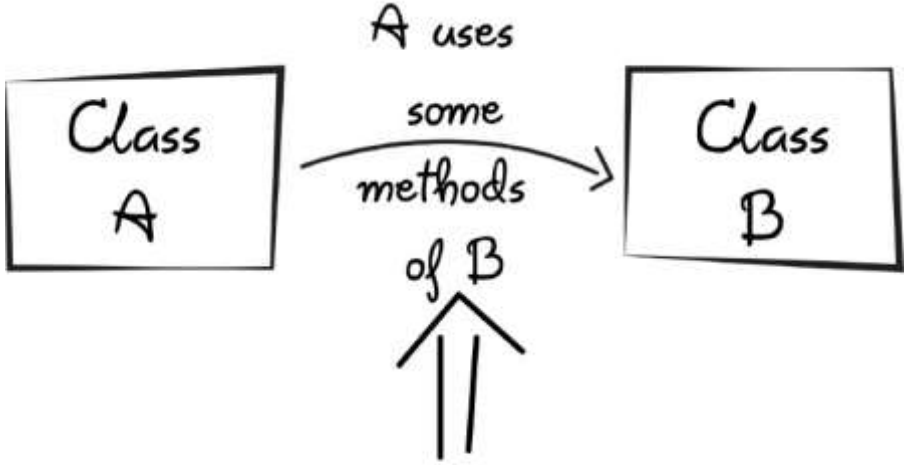
Dependency Injection

의존성 주입, 스프링에만 있는 개념은 아님

0. Object Dependencies



객체 사이에 의존성 있음

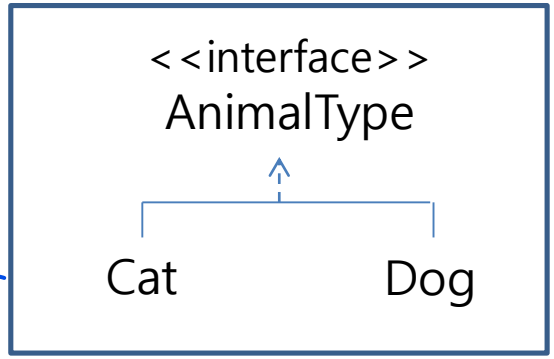
A가 B에 의존

Class A depends on Class B

spring

Object Dependencies

```
public class PetOwner{
    private AnimalType animal;

    public PetOwner() {
        this.animal = new Dog();
    }
}
```



우리는 이런식으로 new를 써서 객체를 생성해옴

- The PetOwner object depends on the AnimalType object(in this case, Dog)

하지만 이런 방식은 문제가 있다.

Problems with this approach

- The PetOwner object controls the creation of the AnimalType object
- There is **tight coupling** between the PetOwner and AnimalType objects
- So, changes in the AnimalType object will lead to the changes in PetOwner object

PetOwner와 Dog의 의존성이 너무 강해서

Dog을 Cat으로 바꾸려면 코드를 직접 수정해야함

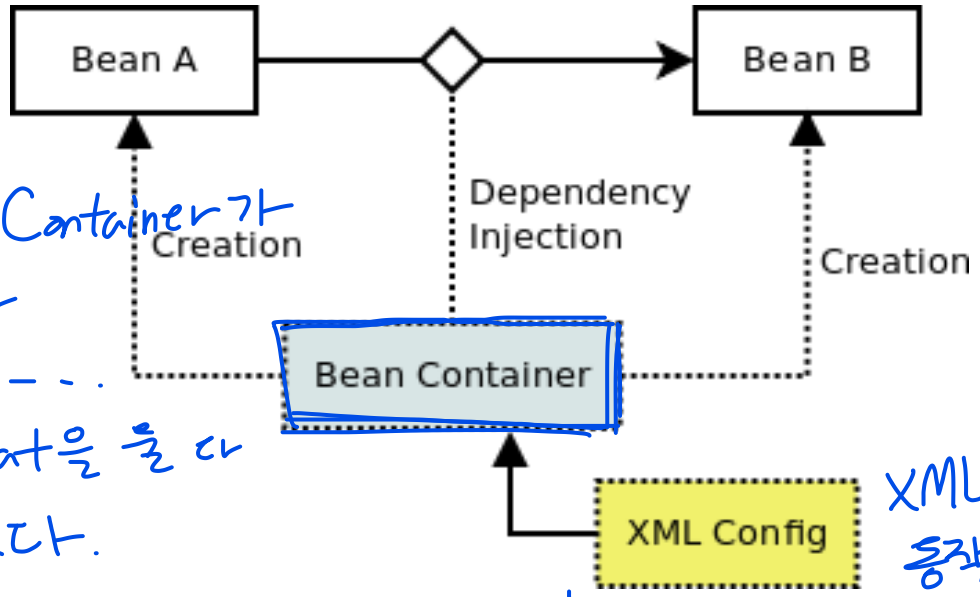
spring

(Bean = 객체)

1. Dependency Injection

Bean Container는 아주 중요한 역할을 한다.
Bean 들을 생성하고, Bean 들을 포함한다

Bean을 사이에
의존성이 있으면
외부에 있는 Bean Container가
의존성을 주입한다



이로써 dog 와 cat을 둘 다
가질 수 있게 됐다.

