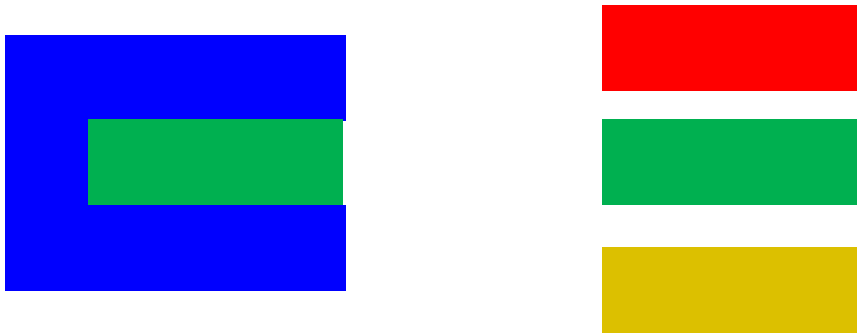
2020년도 2학기

스프링프레임워크

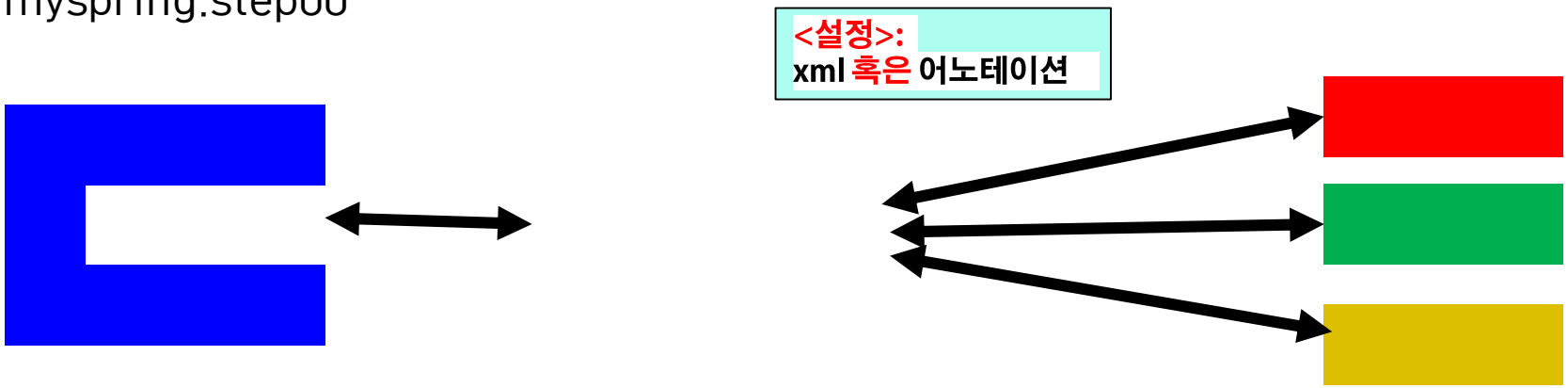
[교재 Class04 의존성 주입] p65~

스프링 실습 프로젝트 정리 실습

- SpringProject1 : (5주차) //개발자에 의한 객체 결합
  - com.myspring.step1 : 객체간의 결합도가 높은 예제
  - com.myspring.step2 : 다형성을 이용해 결합도를 낮춤
  - com.myspring.step3 : Factory디자인패턴을 이용해 결합도를 낮춤
  - com.myspring.step4 : 별도의 텍스트파일을 이용해 소스를 다시 컴파일하지않도록 수정



- SpringProject2 : (6주차) //스프링컨테이너에 의한 객체 결합
  - com.myspring.step00



IoC(Inversion of Control) : 제어권의 역전

IoC(Inversion of Control)란 "제어의 역전"이라는 뜻으로 프로그램의 흐름을 개발자가 아닌 프레임워크가 주도하게 된다는 디자인패턴으로서,
객체의 생성에서 소멸까지 프레임워크가 관리를 하면서 DI(Dependency Injection)나 AOP(Aspect Oriented Programming)가 가능

- DI가 아닌 경우 : 개발자가 new로 객체생성을 함.

