 XML 을 이용한 DI 설정 방법
 - XML file에 Java file을 포함시켜 사용하는 방법
 - Java file에 XML file을 포함시켜 사용하는 방법

Setter를 이용한 의존주입하기 – Setter Injection

- Setter method를 통해 의존 관계가 있는 bean을 주입하려면 **<property>** 태그를 사용할 수 있다.
- **ref** 속성은 사용하면 bean이름을 이용해서 주입할 bean을 찾는다.
- **value** 속성은 단순 값 또는 bean이 아닌 객체를 주입할 때 사용한다.
- 단순 값(문자열이나 숫자)의 주입
 - Setter method를 통해 bean의 레퍼런스가 아니라 단순 값을 주입하려고 할 때는 **<property>** 태그의 **value**속성을 사용한다.

```
public void setName(String name) {  
    this.name = name;  
}
```

```
<bean id="hello" class="com.example.Hello">  
    <property name="name" value="Spring" />  
</bean>
```

Setter를 이용한 의존주입하기 – Setter Injection (Cont.)

■ Collection 타입의 값 주입

- Spring은 List, Set, Map, Properties와 같은 Collection 타입을 XML로 작성해서 property에 주입하는 방법을 제공한다.
- List 타입 : `<list>`와 `<value>` 태그를 이용
- Set 타입 : `<set>`과 `<value>` 태그를 이용
- Map 타입 : `<map>`과 `<entry>` 태그를 이용
- Properties 타입 : `<props>`와 `<prop>`를 이용

Setter를 이용한 의존주입하기 – Setter Injection (Cont.)

■ Collection 타입의 값 주입

- List 타입 : `<list>`와 `<value>` 태그를 이용
- Set 타입 : `<set>`과 `<value>` 태그를 이용

```
public class Hello{  
    List<String> names;  
    public void setNames(List<String> list) {  
        this.names = list;  
    }  
}
```

Setter를 이용한 의존주입하기 – Setter Injection (Cont.)

■ Collection 타입의 값 주입

- List 타입 : `<list>`와 `<value>` 태그를 이용
- Set 타입 : `<set>`과 `<value>` 태그를 이용

```
<bean id="hello" class="com.example">
    <property name="names">
        <list>
            <value>Spring</value>
            <value>IoC</value>
            <value>DI</value>
        </list>
    </property>
</bean>
```


Setter를 이용한 의존주입하기 – Setter Injection (Cont.)

■ Collection 타입의 값 주입

- List 타입 : `<list>`와 `<value>` 태그를 이용
- Set 타입 : `<set>`과 `<value>` 태그를 이용

```
<bean id="hello" class="com.example">
```

```
    <property name="foods">
```

```
        <set>
```

```
            <value>Chicken</value>
```

```
            <value>Pizza</value>
```

```
            <value>Bread</value>
```

```
        </set>
```

```
    </property>
```

```
</bean>
```

Setter를 이용한 의존주입하기 – Setter Injection (Cont.)

- Collection 타입의 값 주입
 - Map 타입 : `<map>`과 `<entry>` 태그를 이용

```
public class Hello{  
    Map<String, Integer> ages;  
  
    public void setAges(Map<String, Integer> ages) {  
        this.ages = ages;  
    }  
}
```

Setter를 이용한 의존주입하기 – Setter Injection (Cont.)

■ Collection 타입의 값 주입

- Map 타입 : `<map>`과 `<entry>` 태그를 이용

```
<bean id="hello" class="com.example.Hello">
  <property name="ages">
    <map>
      <entry key="나훈아" value="30" />
      <entry key="이미자" value="50" />
      <entry>
        <key>
          <value>설운도</value>
        </key>
        <value>60</value>
      </entry>
    </map>
  </property>
</bean>
```

Setter를 이용한 의존주입하기 – Setter Injection (Cont.)

■ Collection 타입의 값 주입

- Properties 타입 : **<props>**와 **<prop>**를 이용

```
<bean id="hello" class="com.example.Hello">
  <property name="ages">
    <props>
      <prop key="나훈아">서울시 강남구 역삼동</prop>
      <prop key="이미자">경기도 수원시 장안구</prop>
    </props>
  </property>
</bean>
```

Setter를 이용한 의존주입하기 – Setter Injection (Cont.)

■ Collection 타입의 값 주입

- null값 추가

```
<set>
    <value>Element 1</value>
    <value>Element 2</value>
    <null />
</set>
```

```
<map>
    <entry>
        <key>
            <null />
        </key>
        <null />
    </entry>
</map>
```

Setter를 이용한 의존주입하기 – Setter Injection (Cont.)

■ 배열의 값 지정

```
<property name="">  
    <array>  
        <value>1</value>  
        <value>2</value>  
    </array>  
</property>
```

Setter를 이용한 의존주입하기 – Setter Injection (Cont.)

- 실제 Application 개발 Scenario에서 사용되는 Spring bean의 속성과 생성자 인자 형식은 String 형식, 다른 bean의 참조, 여러 표준 형식 (`java.util.Date`, `java.util.Map` 등) 또는 사용자 지정 형식(예, `Address`)까지 매우 다양하다.
- `java.util.Date`, `java.util.Currency`, 기본 형식 등의 bean 속성과 생성자 인자를 간편하게 전달하기 위해 Spring 에서는 기본적으로 `PropertyEditor`를 제공하고 있다.

Java Annotation을 이용한 의존주입하기

■ **@Configuration** Annotation

- 해당 Class를 Spring 설정 Class로 지정.

```
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
```

```
public class AppCtx {
```

```
    ...
```

```
    ...
```

```
}
```


Java Annotation을 이용한 의존주입하기 (Cont.)