XML로 Bean Container의 동작을 설정

의존성은 이제 다 빈 컨테이너가 제어함

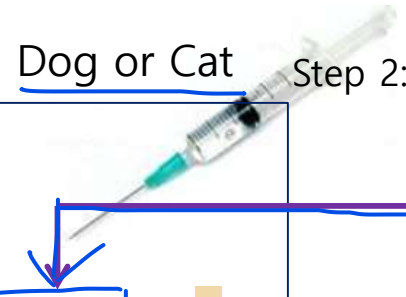
① ②

- Bean Container creates beans and performs dependency injection as specified in configuration instructions
- IoC(Inversion of Control) is also known as dependency injection (DI)

spring

Dependency Injection

생성자의 매개변수인 animal에 Dog 이나 Cat을 넣는 방식,
이렇게 하면 의존성이 완화됨



Dog or Cat

Step 2: Passes the AnimalType object to the PetOwner object (Dependency Injection)

```
public class PetOwner{
    private AnimalType animal;

    public PetOwner(AnimalType animal) {
        this.animal = animal;
    }
}
```

new가 없다

Spring Container = Bean Container

Step 1: Create the AnimalType object (Dog, Cat)

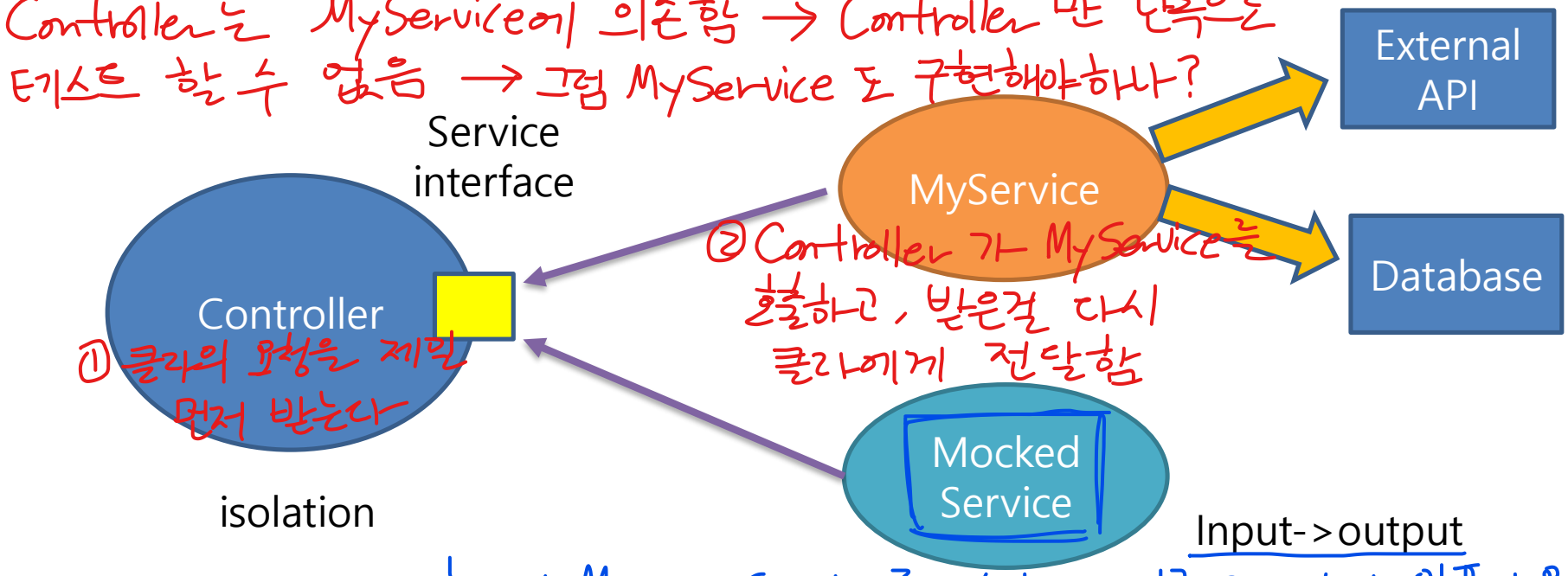
```
public class Dog
        implements AnimalType{
    //…
}
```

```
public class Cat
        implements AnimalType{
    //…
}
```

의존성 주입은 밥먹듯이, 다양하게 쓰인다.

spring

Dependency Injection



Controller Object depends on MyService Object
to make external Service call and Database call.