- 스프링 컨테이너가 객체 생성하여 개발자 코드에 주입

- DI가 아닌 경우 : 개발자가 new로 객체생성을 함.

```
public static void main(String[] args) {

    MyCalculator myCalculator = new MyCalculator();
    myCalculator.setCalculator(new Calculator());

    myCalculator.setFirstNum(10);
    myCalculator.setSecondNum(2);

    myCalculator.add();
    myCalculator.sub();
    myCalculator.mul();
    myCalculator.div();

}
```

- 스프링 컨테이너가 객체 생성하여 개발자 코드에 주입

```
public static void main(String[] args) {

    String configLocation = "classpath:applicationCTX.xml";
    AbstractApplicationContext ctx = new GenericXmlApplicationContext(configLocation);
    MyCalculator myCalculator = ctx.getBean("myCalculator", MyCalculator.class);

    myCalculator.add();
    myCalculator.sub();
    myCalculator.mul();
    myCalculator.div();

}
```

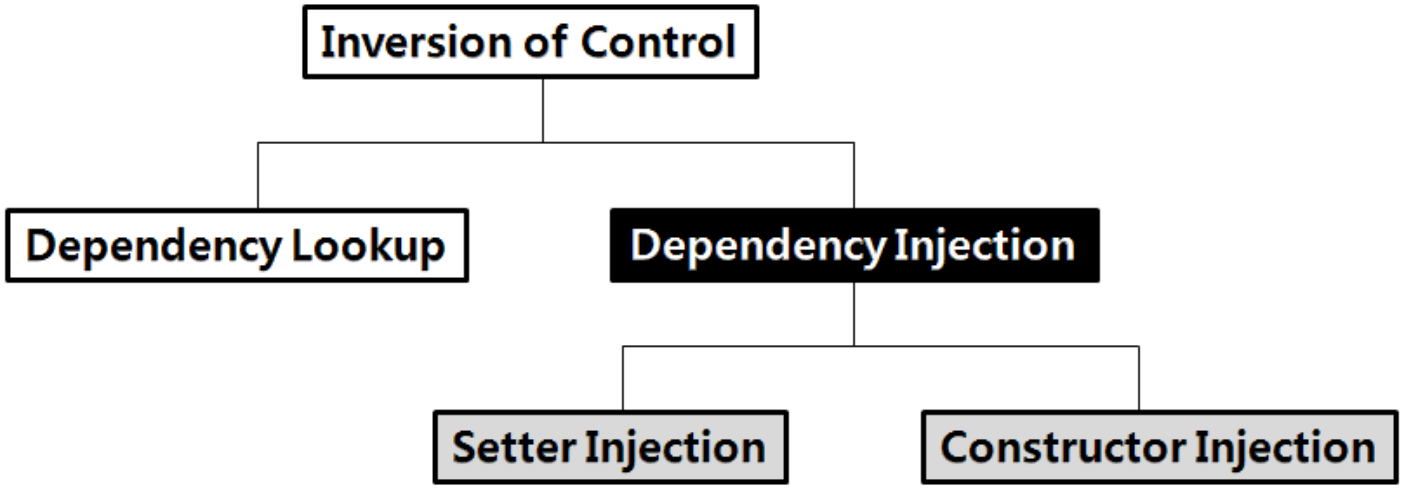
DI(Dependency Injection) vs. DL(Dependency Lookup)

- DL
  - 저장소에 저장되어 있는 bean에 접근하기 위해 컨테이너가 제공하는 API를 이용하여 bean을 lookup 하는 방식

```
ApplicationContext factory = new ClassPathXmlApplicationContext("~~~");
factory.getBean("b");
```

  - Container 밖에서 실행 할 수 없음
  - 테스트하기 매우 어려우며, 코드양이 증가함
  - String 타입이 아니므로 Object 타입을 받아서 매번 Casting해야함

실제 애플리케이션 개발과정에서 사용X



DI(Dependency Injection) vs. DL(Dependency Lookup)

- DI : 컨테이너가 직접 의존 관계를 bean 설정 정보를 바탕으로 자동으로 연결해 주는 것 .

```
<bean id="a" class="~~~~~"/>
<bean id="b" cass="~~~~~">
 <construct-arg>
  <ref bean="a"/>
 </contruct-arg>
</bean>
```

  - 컨테이너가 흐름의 주체가 되어 application코드에 의존관계를 주입하면 실행시에 동적으로 의존관계가 생성.
  - 클래스의 상속이 필요 없음
  - 개발자들은 bean 설정파일에서 의존관계가 필요하다는 것을 추가하면 된다.