■ @Bean Annotation

- Method에 붙여서 해당 Method가 생성한 객체를 Spring이 관리하는 Bean 객체로 등록.
- 해당 Method의 이름은 Bean객체를 구분할 때 사용.

```
import org.springframework.context.annotation.Bean;  
  
@Bean  
public Hello hello() {  
    Hello hello = new Hello();  
    ...  
    return hello;  
}
```

Java Annotation을 이용한 의존주입하기 (Cont.)

```
10 @Configuration
11 public class AppCtx {
12
13     @Bean
14     public Hello hello() {
15         Hello hello = new Hello();
16         hello.setName("Spring");
17         hello.setPrinter(this.printer());
18         return hello;
19     }
20
21     @Bean
22     public StringPrinter printer() {
23         return new StringPrinter();
24     }
25 }
```

Bean의 Setter Method를 이용하여
의존 주입

Bean의 Setter Method를 이용하여
의존 주입

Java Annotation을 이용한 의존주입하기 (Cont.)

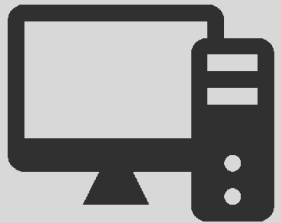
```
3 import org.springframework.context.ApplicationContext;
4 import org.springframework.context.annotation.AnnotationConfigApplicationContext;
5
6 import com.example.Hello;
7 import com.example.Printer;
8 import com.example.config.AppCtx;
9
10 public class HelloBeanTest {
11     public static void main(String[] args) {
12         // 1. IoC Container 생성
13         ApplicationContext ctx = new AnnotationConfigApplicationContext(AppCtx.class);
14
15         // 2. Hello Beans 가져오기
16         Hello hello = (Hello)ctx.getBean("hello");
17         System.out.println(hello.sayHello());
18         hello.print();
19
20         // 3. SpringPrinter 가져오기
21         Printer printer = (Printer) ctx.getBean("printer");
22         System.out.println(printer.toString());
23         Hello hello2 = ctx.getBean("hello", Hello.class);
24         hello2.print();
25         System.out.println(hello == hello2);
26     }
27 }
```

Java 설정에서 정보를 읽어서 Bean 객체를 생성하고 관리

@Configuration으로 선언한 Java 설정 Class의 Instance

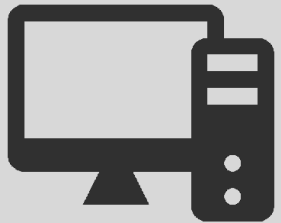
AnnotationConfigApplicationContext가 Java 설정을 읽어서 생성한 Bean 객체를 검색할 때 사용

첫 번째 Parameter는 @Bean의 Method 이름 (Bean객체의 이름),
두 번째 Parameter는 검색할 Bean 객체의 Type.



THE LAB

Task 6. Java Annotation을 이용하여 setter를 이용한 의존주입하기 실습



THE LAB

Task 7. Setter를 이용한 의존주입하기 실습

생성자를 이용한 의존주입하기 – Constructor Injection

- Constructor를 통해 의존관계가 있는 Bean을 주입하려면 **<constructor-arg>** tag를 사용할 수 있다.
- Constructor 주입방식은 생성자의 parameter를 이용하기 때문에 한번에 여러 개의 객체를 주입할 수 있다.

표준 Java 형식 및 사용자 지정 형식을 지정하는 생성자 인자

- 생성자 인자 형식이 Primitive type, String 형식 또는 사용자 지정 형식(Address 같은...)인 경우 **<constructor-arg>** 요소의 **value** 속성으로 값을 지정한다.
- **value** 속성으로 지정한 문자열 값을 두 개 이상의 생성자 인자로 변환할 수 있는 경우 Spring container가 생성자 인자의 형식을 유추할 수 없다.
- 예를 들어, 값이 int, long 또는 String 중 어떤 형식인지 알 수 없는 경우가 있다.
- 이런 경우 **type** 속성을 사용해 생성자 인자의 형식을 명시적으로 지정할 수 있다.

표준 Java 형식 및 사용자 지정 형식을 지정하는 생성자 인자 (Cont.)

- 아래의 Student class의 생성자를 보면

```
public Student(String name, int age, int grade, int classNum)
{
    this.name = name;
    this.age = age;
    this.grade = grade;
    this.classNum = classNum;
}
```


표준 Java 형식 및 사용자 지정 형식을 지정하는 생성자 인자 (Cont.)

- Student class에 주입되는 bean 설정 file의 형식은 아래와 같다.

```
<bean id="student2" class="com.example.Student">  
    <constructor-arg value="설운도" />  
    <constructor-arg value="16" />  
    <constructor-arg value="3" />  
    <constructor-arg value="7" />  
</bean>
```

표준 Java 형식 및 사용자 지정 형식을 지정하는 생성자 인자 (Cont.)

- Spring container는 Student bean 정의에서 **<constructor-arg>** 요소가 나온 순서대로 생성자에 적용한다.
- 이러한 모호함을 해결하기 위해서 다음 Code처럼 **type** 속성으로 생성자 인자의 형식을 지정할 수 있다.

```
<bean id="student2" class="com.example.Student">  
    <constructor-arg type="java.lang.String" value="설운도" />  
    <constructor-arg type="int" value="16" />  
    <constructor-arg type="int" value="3" />  
    <constructor-arg type="int" value="7" />  
</bean>
```

이름을 기준으로 한 생성자 인자 연결

- **<constructor-arg>**요소의 **name** 속성에는 **<constructor-arg>** 요소가 적용되는 생성자 인자의 이름을 지정한다.
- 아래 Code에서 Student class의 생성자를 보면

```
public Student(String name, int age, int grade, int classNum)
{
    this.name = name;
    this.age = age;
    this.grade = grade;
    this.classNum = classNum;
}
```

이름을 기준으로 한 생성자 인자 연결 (Cont.)

- **<constructor-arg>** 요소의 **name** 속성으로 이 요소가 적용될 생성자 인자의 이름을 지정한다.

```
<bean id="student2" class="com.example.Student">
    <constructor-arg name="name" value="설운도" />
    <constructor-arg name="age" value="16" />
    <constructor-arg name="grade" value="3" />
    <constructor-arg name="classNum" value="7" />
</bean>
```

이름을 기준으로 한 생성자 인자 연결 (Cont.)