To write unit test for Controller method,
we need to stub or mock the depended class object(MyService) behavior
by using any mock framework.

spring

Dependency Injection

의존성 주입

- It is design pattern in which an object's dependency is injected by the *framework* rather than by the object itself

- It reduces coupling between multiple objects as it is dynamically injected by the framework

객체의 의존성이 외부에 있는 프레임워크(컨테이너)에 의해서 주입된다

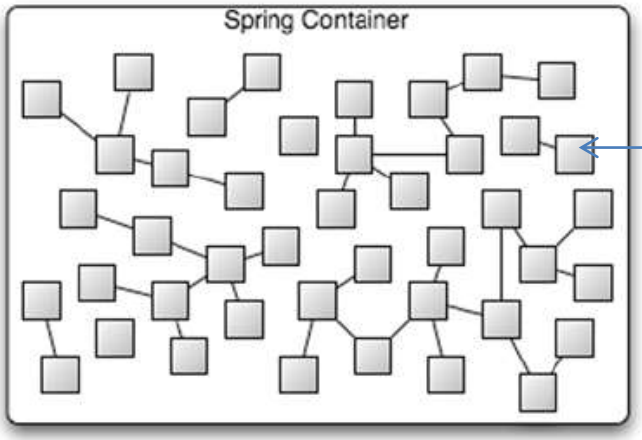
이러한 설계 패턴을 Dependency Injection 이라고 한다

(동적으로 주입되며, 컴파일이 아닌 런타임 시 생성자에 주입됨)

spring

2. The Spring (IoC) Container

- Core component of the Spring framework
- Primary functions
  - Create and manage objects
  - Inject object's dependencies



bean = 객체 in Spring

Figure 2.1
In a Spring application, objects are created, wired together, and live within the Spring container.

spring

The Spring Container

컨테이너는 사실, 알아서 동작하는 게 아니고 설정이 필요함

- The configuration metadata can be represented either by 설정 방법은 3가지가 있다
  - XML
  - Java annotations
  - Java-based Configuration (클래스)

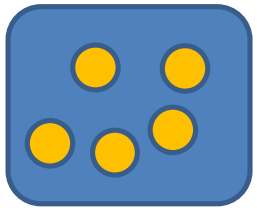
spring

The Spring Container

컨테이너는 2가지 유형이 있다

- Spring comes with two types of containers
  - BeanFactory
  - ApplicationContext

The Spring Container



BeanFactory	ApplicationContext
The simplest factory, mainly for DI	The advanced and more complex factory
Used when resources are limited, e.g., mobile, applets etc.	Used elsewhere and has the below features 1) Enterprise aware functions 2) Publish application events to listeners 3) Wire and dispose beans on request
org.springframework.beans.factory.BeanFactory	org.springframework.context.ApplicationContext
XMLBeanFactory : most commonly used implementation	*ClassPathXmlApplicationContext:* ← 우리가 쓸 것 most commonly used implementation

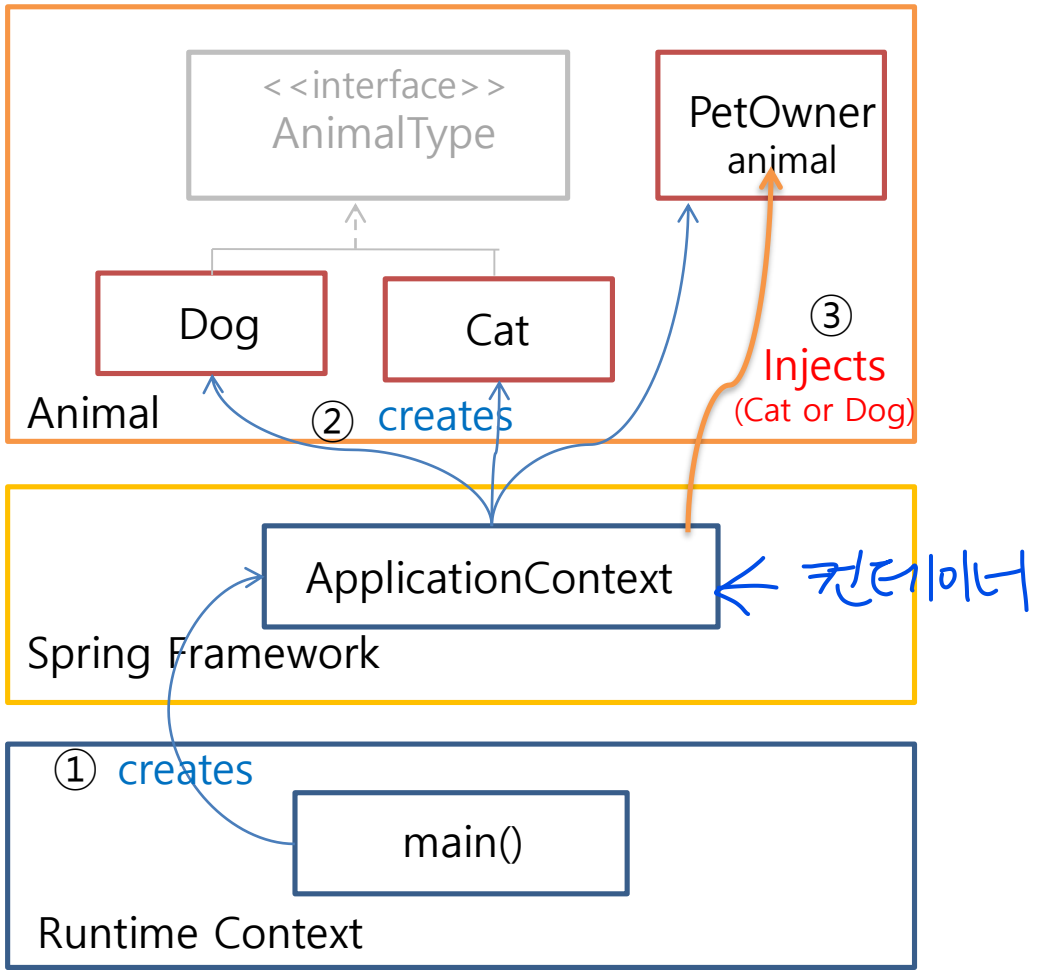
```
ApplicationContext context =
        new ClassPathXmlApplicationContext("kr/ac/hansung/spring/beans/bean.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
```