DI 구현 방법

- **setter injection**
  - JavaBeans의 property 구조를 이용한 방식, SpringFramework가 주로 이용하는 방법
  - 오브젝트가 Container에 의해서 만들어지고 나서 모든 Dependency들이 Setter 메소드를 통해서 주입됨
  - 장점
    - JavaBeans property 구조를 사용하기 때문에 IDE등에서 개발하기 편리함
    - 상속시 구조가 그대로 전달됨
    - Getter 메소드를 통해서 현재 오브젝트의 상태 정보를 얻어올 수 있음
  - 단점
    - Setting순서를 지정 할 수 없음
    - 모든 필요한 property가 세팅되는 것에 대해서 보장 할 수 없음
- **constructor injection**
  - 클래스의 생성자를 이용하는 방법
  - 필요한 의존성을 포함하는 클래스의 생성자를 만들고 이를 통해 의존성을 주입한다.
  - 생성자의 매개변수를 이용하여 여러개의 인자를 주입할 수 있다.

스프링 실습 프로젝트 정리 실습

1. **프로그래머에 의한 인젝션 :** SpringProject2 프로젝트-com.myspring.step01 패키지

2. **생성자 인젝션 :** SpringProject2 프로젝트-com.myspring.step02 패키지
   - 기본 생성자 말고 매개변수를 가지는 다른 생성자 호출하도록 설정
   - 다중 변수 매핑 가능 (다음 9번 슬라이드 참조)

3. **생성자 인젝션 의존관계 변경 :** SpringProject2 프로젝트-com.myspring.step03 패키지

4. **Setter 인젝션 :** SpringProject2 프로젝트-com.myspring.step04 패키지
   - <property>엘리먼트 사용하며, name속성값이 호출 메소드 이름
     예) <property name="speaker" ..> ▶ setSpeaker( )
     <property name="boardDAO" ..> ▶ setBoardDAO( )
   - 생성자 인젝션과 마찬가지로 Setter메소드 호출시 객체를 인자로 넘기려면 ref속성, 기본형 데이터를 넘기려면 value 속성 이용

5. **p 네임스페이스 사용 :** applicationContext.xml 만 작업
   - p네임스페이스 이용하여 좀 더 효율적으로 DI 처리 (다음 10번 슬라이드 참조)

6. **컬렉션 객체 설정 :**
   - List 타입 매핑 (**SpringProject2** 프로젝트-com.myspring.step05 패키지) : 배열이나 List타입
   - Set 타입 매핑 (**SpringProject2** 프로젝트-com.myspring.step06 패키지) : 중복되지않는 집합 객체
   - Map 타입 매핑 (**SpringProject2** 프로젝트-com.myspring.step07 패키지) : 특정 key로 데이터 등록
   - Properties 타입 매핑 (**SpringProject2** 프로젝트-com.myspring.step08 패키지) : key=value 형태

DI 구현 방법 : 생성자 인젝션 - 다중 변수 매핑 가능

```
public class SamsungTV implements TV {
	private SonySpeaker speaker;
	private int price;

	public SamsungTV() {
		System.out.println("===> SamsungTV(1) 객체 생성");
	}

	public SamsungTV(SonySpeaker speaker) {
		System.out.println("===> SamsungTV(2) 객체 생성");
		this.speaker = speaker;
	}

	public SamsungTV(SonySpeaker speaker, int price) {
		System.out.println("===> SamsungTV(3) 객체 생성");
		this.speaker = speaker;
		this.price = price;
	}

	public void powerOn() {
		System.out.println("SamsungTV---전원 켠다. (가격 : " + price + ")");
	}

	public void powerOff() {
		System.out.println("SamsungTV---전원 끈다.");
	}

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

	public void volumeDown() {
		speaker.volumeDown();
	}
}
```

```
<?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:p="http://www.springframework.org/schema/p"
		xmlns:context="http://www.springframework.org/schema/context"
		xsi:schemaLocation="http://www.springframework.org/sc
http://www.springframework.org/schema/beans/spring-beans.xsd
		http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.2.xsd

		<bean id="tv" class="polymorphism.SamsungTV">
			<constructor-arg  ref="sony"></construc
			<constructor-arg  value="2700000"></cc
		</bean>

		<bean id="sony" class="polymorphism.SonySpeaker"/>
</beans>
```

