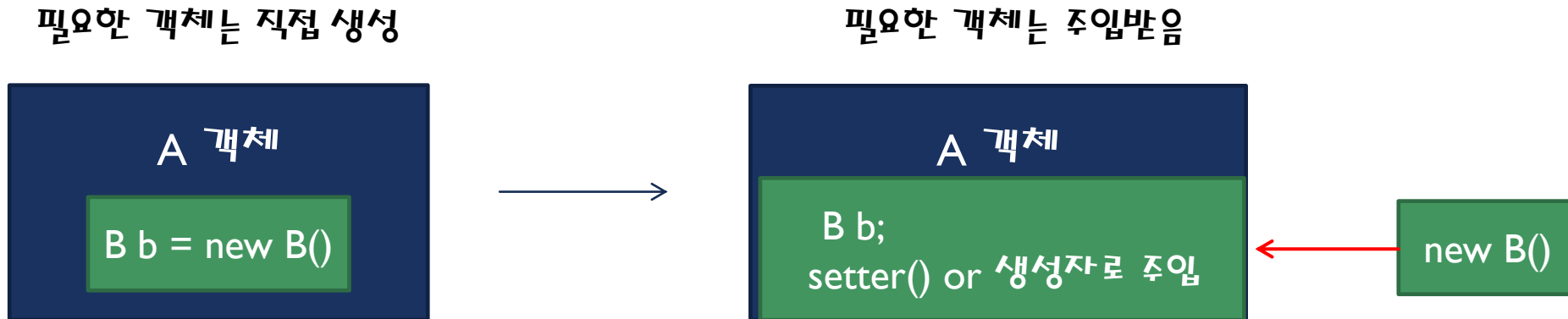


2. IoC

1. IoC란
2. DI(Dependency Injection)
3. 애노테이션 기반 IoC 설정
4. Junit을 사용한 테스트

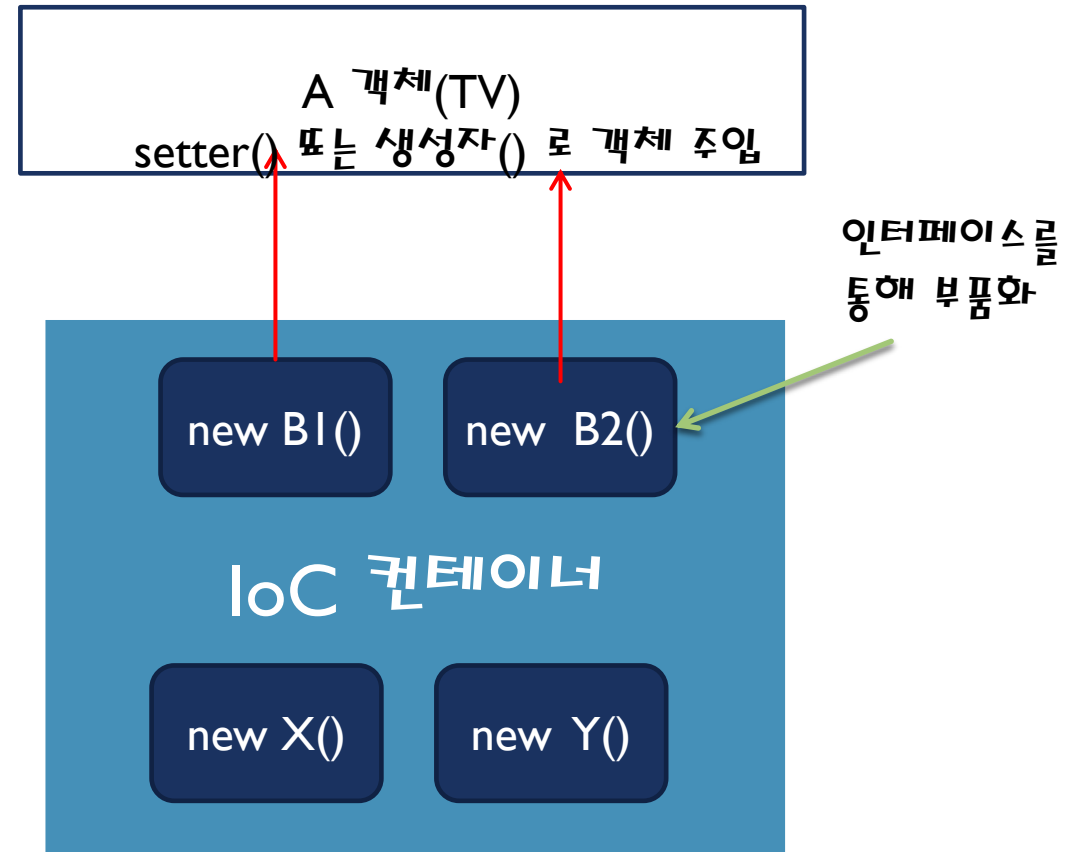
I.I IoC란