아이디를 부여하여 bean에 접근 가능

spring

전체적인 코드 구조

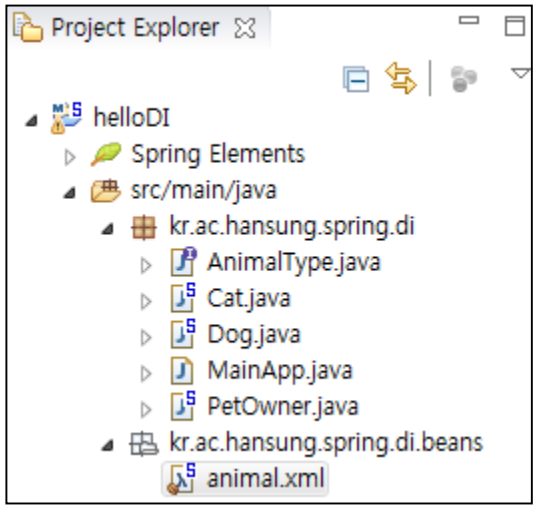
3. Spring Container and DI



spring

예시 코드

Example for DI



Cat.java

```java
package kr.ac.hansung.spring;

public class Cat implements AnimalType{

    String myName;

    public void setMyName(String name) {
        this.myName = name;
    }

    public void sound() {
        System.out.println("Cat name = " + myName + ": " + "Meow!");
    }

}
```

AnimalType.java

```java
package kr.ac.hansung.spring;

public interface AnimalType {

    public void sound();

}
```

Dog.java

```java
package kr.ac.hansung.spring;

public class Dog implements AnimalType {

    String myName;

    public void setMyName(String name) {
        this.myName = name;
    }

    public void sound() {
        System.out.println("Dog name = " + myName + ": " + "Bow Wow");
    }

}
```

Configure beans and dependencies

PetOwner.java

```java
package kr.ac.hansung.spring;

public class PetOwner {

    public AnimalType animal;

    public PetOwner(AnimalType animal) {
        this.animal = animal;
    }

    public void  play() {
        animal.sound();
    }
}
```

Dog 일지 Cat 일지는 바뀔 수 있음

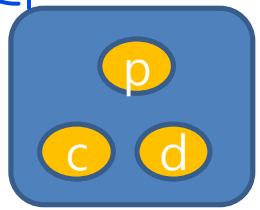
Bean Definition

Dependency injection

animal.xml

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

    <bean id="dog" class="kr.ac.hansung.spring.di.Dog">
        <property name="myName" value="poodle"></property>
    </bean>

    <bean id="cat" class="kr.ac.hansung.spring.di.Cat">
        <property name="myName" value="bella"></property>
    </bean>

    <bean id="petOwner" class="kr.ac.hansung.spring.di.PetOwner">
        <constructor-arg ref="cat"></constructor-arg>
    </bean>

</beans>
```

Dog로 바꾸더라도 java 소스에는 영향이 없음!!!



MainApp.java

```java
package kr.ac.hansung.spring.di;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

    public static void main(String[] args) {

        ClassPathXmlApplicationContext context =
                new ClassPathXmlApplicationContext("kr/ac/hansung/spring/di/beans/animal.xml");

        PetOwner person = (PetOwner) context.getBean("petOwner");
        person.play();

        context.close();