생성자 오버로딩 되어있을시, index 속성 사용 가능

예) <bean id="tv" class="....SamsungTV">
<constructor-arg index="0" ref="sony"></construct
<constructor-arg index="1" value="2700000"></constructor-arg>
</bean>

4. 생성자 아규먼트 매칭

스프링은 생성자가 두 개이상의
면 다음과 같다.

```
<bean id="billingService"
   class="com.lizjason.spri
   <constructor-arg index=
   <constructor-arg index=
</bean>
```

이것은 다음과 같이 **type** 속성을

```
<bean id="billingService"
   class="com.lizjason.spri
   <constructor-arg type=
      value="lizjason"/>
   <constructor-arg type=
</bean>
```

index를 사용하는 것이 어느 정도
제가 있을 경우에만 **index**를 사용

DI 구현 방법 : p 네임스페이스 사용

p:변수명-ref="참조할 객체의 이름이나 아이디"
p:변수명="설정할 값 "

예)

```
<?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:p="http://www.springframework.org/schema/p"
		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-
4.2.xsd">

		<bean id="tv" class="polymorphism.SamsungTV" p:speaker-ref="sony" p:price="2700000"></bean>

		<bean id="sony" class="polymorphism.SonySpeaker"/>
		<bean id="apple" class="polymorphism.AppleSpeaker"/>
</beans>
```

DI 어노테이션 기반 설정

- Context 네임스페이스 추가
  - [Namespaces]탭에서 'context'항목만 체크

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    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-4.2.xsd">

        <context:component-scan base-package="com.springbook.biz"/>

</beans>
```

com.springbook.biz 패키지로 시작하는 모든 클래스를 스캔
예) com. com.springbook.biz
com.springbook.biz.example1
com.springbook.user.user1

DI 어노테이션 기반 설정 - @Component 설정

@Component 설정

xml 방식일 경우의 id

```
<bean id="tv" class="polymorphism.LgTV"></bean>
```

```
@Component("tv")
public class LgTV implements TV {
	public LgTV() {
		System.out.println("===> LgTV 객
	}
}
```

"SpringProject3" 프로젝트 com.myspring.step01

DI 어노테이션 기반 설정 - Dependency Injection 설정

어노테이션	설명
@Autowired	주로 변수 위에 설정하여 해당 타입의 객체를 찾아서 자동으로 할당 org.springframework.beans.factory.annotation.Autowired
@Qualifier	특정 객체의 이름을 이용하여 의존성 주입할 때 사용 org.springframework.beans.factory.annotation.Qualifier
@Inject	@Autowired와 동일한 기능을 제공 javax.annotation.Resource
@Resource	@Autowired와 @Qualifier의 기능을 결합한 어노테이션 javax.inject.Inject

DI 어노테이션 기반 설정 - @Autowired 설정

@Autowired 설정

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

	public LgTV() {
		System.out.println("===> LgTV 객          생성");
	}
	public void volumeUp() {
		speaker.volumeUp();
	}
	public void volumeDown() {
		speaker.volumeDown();
	}
}
```