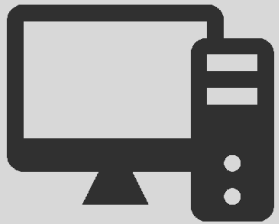
- 이 구성은 해당 class를 compile할 때 debug flag를 활성화해야 제대로 작동한다.
- Debug flag를 활성화하면 생성된 .class file에 생성자 인자 이름이 유지된다.
- 만일 debug flag를 활성화하지 않고 compile하면 compile 중에 생성자 이름이 손실되므로 Spring이 **<constructor-arg>** 요소의 **name** 속성에 지정된 생성자 인자 이름에 해당하는 생성자 인자를 찾을 수 없다.
- 만일 debug flag를 활성화하고 class를 compile하고 싶지 않은 경우, **@ConstructorProperties** annotation(Java SE 6에서 추가됨)을 사용해서 생성자 인자 이름을 명시적으로 지정하면 된다.

이름을 기준으로 한 생성자 인자 연결 (Cont.)

```
public class Student {  
    private String name;  
    private int age;  
    private int grade;  
    private int classNum;  
  
    @ConstructorProperties({"name", "age", "grade", "classNum"})  
    public Student(String name, int age, int grade, int classNum) {  
        this.name = name;  
        this.age = age;  
        this.grade = grade;  
        this.classNum = classNum;  
    }  
}
```

이름을 기준으로 한 생성자 인자 연결 (Cont.)

- 이 예제에서는 **@ConstructorProperties** annotation에 생성자 인자의 이름이 bean class의 생성자에 나오는 순서대로 지정돼 있다.
- 이름을 지정할 때는 **<constructor-arg>** 요소의 생성자 인자 이름과 완전히 동일하게 지정해야 한다.



THE LAB

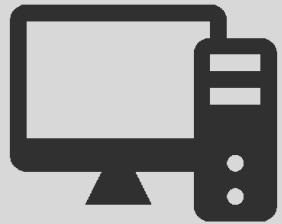
Task 8. 생성자를 이용하여 의존주입하기 실습

Java Annotation을 이용한 의존주입하기

```
9 @Configuration
10 public class ApplicationCtx {
11     @Bean
12     public Student student1() {
13         return new Student("한지민", 15, 2, 5);
14     }
15
16     @Bean
17     public Student student2() {
18         return new Student("김지민", 16, 3, 7);
19     }
20
21     @Bean
22     public StudentInfo studentInfo() {
23         return new StudentInfo(this.student1());
24     }
25 }
```

Bean의 생성자를 이용하여
의존 주입

Bean의 생성자를 이용하여
의존 주입



THE LAB

Task 9. Java Annotation을 이용한 생성 자를 이용하여 의존주입하기 실습

@ConstructorProperties Annotation과 Bean 정의 상속

- 부모 bean 정의에 해당하는 class의 생성자에 **@ConstructorProperties** Annotation을 지정한 경우 자식 Bean 정의에 해당하는 Bean class에도 **@ConstructorProperties** Annotation을 지정해야 한다.

```
<bean id="human" class="com.example.Human">  
    <constructor-arg name="name" value=" 김지민 " />  
</bean>
```

```
<bean id="student" class="com.example.Student" parent="human">  
    <constructor-arg name="age" value="16" />  
    <constructor-arg name="grade" value="3" />  
    <constructor-arg name="classNum" value="7" />  
</bean>
```

@ConstructorProperties Annotation과 Bean 정의 상속 (Cont.)

- 이 예제에서 **human** Bean 정의는 추상이 아니므로(만일 추상이었다면 **abstract="true"**라고 선언해야 함), Spring Container는 이 Bean의 Instance를 생성한다.
- **human** Bean의 **<constructor-arg>** 구성은 자식 Bean 정의인 **student**으로 상속된다.
- 자식 생성자에 **@ConstructorProperties** Annotation을 지정하지 않으면 Spring Container가 상속된 **<constructor-arg>** 요소와 자식 class의 생성자에 지정된 생성자 인자를 연결할 수 없다.
- 하지만, static 또는 instance Factory Method의 인자를 이름으로 전달하는 데는 **@ConstructorProperties** Annotation을 사용할 수 없다.

@ConstructorProperties Annotation과 Factory Method

■ static Factory Method를 이용한 Bean Instance화하기

- 만일 Calendar class가 Instance를 생성하기 위해 **Calendar.getInstance()**를 사용하는 것처럼 static Factory Method를 이용한 Bean Instance화에 대해 살펴보자.
- 이럴 경우에 Spring 에서는 아래처럼 Bean 정의를 해야 한다.

```
public class CalendarFactory{  
    private CalendarFactory() {}
```

```
    public static Calendar getInstance(String country) {  
        Calendar calendar = null;
```

```
        ...
```

```
        return calendar;
```

```
    }
```