- Inversion of Control
- 제어의 역전이란 객체의 생성, 관리에 대한 제어권이 바뀜



1.2 IoC 컨테이너

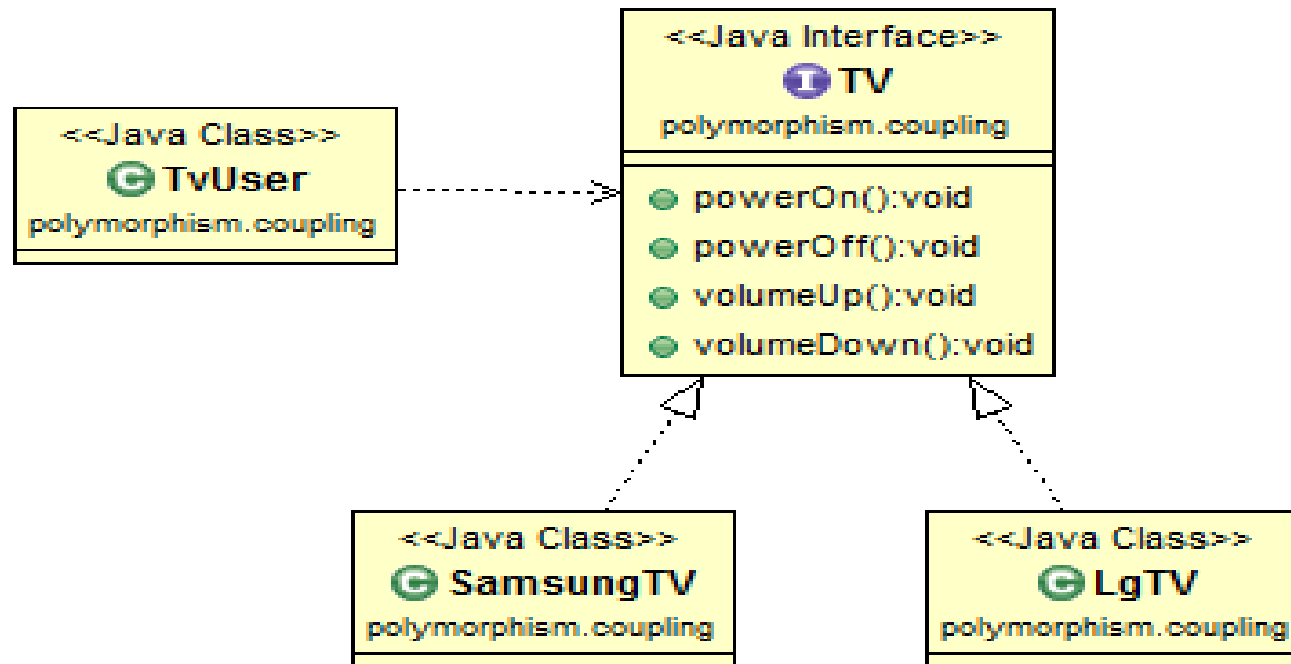
- IoC 컨테이너
 - 객체를 생성하고 조립
 - 객체의 생성을 책임지고 의존성을 관리
 - Bean : 컨테이너를 해 생성된 객체