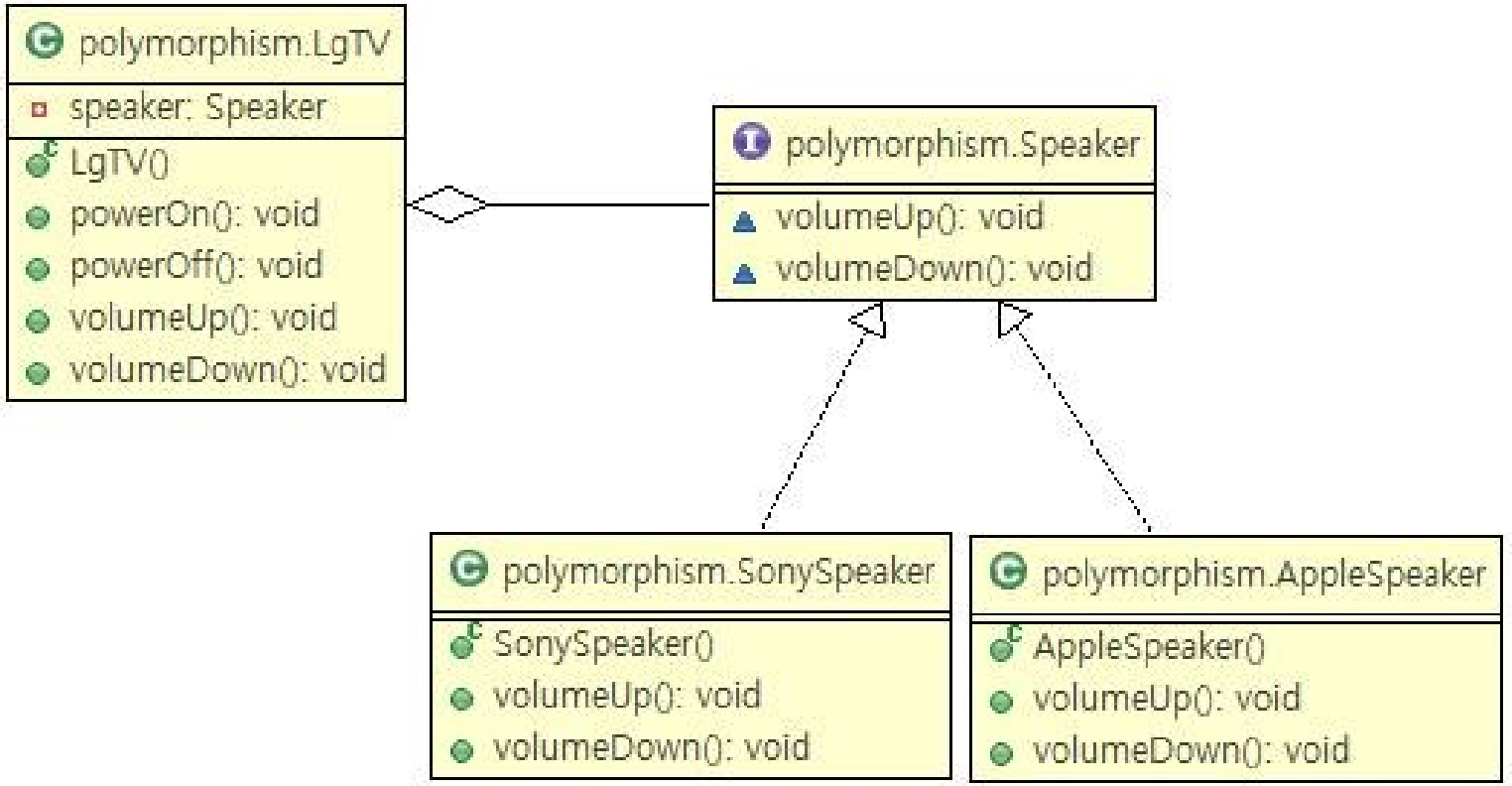
Speaker 타입의 객체를
메모리에서 찾아 할당한다.

"SpringProject3" 프로젝트 com.myspring.step02

"SpringProject3" 프로젝트 com.myspring.step02_1

DI 어노테이션 기반 설정 - @Qualifier 설정

@Qualifier 설정



"SpringProject3" 프로젝트 com.myspring.step03

DI 어노테이션 기반 설정 - @Qualifier 설정

@Qualifier 설정

- 의존성 주입 대상 객체가 두 개 이상일 때 에러 발생.