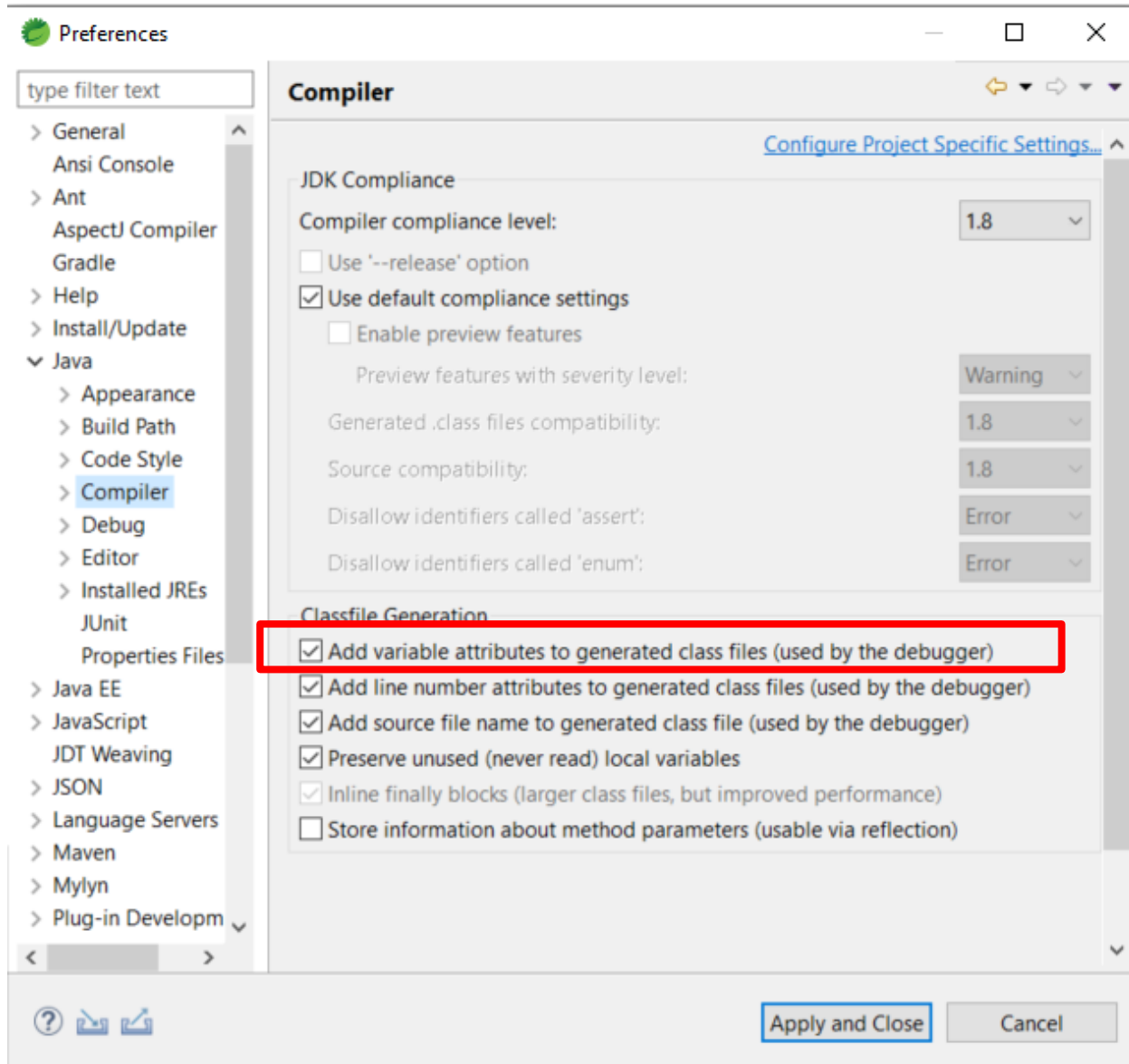
```
}
```

```
<bean id="myCalendar" class="java.util.Calendar" factory-method="getInstance"  
>  
    <constructor-arg index="0" value="kor" />  
</bean>
```

@ConstructorProperties Annotation과 Factory Method (Cont.)

- 만일 이때, **<constructor-arg>**요소의 **name** 속성을 지정하고 Factory Method에 **@ConstructorProperties** Annotation을 지정하면 **static** 및 instance Factory Method에 인자를 이름으로 전달할 수 있지 않을까?
- 하지만, **@ConstructorProperties** Annotation은 생성자에만 지정할 수 있으며, Method에는 지정할 수 없다.
- 즉, **static** 또는 instance Factory Method에 인자를 이름으로 전달하려면 **Debug Flag**를 활성화하고 Class를 Compile하는 것이 유일한 방법이다.
- **Debug Flag**를 활성화하고 Class를 Compile하면, .class File의 크기는 커지지만 Application의 실행 성능에는 영향을 주지 않으며, Class를 Loading 하는 시간만 약간 늘어난다.

Eclipse IDE에서 Debug flag 활성화/비활성화하는 방법



- Check → Debug Flag 활성화
- Uncheck → Debug Flag 비활성화

DI의 장점

- Java file의 수정 없이 Spring 설정 file만을 수정하여 부품들을 생성/조립할 수 있다.

```
package com.example;
```

```
public interface Car {  
    void drive();  
}
```


DI의 장점 (Cont.)

```
package com.example;

public class Sonata implements Car {

    @Override
    public void drive() {
        System.out.println("Drive a Sonata");
    }

}
```

```
package com.example;

public class Carnival implements Car {

    @Override
    public void drive() {
        System.out.println("Drive a Carnival");
    }

}
```

DI의 장점 (Cont.)

```
package com.example;

public class HybridCar extends Sonata implements Car {
    @Override
    public void drive(){
        System.out.println("Drive a HybridCar with Sonata");
    }
}
```

DI의 장점 (Cont.)

■ CarContext.xml

```
<!-- <bean id="car" class="com.example.Sonata" /> -->
```

```
<!-- <bean id="car" class="com.example.Carnival" /> -->
```

```
<bean id="car" class="com.example.HybridCar" />
```

//CarMainClass를 변경하지 않고,

//CarContext.xml만 변경해도 여러 class를 이용할 수 있다.

DI의 장점 (Cont.)

■ CarMainClass.java

```
public class CarMainClass {  
    public static void main(String[] args) {  
        String configFile = "classpath:CarContext.xml";  
        AbstractApplicationContext context =  
            new GenericXmlApplicationContext(configFile);  
        Car car = context.getBean("car", Car.class);  
        car.drive();  
  
        context.close();  
    }  
}
```

생성자 DI vs Setter DI

■ 생성자 방식

- 장점

- Bean객체를 생성하는 시점에 모든 의존 객체 주입

- 단점

- Parameter의 개수가 많을 경우 각 인자가 어떤 의존 객체를 설정하는지 알기 위해 생성자의 Code를 일일이 확인해야 하는 어려움

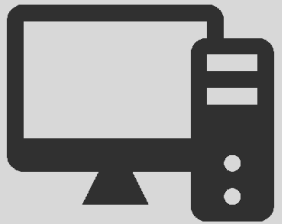
■ Setter 방식

- 장점

- Setter 이름을 통해 어떤 객체가 주입되는지 알 수 있음.