    }

}
```

Advantages of DI

의존성 주입의 4가지 장점

- Reduced Dependencies  의존성 감소
  - When a components dependencies are reduced, it is less vulnerable to changes

- More Reusable Code 재사용성 ↑
  - If a different implementation of some interface is needed in a different context, the component can be configured to work with that implementation. There is no need to change the code

spring

Advantages of DI

- More Testable Code 테스트 ↑
  - When dependencies can be injected into a component, it is possible to inject mock implementations of these dependencies. Mock objects are used for testing as replacement for a real implementation
  - For instance, handling both when the mock returns a correct object, when null is returned and when an exception is thrown

- More Readable Code 가독성 ↑
  - This makes it easier to see what dependencies a component has, making the code more readable

spring

Bean = POJO = 상속, 인터페이스 없는 객체

4. What are Beans?

- In Spring, POJO's (plain old java object) are called 'beans' and those objects instantiated and managed by Spring container
- Beans are created with the configuration metadata (XML file) that we supply to the container
  - With this metadata, container knows how to create bean, beans lifecycle, beans dependencies
- Instances of those objects will be reached by getBean() method

spring

Spring Bean Definition

- Defines a bean for Spring to manage
  - Key attributes
    - class (required): fully qualified java class name
    - id: the unique identifier for this bean
    - scope: the scope of the objects(singleton, prototype)
    - constructor-arg: arguments to pass to the constructor at creation time
    - property: arguments to pass to the setters at creation time
    - init, destroy method

클래스의 객체가 1개
1개 아님
생성자 인자
setter 인자
bean 이
초기화되거나 삭제될 때
실행하는 함수 지정 가능

spring

Spring Bean Definition

XML based configuration file

```
<!-- A simple bean definition -->
<bean id="..." class="...">
</bean>

<!-- A bean definition with scope-->
<bean id="..." class="..." scope="singleton">
</bean>

<!-- A bean definition with property -->
<bean id="..." class="...">
        <property name="message" value="Hello World!"/>
</bean>