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

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

Speaker 타입의 객체 중
아이디가 "apple"인 객체 할당

DI 어노테이션 기반 설정 - @Resource 설정

@Resource 설정

- @Resource는 객체의 이름을 이용하여 의존성 주입을 처리

```
@Component("tv")
public class LgTV implements TV {
	@Resource(name="apple")
	private Speaker speaker;

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

Speaker 타입의 객체 중
아이디가 "apple"인 객체 할당

"SpringProject3" 프로젝트 com.myspring.step03

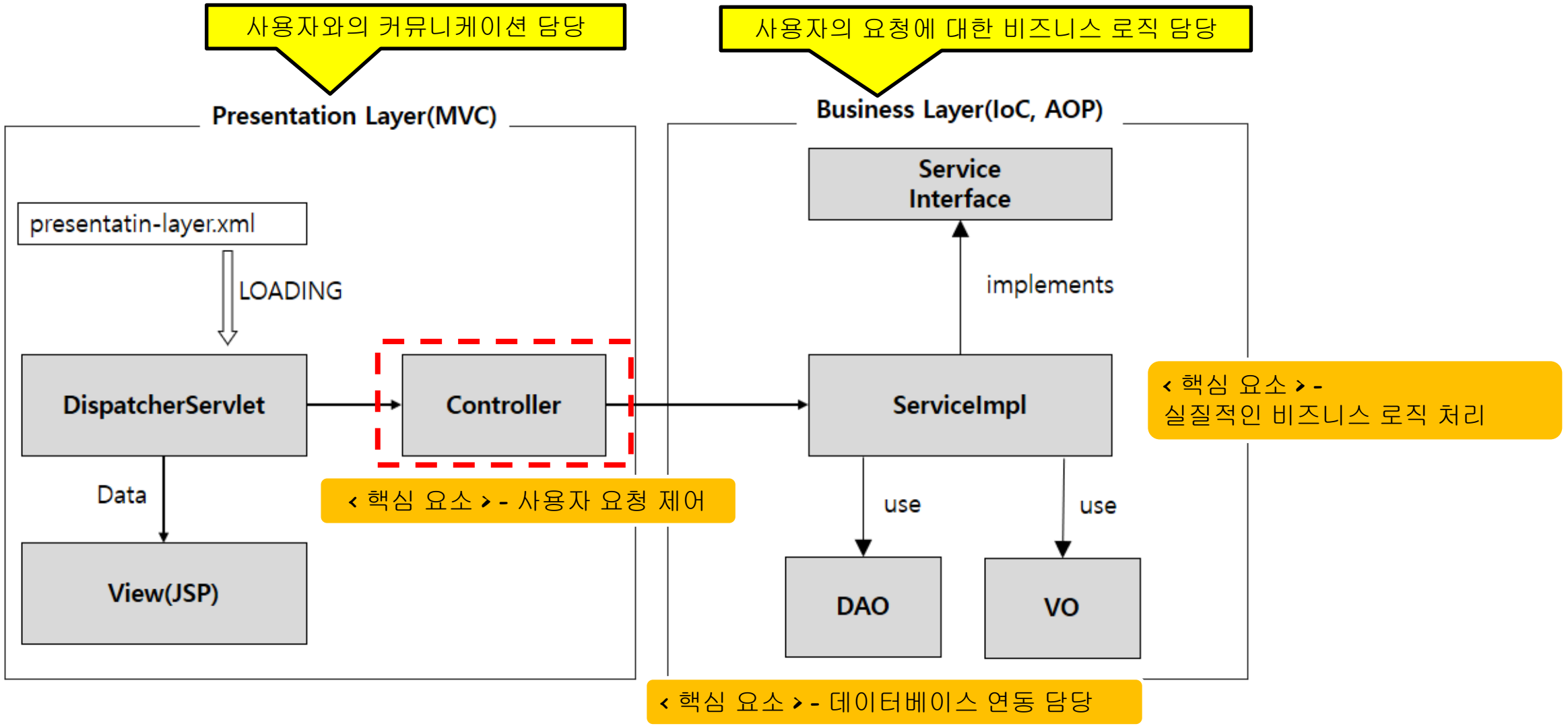
Annotation VS. <bean> 등록

- 유지 보수 과정에서 자주 변경되는 객체는 <bean> 등록으로 처리한다.
- 유지 보수 과정에서 자주 변경되지 않는 객체는 Annotation으로 처리한다.
- 의존성 주입은 Annotation으로 처리한다.

xml설정과 어노테이션 설정의 혼합 :
"SpringProject3" 프로젝트 com.myspring.step04
"SpringProject3" 프로젝트 com.myspring.step04_1

설정파일을 포함해서 모든 것을 어노테이션 방식으로 작업 :
"SpringProject3" 프로젝트 com.myspring.step05

Layered Architecture



Layer 별 Annotation

@Component를 상속하여 수행 역할별 분류

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