- 단점

- Setter를 사용해서 필요한 의존 객체를 전달하지 않아도 Bean객체가 생성되기 때문에 객체를 사용하는 시점에 **NullPointerException**이 발행할 가능성 있음.



THE LAB



Task 10. Context file 여러 개 사용하기

p 및 c Namespace를 이용해 간결하게 Bean 정의 작성하기

- Spring은 각각 Bean 속성과 생성자 인자의 값을 지정할 수 있는 p 및 c Namespace를 제공함으로써 Application Context XML File에서 Bean 정의를 간결하게 작성하도록 돕는다.
- p 및 c namespace는 각각 **<property>** 및 **<constructor-arg>** 요소 대신 사용할 수 있다.

p 및 c Namespace를 이용해 간결하게 Bean 정의 작성하기 (Cont.)

■ p Namespace

- p Namespace를 사용해서 Bean 속성을 설정하려면 Bean 속성을 **<bean>** 요소의 속성으로 지정하고 p Namespace 안에서 각 Bean 속성을 지정한다.

```
<bean id="student2" class="com.example.Student"
      p:name="홍길동" p:age="24"
      p:height="179.4" p:weight="68.4" />
```

- Bean 속성이 Bean 참조가 아닌 경우 다음 구분을 사용해 지정한다.

```
p:<속성이름>="<속성 값>"
```

- Bean 속성이 Bean 참조인 경우 다음 구분을 사용해 지정한다.

```
p:<속성이름>-ref="<bean 참조>"
```


p 및 c Namespace를 이용해 간결하게 Bean 정의 작성하기 (Cont.)

■ c Namespace

- c Namespace를 사용해 생성자 인자의 값을 제공하려면 생성자 인자를 **<bean>** 요소의 속성으로 지정하고 c Namespace 안에서 각 생성자 인자를 지정한다.

```
public class Student{  
    ...  
    @ConstructorProperties({"name", "age", "height", "weight" })  
    public Student(String name, int age, double height, double weight){  
        ...  
    }  
}
```

```
<bean id="student2" class="com.example.Student"  
    c:name="한지민" c:age="24" c:height="158.3" c:weight="52.4" />
```

p 및 c Namespace를 이용해 간결하게 Bean 정의 작성하기 (Cont.)

- 생성자 인자가 Bean 참조가 아닌 경우 다음 구문을 사용해 지정한다.

c:<생성자 인자 이름>=<생성자 인자 값>

- 생성자 인자가 Bean 참조인 경우 다음 구문을 사용해 지정한다.

c:<생성자 인자 이름>-ref=<bean 참조>

p 및 c Namespace를 이용해 간결하게 Bean 정의 작성하기 (Cont.)

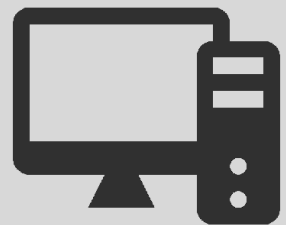
- Debug Flag를 활성화하고 Class를 Compile하면 생성된 .class File에 생성자 인자 이름이 보존된다.
- 만일 Debug Flag를 활성화하지 않고 Class를 Compile하면 예제는 작동되지 않는다.
- 이 경우에는 다음과 같이 Index를 사용해서 생성자 인자의 값을 지정하면 된다.

```
<bean id="student2" class="com.example.Student"
```

```
    c:_0="한지민" c:_1="24" c:_2="158.3" c:_3="52.4"/>
```

- XML에서는 속성 이름을 숫자 값으로 시작할 수 없기 때문에 생성자 인자의 index 앞에 밑줄을 붙인다.
- 생성자 인자가 다른 Bean의 참조인 경우 생성자 인자의 Index 뒤에 **-ref**를 붙여야 한다.

```
    c:_0-ref
```



THE LAB

Task 11. Java Annotation을 이용하여 두 개 이상의 설정 파일로 DI 설정하기

Bean 등록 메타 정보 구성 전략

■ 전략 1 - XML 단독 사용

- 모든 Bean을 명시적으로 XML에 등록하는 방법이다.
- 생성되는 모든 Bean을 XML에서 확인할 수 있다는 장점이 있으나 Bean의 갯수가 많아지면 XML File을 관리하기 번거로울 수 있다.
- 여러 개발자가 같은 설정 File을 공유해서 개발하다보면 설정 File을 동시에 수정하다가 충돌이 일어나는 경우도 적지 않다.
- DI에 필요한 적절한 Setter Method 또는 Constructor가 Code 내에 반드시 존재해야 한다.
- 개발 중에는 Annotation 설정방법을 사용했지만, 운영 중에는 관리의 편의성을 위해 XML 설정으로 변경하는 전략을 쓸 수도 있다.

Bean 등록 메타 정보 구성 전략 (Cont.)

■ 전략 2 - XML과 Bean Scanning의 혼용

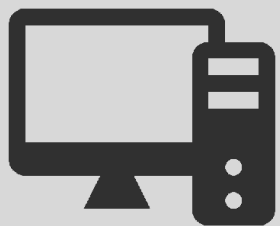
- Bean으로 사용될 Class에 특별한 Annotation을 부여해주면 이런 Class를 자동으로 찾아서 Bean으로 등록한다.
- 특정 Annotation이 붙은 Class를 자동으로 찾아서 Bean으로 등록해 주는 방식을 Bean Scanning을 통한 자동인식 Bean 등록기능이라고 한다.
- Annotation을 부여하고 자동 스캔으로 빈을 등록하면 XML 문서 생성과 관리에 따른 수고를 덜어주고 개발 속도를 향상시킬 수 있다.
- Application에 등록될 Bean이 어떤 것들이 있고, Bean들 간의 의존관계가 어떻게 되는지를 한눈에 파악할 수 없다는 단점이 있다.