- 스프링은 부품을 생성하고 조립하는 라이브러리 집합체

1.3 결합도 낮추기

- 다형성 이용하기 - 인터페이스



결합도 낮추기

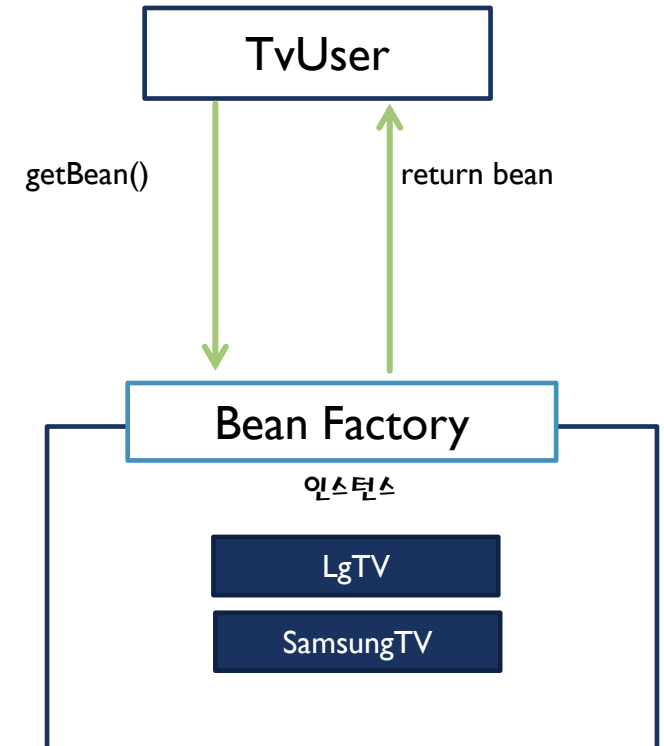
■ 디자인 패턴 이용 - Factory 패턴

클라이언트(TvUser)

```
public class TVUser {
    public static void main(String[] args) {
        TV tv = (TV)BeanFactory.getBean(args[0]);
        tv.powerOn();
        tv.volumeUp();
        tv.powerOff();
    }
}
```

Bean Factory

```
public class BeanFactory {
    public static Object getBean(String beanName) {
        if ( beanName.equals("samsung") ) {
            return new SamsungTV();
        } else if ( beanName.equals("lg") ) {
            return new LgTV();
        }
        return null;
    }
}
```



IoC 컨테이너

▶ ApplicationContext API

<https://docs.spring.io/spring/docs/4.3.23.RELEASE/javadoc-api/>

The screenshot shows the Javadoc API for the `org.springframework.context` package. The left sidebar lists interfaces and classes. The main content area shows the 'All Known Implementing Classes' list. Annotations with red arrows point to specific classes:

- 컨테이너 (Container):** Points to the `ApplicationContext` interface and its implementing classes.
 - 즉시 로딩(pre Loading)
 - 빈 객체 관리 기능외 에도 트랜잭션 관리, 다국어 처리 등을 지원
- Bean Factory:** Points to the `ApplicationContext` interface and its implementing classes.
 - 지연 로딩(Lazy Loading)
- 파일시스템이나 클래스 경로에 있는 XML 설정 파일을 로딩하여 구동하는 컨테이너:** Points to the `ClassPathXmlApplicationContext` class.
- 웹 기반의 스프링 애플리케이션을 개발할 때 사용. 직접 생성하지 않음:** Points to the `XmlWebApplicationContext` class.

XML을 이용한 DI 설정

■ ApplicationContext

클라이언트(TvUser)

```
public class TVUser {
    public static void main(String[] args) {
        //spring 컨테이너 구동
        AbstractApplicationContext factory =
            new GenericXmlApplicationContext("applicationContext.xml");

        //객체 요청
        TV tv = (TV)factory.getBean("tv");

        tv.powerOn();
        tv.volumeUp();
        tv.powerOff();

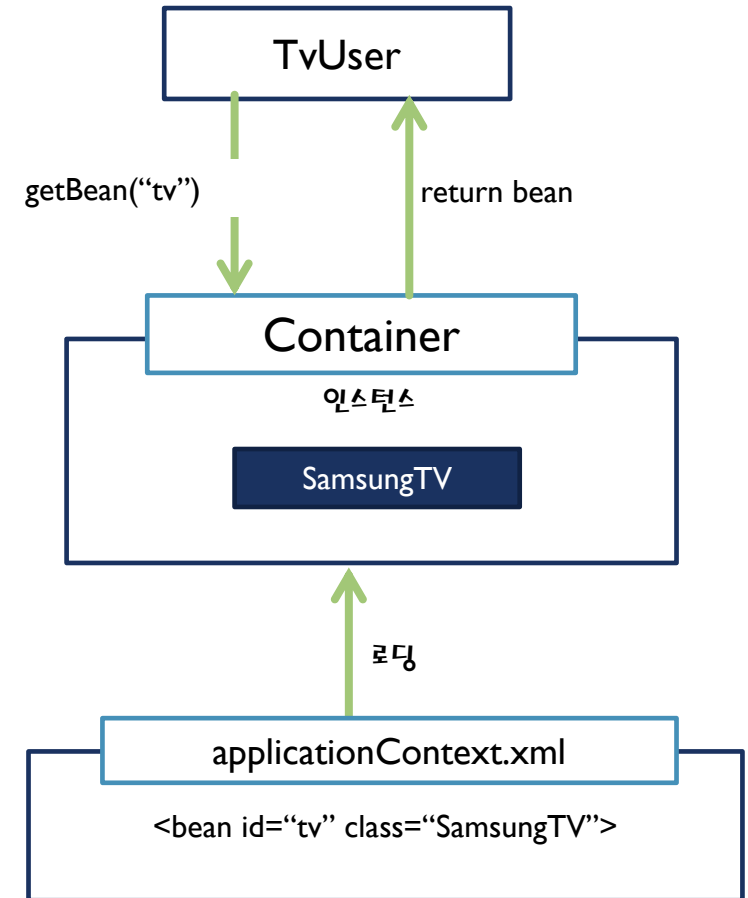
        //컨테이너 종료
        factory.close();
    }
}
```