<!-- A bean definition with initialization method -->
<bean id="..." class="..." init-method="..." >
</bean>
```

spring

5. Spring Bean Scope

Bean Scope	Description
singleton	Single instance of bean in every getBean() call [Default]
prototype	New instance of bean in every getBean() call
request	Single instance of bean per HTTP request
session	Single instance of bean per HTTP session
global-session	Single instance of bean per global HTTP session

클래스의 인스턴스 1개

새로운 인스턴스 만들어짐

Valid in the context of **web-aware** ApplicationContext

Spring Bean Scope

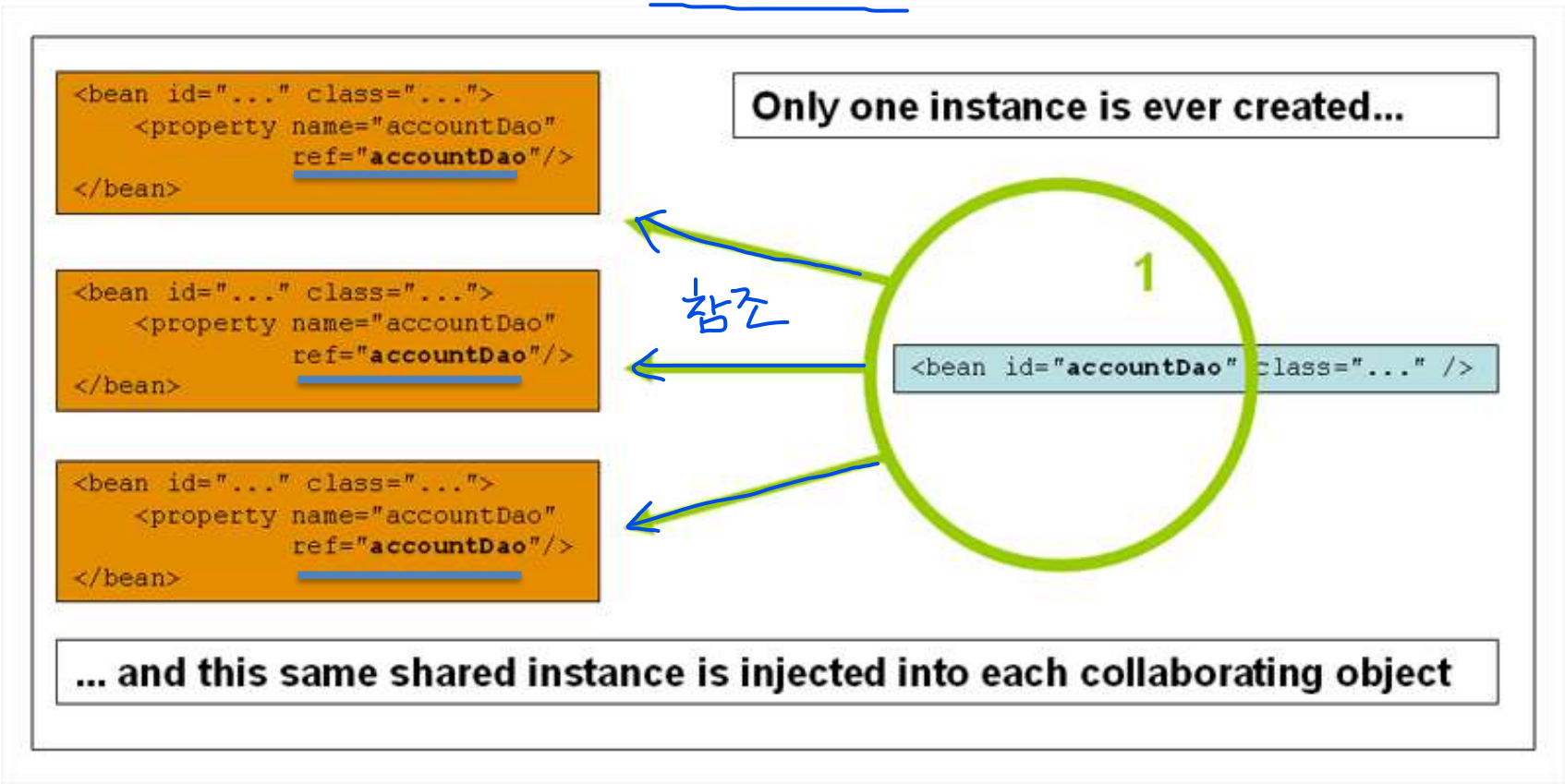
- Bean can be of two types
  1) Singleton (e.g., <bean scope="singleton" … />)
  2) Prototype (e.g., <bean scope="prototype" … />)

단 한 번 만들어지고 항상 동일한 bean이 참조됨, 컨테이너 셧다운 시
- A 'singleton' bean is created once in the Spring container. 삭제됨
  This 'singleton' bean is given every time when referred. It is garbage collected when the container shuts down
- A 'prototype' bean means a new object in every request. It is garbage collected in the normal way, i.e., when there is no reference for this object
- By default, every bean is singleton if not specified explicitly

참조할 때마다 새로운 객체 생성, 더 이상 참조가 없을 시 삭제됨

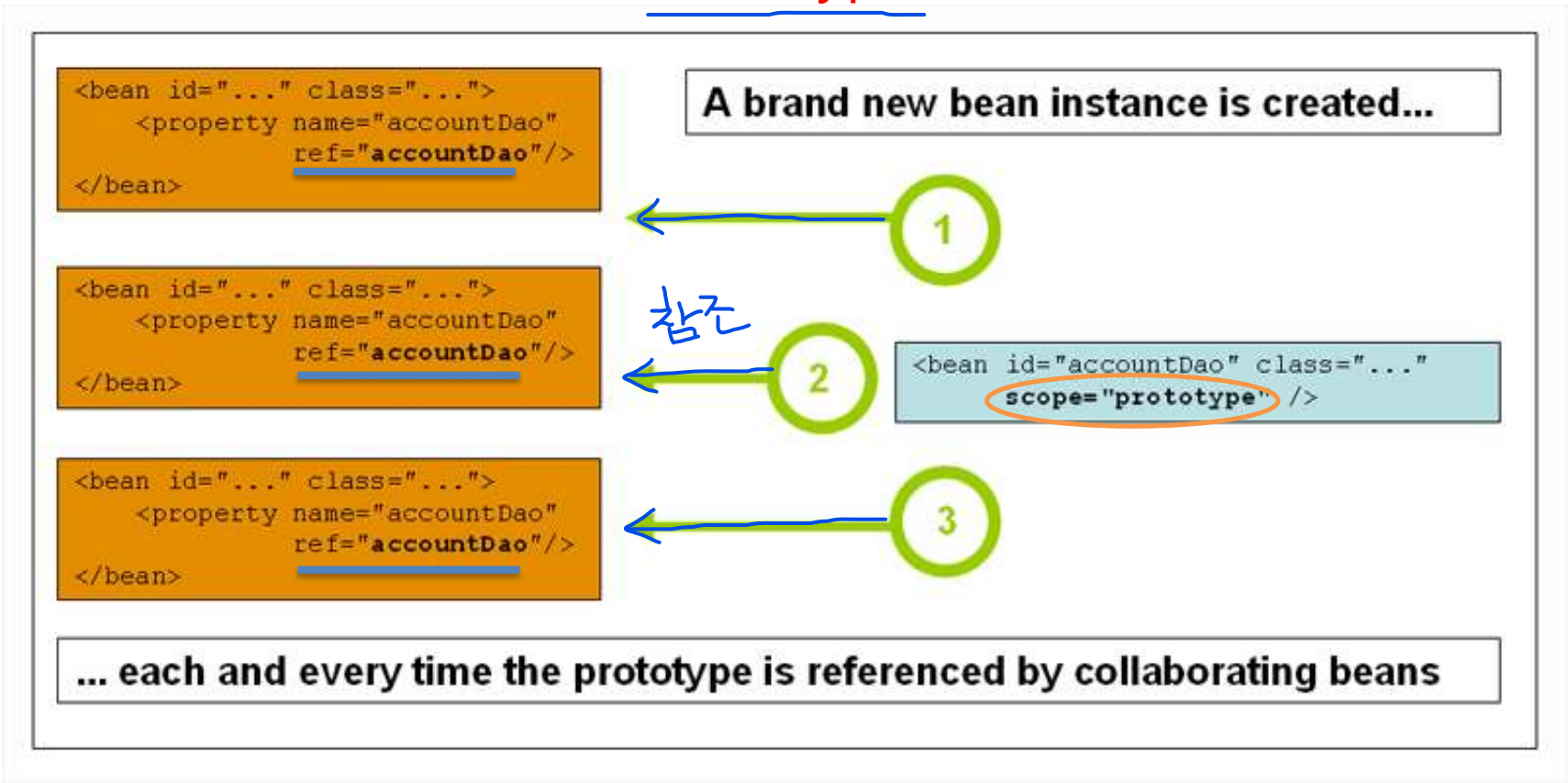
Spring Bean Scope

Singleton



Spring Bean Scope

Prototype



spring

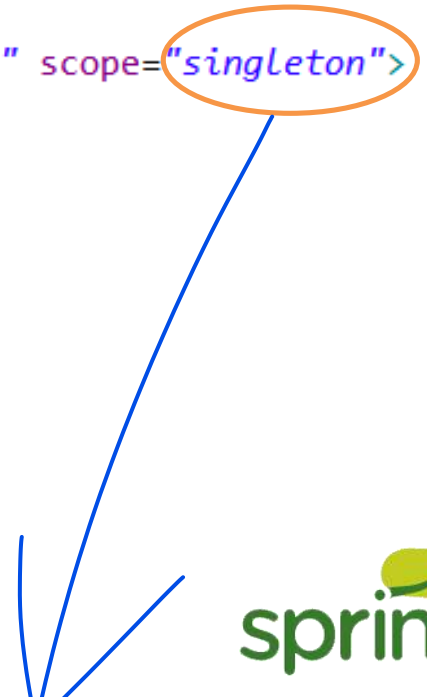
Example

PetOwner.java