Bean 등록 메타 정보 구성 전략 (Cont.)

■ 주의 사항

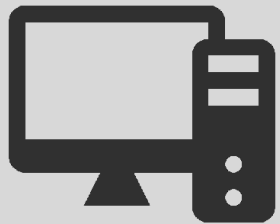
- Library형태로 제공되는 Class는 반드시 XML 설정을 통해서만 사용할 수 있다.
- 예를 들면, Apache에서 제공하는 **BasicDataSource** class를 사용하여 DB 연동을 처리한다면 commons-dbcp-1.4.jar file에 있는 **BasicDataSource** Class에 관련된 Annotation을 추가할 수 없다.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  <property name="driverClassName" value="org.h2.Driver" />
  <property name="url" value="jdbc:h2:tcp://localhost/~/test" />
  <property name="username" value="sa" />
  <property name="password" value="" />
</bean>
```



THE LAB

Task 12. Java Annotation과 XML을 이용한 DI 설정방법



THE LAB

Task 13. Java Annotation과 XML을 이용한 DI 설정방법

Bean등록 Annotation

■ @Component

- Component를 나타내는 일반적인 스테레오 타입으로 **<bean>** 태그와 동일한 역할을 함
- Spring Component Class를 지정하는 형식 단계의 Annotation
- 이 Annotation보다는 아래에 소개된 각각의 Annotation을 사용하는 것이 좋다.

■ @Repository

- Persistence Layer, 영속성을 가지는 속성(File, Database)을 가진 Class

■ @Service

- Service Layer, Business Logic을 가진 Class

■ @Controller

- Presentation Layer, Web Application에서 Web Request와 Response를 처리하는 Class

Bean등록 Annotation (Cont.)

- `@Repository`, `@Service`, `@Controller`는 더 특정한 UseCase에 대한 `@Component`의 구체화된 형태이다.

```
@Service(value = "TestService")
```

```
public class TestServiceImpl implements Service {...}
```

- 위의 코드처럼 `@Service` Annotation은 Bean을 Spring Container에 Component로 등록하는 데 사용할 이름을 지정하는 `value` 속성을 받는다.
- 즉, `TestServiceImpl` Class를 `TestService`라는 이름의 Bean으로 Spring Container에 등록했다.
- `value` 속성은 `<bean>` 요소의 `id` 속성과 동일한 역할을 한다.

Bean등록 Annotation (Cont.)

- **@Component, @Controller, @Repository** Annotation에서도 **value** 속성으로 Component의 이름을 지정할 수 있지만, **value** 속성 없이도 Spring Component의 이름을 지정할 수 있다.
- 즉, **@Service(value = "TestService")** 및 **@Service("TestService")**는 동일하다.
- Component의 이름을 지정하지 않으면 Spring은 Component의 이름이 Component class와 같다고 가정한다.
- 다만 Component의 이름은 소문자로 시작한다.

Bean등록 Annotation (Cont.)

- Component에 사용자 지정 이름을 지정하면 *이름을 기준으로* 자동 연결을 수행할 때 특히 유용하다.
- Spring의 Class 경로 검사 기능을 활성화한 경우 **@Component**, **@Controller**, **@Service** 또는 **@Repository** Annotation으로 지정한 Bean Class가 자동으로 Spring Container에 등록된다.
- Class 경로 검사 기능은 Spring Context Schema의 **<component-scan>** 요소를 사용해 활성화할 수 있다.

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

Bean 의존관계 주입 Annotation

- **@Autowired**, **@Resource** Annotation은 의존하는 객체를 자동으로 주입해 주는 Annotation이다.
- **@Autowired**는 Type으로, **@Resource**는 이름으로 연결한다는 점이 다르다.
- **@Autowired**
 - 정밀한 의존관계 주입(Dependency Injection)이 필요한 경우에 유용하다.
 - property, setter method, 생성자, 일반 method에 적용 가능하다.
 - 의존하는 객체를 주입할 때 주로 Type을 이용하게 된다.
 - **<property>**, **<constructor-arg>**태그와 동일한 역할을 한다.

Bean 의존관계 주입 Annotation (Cont.)

■ @Autowired

- 아래 Component Scan을 지원하는 Tag부분에서 언급했듯이 Bean 설정 File에서 **<context:component-scan>**를 설정해 주면 DI Container는 해당 Package에서 **@Autowired**가 붙은 Instance 변수의 형에 대입할 수 있는 Class를 **@Component**가 붙은 Class 중에서 찾아내 그 Instance를 Injection해 준다.
- 그리고 Instance 변수로 Injection은 Access Modifier가 **private**이라도 Injection할 수 있다.
- 만일 **@Autowired**가 붙은 객체가 Memory에 없다면 Container는 **NoSuchBeanDefinitionException**을 발생시킨다.
- 이 Annotation은 Setter Method를 필요로 하지 않게 한다.

@Autowired

private String name;

Bean 의존관계 주입 Annotation (Cont.)

■ @Autowired

- 이 Annotation을 사용할 때 형식과 일치하는 Bean이 발견되지 않으면 예외가 발생한다.
- 이 Annotation의 **required** 속성은 자동 연결 의존성이 필수 인지 또는 선택사항 인지 지정한다.
- 만일 **required** 속성을 **false**로 지정하면 의존성 자동 연결을 선택 사항으로 취급한다.
- 즉, 필수 형식과 일치하는 Bean이 발견되지 않아도 예외가 발생하지 않는다는 것을 의미한다.
- 기본적으로 **required** 속성은 **true**이므로 Spring Container가 의존성을 충족할 수 있어야 한다.

Bean 의존관계 주입 Annotation (Cont.)

■ @Resource

- Spring은 JSR 250의 이 Annotation을 통해 Field와 Method를 이름을 기준으로 자동 연결할 때 사용된다.
- Property, setter method에 적용 가능하다.
- 생성자 인자를 자동 연결하는 데는 사용할 수 없다.
- 의존하는 객체를 주입할 때 주로 **name**을 이용하게 된다.
- 이름을 기준으로 의존성을 자동 연결할 때는 **@Autowired** 및 **@Qualifier** Annotation보다는 **@Resource** Annotation을 사용하는 것이 좋다.
- 왜냐하면, Spring은 **@Autowired-@Qualifier** 조합을 사용하면 먼저 Field의 형식(또는 Method 인자나 생성자 인자의 형식)을 기준으로 Bean을 찾은 다음, **@Qualifier** Annotation으로 지정된 Bean이름으로 범위를 좁히는 과정을 거친다.

Bean 의존관계 주입 Annotation (Cont.)

■ @Resource

- 반면, @Resource Annotation을 사용하면 @Resource Annotation으로 지정된 Bean 이름으로 곧바로 고유한 Bean을 찾는다.
- 즉 @Resource Annotation을 사용하면 자동 연결할 Field(또는 Setter Method 인자)의 형식은 관계가 없다.

```
@Resource(name="stringPrinter")  
private Printer printer;
```

Bean 의존관계 주입 Annotation (Cont.)

■ @Value

- 단순한 값을 주입할 때 사용되는 Annotation이다.
- `@Value("Spring")`은 `<property ... value="Spring" />`와 동일한 역할을 한다.
- Field, Method, Method 매개변수 및 생성자 인자 단계에서 사용 가능하다.

```
@Value("#{configuration.environment}")  
private String environment;
```

```
@Value("#{configuration.getCountry()}")  
private String country;
```

Bean 의존관계 주입 Annotation (Cont.)

■ @Value

- *SpEL*(Spring Expression Language) 식을 **@Value** Annotation의 값으로 사용할 수도 있다.
- *SpEL*은 실행 시 객체를 조회 및 처리할 수 있는 식 언어다.
- **@Value** Annotation으로 지정한 *SpEL* 식은 **BeanPostProcessor**가 처리한다.
- *SpEL* 식은 **<Bean 이름>.<Field 또는 속성 또는 Method>** 형식을 사용해 값을 얻을 수 있다.

Bean 의존관계 주입 Annotation (Cont.)

■ @Qualifier

- **@Autowired** Annotation과 같이 사용된다.
- Field, Method Parameter 및 생성자 인자 단계에 사용 가능하다.
- **@Autowired**는 Type으로 찾아서 주입하므로, 동일한 Type의 Bean객체가 여러 개 존재할 때 특정 Bean을 찾기 위해서는 **@Qualifier**를 같이 사용해야 한다.
- 의존성 주입 대상이 되는 객체가 두 개 이상일 때 Container는 어떤 객체를 할당할지 스스로 판단할 수 없어서 Error를 발생한다.
- **NoUniqueBeanDefinitionException**이 발생한다.

Bean 의존관계 주입 Annotation (Cont.)

■ @Qualifier

- 아래의 Code처럼, Spring은 먼저 **@Autowired** Annotation이 지정된 Field, 생성자, Method에서 Type을 기준으로 자동 연결 후보를 찾을 다음, **@Qualifier** Annotation으로 지정된 Bean 이름을 사용해서 자동 연결 후보 목록에서 고유한 Bean을 찾는다.

```
@Autowired
```

```
@Qualifier("stringPrinter")
```

```
private Printer printer;
```

Bean 의존관계 주입 Annotation (Cont.)

■ @Inject와 @Named

- @Autowired와 동일한 기능을 제공
- Method, 생성자, field 단계에서 사용 가능.
- @Autowired와 달리 **required** 속성이 없으므로 의존성 자동 연결이 필수인지 선택인지를 지정할 수 없다.
- JSR 330에서 제안
- JSR 330(Java 의존성 주입)은 Java Platform을 위한 의존성 주입 Annotation을 표준화해서 Spring의 @Autowired 및 @Qualifier Annotation과 비슷한 @Inject 및 @Named Annotation을 정의하고 있다.

Bean 의존관계 주입 Annotation (Cont.)

■ @Inject와 @Named

- **@Named** Annotation은 Type 단계에 지정된 경우 Spring **@Component** Annotation과 비슷하게 작동하며, Method 매개변수나 생성자 인자 단계에 지정된 경우에는 Spring의 **@Qualifier** Annotation과 비슷하게 작동한다.
- 만일 **@Named** Annotation을 Class에 지정한 경우 Spring Context Schema의 **<component-scan>** 요소는 이 Class를 **@Component** Annotation이 지정된 Component Class와 같이 취급한다.
- **@Named** 및 **@Inject** Annotation을 사용하려면 JSR 330 JAR File을 Project에 포함시켜야 한다.

Bean 의존관계 주입 Annotation (Cont.)

■ @Inject와 @Named

- pom.xml에 다음과 같이 **<dependency>** 요소를 사용해서 JSR 330 JAR File을 포함시킨다.

```
<dependency>  
    <groupId>javax.inject</groupId>  
    <artifactId>javax.inject</artifactId>  
    <version>1</version>  
</dependency>
```

Bean 의존관계 주입 Annotation (Cont.)

■ @Scope

- Spring Component의 범위(Prototype or Singleton)를 지정
- 기본적으로 Singleton 범위를 가지며, Prototype 범위로 지정하기 위해 사용
- **<bean>** 요소의 **scope** 속성과 같은 역할
- **value** 속성 값으로 **SCOPE_SINGLETON**과 **SCOPE_PROTOTYPE** 상수를 지정할 수 있다.

```
import org.springframework.beans.factory.config.ConfigurableBeanFactory;  
import org.springframework.context.annotation.Scope;  
@Scope(value=ConfigurableBeanFactory.SCOPE_PROTOTYPE)  
public class Test {...}
```

Bean 의존관계 주입 Annotation (Cont.)

■ @Lazy

- 기본적으로 singleton 범위 Spring Component는 사전 초기화된다.
- 즉, Spring Container가 생성될 때 함께 Instance화 된다.
- Singleton 범위 구성 요소를 지연 생성하려면 Singleton 범위 Component의 Component Class에 **@Lazy** Annotation을 지정하면 된다.
- **<bean>** 요소의 **lazy-init** 속성과 같은 역할을 한다.