```java
package kr.ac.hansung.spring;

public class PetOwner {
    String userName;

    public String getUserName() {
        System.out.println("Person name is " + userName);
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public AnimalType animal;

    public PetOwner(AnimalType animal) {
        this.animal = animal;
    }

    public void  play() {
        animal.sound();
    }
}
```

Example

```
<bean id="dog" class="kr.ac.hansung.spring.Dog">
    <property name="myName" value="poodle" />
</bean>

<bean id="cat" class="kr.ac.hansung.spring.Cat">
    <property name="myName" value="bella" />
</bean>

<bean id="petOwner" class="kr.ac.hansung.spring.PetOwner" scope="singleton">
    <constructor-arg name="animal" ref ="dog" />
</bean>
```



Example

MainApp.java

```java
package kr.ac.hansung.spring;

import org.springframework.context.ApplicationContext;

public class MainApp {

    public static void main(String[] args) {

        ApplicationContext context =
            new ClassPathXmlApplicationContext("kr/ac/hansung/spring/beans/bean.xml");

        PetOwner person1 = (PetOwner) context.getBean("petOwner");
        person1.setUserName("Alice");
        person1.getUserName();

        PetOwner person2 = (PetOwner) context.getBean("petOwner");
        person2.getUserName();

        ((ClassPathXmlApplicationContext) context).close();
    }
}
```

getUserName 했을 때 →

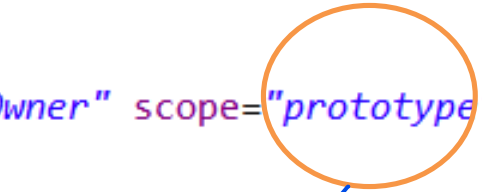
```
Person name is Alice
Person name is Alice
```

결과가 같다

spring

Example

```
<bean id="dog" class="kr.ac.hansung.spring.Dog">
    <property name="myName" value="poodle" />
</bean>

<bean id="cat" class="kr.ac.hansung.spring.Cat">
    <property name="myName" value="bella" />
</bean>

<bean id="petOwner" class="kr.ac.hansung.spring.PetOwner" scope="prototype">
    <constructor-arg name="animal" ref ="dog" />
</bean>
```



Example

MainApp.java

```java
package kr.ac.hansung.spring;

import org.springframework.context.ApplicationContext;

public class MainApp {

    public static void main(String[] args) {

        ApplicationContext context =
            new ClassPathXmlApplicationContext("kr/ac/hansung/spring/beans/bean.xml");

        PetOwner person1 = (PetOwner) context.getBean("petOwner");
        person1.setUserName("Alice");
        person1.getUserName();

        PetOwner person2 = (PetOwner) context.getBean("petOwner");
        person2.getUserName();

        ((ClassPathXmlApplicationContext) context).close();
    }
}
```

결과 다음 →
Person name is Alice
Person name is null

별도의 객체라서,

spring

6. Dependency Injection Methods

의존성 주입 방법 2가지

- Constructor-based Injection 생성자를 통한 방법
  - Pass dependencies in via constructor

- Setter-based Injection Setter를 통한 방법
  - Pass dependencies in via property setters

spring

생성자를 통한 방법

1) Constructor-based Injection

```
<bean id="petOwner" class="kr.ac.hansung.spring.PetOwner" >
    <constructor-arg    ref ="dog" />
</bean>
```

```
public class PetOwner {

    public AnimalType animal;

    public PetOwner(AnimalType animal) {
        this.animal = animal;
    }
    …
}
```

spring

Setter를 통한 방법

2) Setter-based Injection

```
<bean id="petOwner" class="kr.ac.hansung.spring.PetOwner">
    <property name="animal" ref="dog" />
</bean>
```

Setter method is called

```
public class PetOwner {

    public AnimalType animal;

    public void setAnimal(AnimalType animal) {
        this.animal = animal;
    }

    …
}
```

spring

XML 이 아니라 주석 형태를 이용한 설정

Spring Annotation Based Configuration

1. Spring Annotation

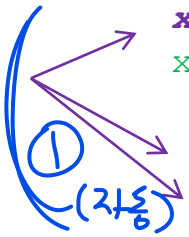
클래스에 직접 설정하지만 XML과 다른게 없다

- Developed and became famous from Spring 2.5
- "Old wine in new bottle" - an alternative of the xml configuration for bean wiring
  - you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration
- XML configuration can be verbose 장황한 XML을 간단하게
  - Annotations minimizes the XML configuration
- Not enabled by default (need explicit enabling) 주석쓸거다고 선언필요
- XML overrides annotation for bean configuration
- IDE support

주석 쓸것다고 선언

2. Enabling Spring Annotation

```
<?xml version="1.0" encoding="UTF-8" ?>

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

    <context:annotation-config/>

    <!--
       <bean ... />
       <bean ... />
           .
           .
           .
       <bean ... />
    -->
</beans>
```



네임스페이스 ②

spring

3.1 @Required

- **Applicable to only setter methods**

```
public class Boy {
 private String name;
 private int age;

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

 @Required
 public void setAge(int age){
  this.age = age;
 }

 //   getters ...
}
```

the bean property must be populated in XML configuration file

the bean property must be populated in XML configuration file

- **Values by name**

인자

```
<bean id="boy" class="Boy">
<property name="name" value="Rony"/>
<property name="age" value="10"/>
</bean>
```

속성

XML 파일에 property 있는지 검사

- **Strict checking**

```
<bean id="boy" class="Boy">
 <property name="name" value="Rony"/>
</bean>
```

```
Property 'age' is required for bean
'boy'
```

오류 메시지

spring

3.2 @Autowired

```
public class Boy {
 private String name;
 private int age;

 //   getters and setters ...
}

public class College {


 private Boy student;

 public void setStudent(Boy aboy){
  this.stuendt = aboy;
 }


}
```

```
<bean id="boy" class="Boy">
 <property name="name" value="Rony"/>
 <property name="age" value="10"/>
</bean>


<bean id="college" class="College">
 <property name="student" ref="boy"/>
</bean>
```

기존 방식?

spring

@Autowired

- **Applicable to methods, fields and constructors**

```
public class Boy {
 private String name;
 private int age;

 //   getters and setters ...
}

public class College {

 @Autowired
 private Boy student;

 public void setStudent(Boy aboy){
   this.stuendt = aboy;
 }


}
```

No setter needed

- **Wiring by type**

```
<bean id="boy" class="Boy">
 <property name="name" value="Rony"/>
 <property name="age" value="10"/>
</bean>
```

Boy 라는 타입을 가진 bean을 주입하라

```
<bean id="college" class="College">
  <property name="student" ref="boy"/>
</bean>
```

No property needed

spring

3.3 @Qualifier

Boy가 2개면 주입하지 않는다

- **Solution to @Autowired for type ambiguity**

```
public class Boy {
 private String name;
 private int age;

 //   getters and setters ...
}

public class College {

@Autowired
@Qualifier(value="tony")
 private Boy student;

  //   getters ...
}
```

지정해주면 해결

null

Qualifier value

Two beans for "Boy" type

```
<bean id="boy1" class="Boy">
 <qualifier value="rony"/>
 <property name="name" value="Rony"/>
 <property name="age" value="10"/>
</bean>

<bean id="boy2" class="Boy">
 <qualifier value="tony"/>
 <property name="name" value="Tony"/>
 <property name="age" value="8"/>
</bean>

<bean id="college" class="College">
</bean>
```

spring

3.4 @Resource

auto wiring 안에 name으로

- @Resource(name="<beanName>") is used for auto wiring by name
- @Resource can be applied in field, argument and methods

통일한 기능 수행

- Note: @Autowired and @Resource work equally well
  (@Resource: auto-wire by name, @Autowired: auto-wire by type)

```
<bean id="boy1" class="Boy">
 <property name="name" value="Rony"/>
 <property name="age" value="10"/>
</bean>

<bean id="boy2" class="Boy">
 <property name="name" value="Tony"/>
 <property name="age" value="8"/>
</bean>

<bean id="college" class="College">
</bean>
```

```
public class College {

  @Resource(name="boy1")
  private Boy student;

  //   getters and setters ...
}
```

spring

결론

Conclusion

- POJO's (plain old java object) are called 'beans' and those objects instantiated, managed, created by Spring container-> Inversion of Control

- The Spring container enforces the dependency injection pattern for your components

- DI helps in gluing loosely coupled classes together

클래스 사이의 연결을 약하게

spring