```
@Lazy(value=true)
```

```
@Component
```

```
public class Sample {...}
```

Bean 의존관계 주입 Annotation (Cont.)

■ @DependsOn

- 암시적 Bean 의존성을 지정하는데 사용

```
@DependsOn(value = {"beanA", "beanB"})  
@Component  
public class Sample {...}
```

- 위 코드는 Sample Class의 @DependsOn Annotation은 Sample Class의 Instance를 생성하기 전에 beanA 및 beanB Bean을 먼저 생성하도록 Spring Container에 지시한다.
- <bean> 요소의 depends-on 속성과 같은 역할을 한다.

Bean 의존관계 주입 Annotation (Cont.)

■ @Primary

- 의존성을 자동 연결을 후보가 여럿인 경우 특정한 Bean을 자동 연결의 기본 후보로 지정할 수 있다.
- `<bean>` 요소의 `primary` 속성과 같은 역할을 한다.

```
@Primary
```

```
@Component
```

```
public class Sample {...}
```

객체의 유효성 검사를 위한 Annotation

- JSR 303(bean 유효성 검사)의 Annotation을 사용해서 JavaBeans Component에 제약 조건을 지정할 수 있다.
- Bean 속성에 JSR 303 Annotation을 지정하면 Spring이 Bean 유효성 검사를 수행하고 결과를 제공한다.
- **@NotNull**
 - Annotation을 지정한 Field가 **null**일 수 없다.

@NotNull

private long id;

객체의 유효성 검사를 위한 Annotation (Cont.)

■ @Min

- Annotation을 지정한 Field가 지정된 값 이상이어야 한다.

```
@Min(1000)
```

```
@Max(500000)
```

```
private float depositAmount;
```

■ @Max

- Annotation을 지정한 Field가 지정된 값 이하여야 한다.

객체의 유효성 검사를 위한 Annotation (Cont.)

■ @NotBlank

- Annotation을 지정한 Field가 **null**이거나 비어 있을 수 없다.

```
@NotBlank
```

```
@Size(min=5, max=100)
```

```
private String email;
```

■ @Size

- Annotation을 지정한 Field의 크기가 지정된 **min** 및 **max** 특성 사이여야 한다.

Component Scan을 지원하는 태그

■ <context:component-scan> Tag

- @Component를 통해 자동으로 Bean을 등록하고, @Autowired로 의존관계를 주입 받는 Annotation을 Class에서 선언하여 사용했을 경우에는 해당 Class가 위치한 특정 Package를 Scan하기 위한 설정을 XML에 해주어야 한다.

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

- Spring Container에 자동 등록할 Component Class를 Filtering 하려면 <component-scan> 요소의 resource-pattern 속성을 사용한다.
- resource-pattern 속성의 기본값은 **/*.class이므로 base-package 속성으로 지정한 Package 이하의 모든 Component Class가 자동 등록된다.

Component Scan을 지원하는 태그 (Cont.)

- `<context:include-filter>` Tag와 `<context:exclude-filter>` Tag를 같이 사용하면 자동 Scan 대상에 포함시킬 Class와 포함시키지 않을 Class를 구체적으로 명시할 수 있다.

```
<beans ...>
```

```
    <context:component-scan base-package="com.example">
```

```
        <context:include-filter type="annotation"
```

```
            expression="com.example.annotation.MyAnnotation" />
```

```
        <context:exclude-filter type="regex"
```

```
            expression=".*Details" />
```

```
    </context:component-scan>
```

```
</beans>
```

- `<exclude-filter>` 및 `<include-filter>` 요소의 `type` 속성에는 Component Class를 Filtering하는 전략을 지정하며, `expression` 속성은 해당하는 Filter 식을 지정한다.

Component Scan을 지원하는 태그 (Cont.)

- 앞의 Code에서 보면, **<include-filter>** 요소에서는 **MyAnnotation** 형식 단계 Annotation을 지정한 Component Class를 Spring Container에 자동으로 등록하도록 지정했으며, **<exclude-filter>** 요소에서는 **<component-scan>** 요소에서 이름이 Detail로 끝나는 Component Class를 무시하도록 지정했다.

type	설명
annotation	expression 속성에는 Component Class에 사용해야 하는 Annotation의 정규화된 Class의 이름을 지정. 예를 들어 expression 속성값이 com.example.annotation.MyAnnotation 인 경우 MyAnnotation Annotation을 지정한 Component Class가 포함(<include-filter> 요소에 사용된 경우) 또는 제외(<exclude-filter> 요소에 사용된 경우).
assignable	expression 속성에는 Component Class를 할당할 수 있어야 하는 Class 또는 Interface의 정규화된 이름을 지정.
aspectj	expression 속성에는 Component Class를 Filtering 하는데 사용되는 AspectJ 식을 지정.
regex	expression 속성에는 Component Class를 이름을 기준으로 Filtering 하는데 사용되는 정규식을 지정.
custom	expression 속성에는 Component Class를 Filtering 하기 위한 org.springframework.core.type.TypeFilter Interface의 구현을 지정.

사용 예

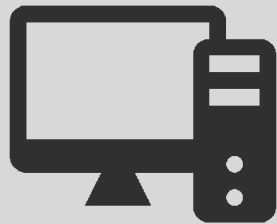
■ SpringPrinter.java

```
package com.example;

import org.springframework.stereotype.Component;

@Component("stringPrinter")
public class StringPrinter implements Printer{
    private StringBuffer buffer = new StringBuffer();
    public void print(String message){
        this.buffer.append(message);
    }

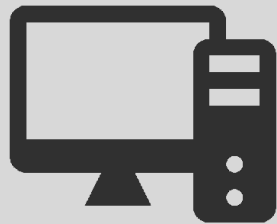
    public String toString(){
        return this.buffer.toString();
    }
}
```



THE LAB

Task 14. Lab





THE LAB

Task 15. Lab