스프링 설정파일(applicationContext.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

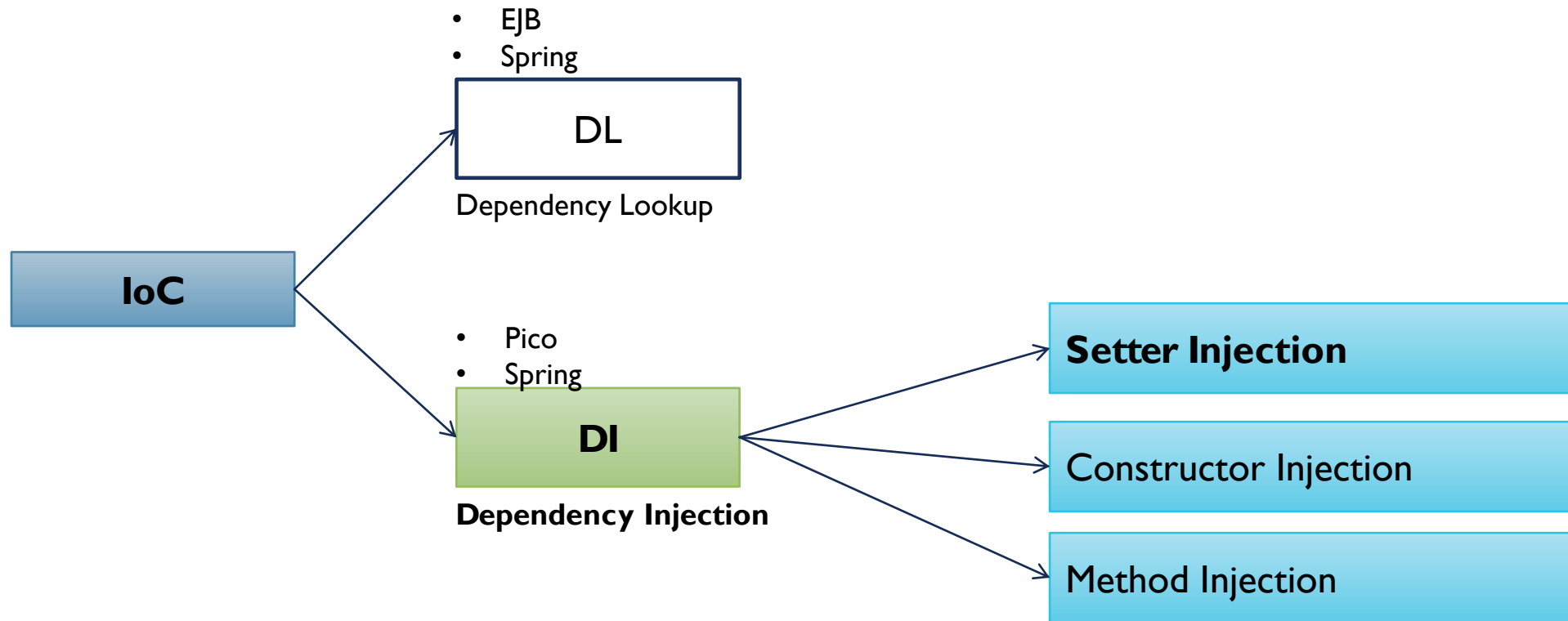
    <bean id="tv" class="polymorphism.SamsungTV"/>

</beans>
```



자바를 이용한 DI 설정

의존성 관리



DI(Dependency Injection)

- 객체 사이의 의존 관계를 스프링 설정 파일에 등록된 정보를 바탕으로 컨테이너가 자동으로 처리
- 의존관계를 변경하고 싶을 때 프로그램 코드 변경 없이 스프링 설정 파일 수정만으로 변경사항 적용

■ 생성자 인젝션(Constructor Injection)

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

<bean id="tv" class="polymorphism.SamsungTV">
  <constructor-arg ref="speaker"/>
</bean>
```

■ 세터 인젝션(Setter Injection)

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

<bean id="tv" class="polymorphism.SamsungTV">
  <property name="speaker" ref="speaker"></property>
</bean>
```

DI(Dependency Injection)

- ApplicationContext에 빈 등록하기 (applicationContext.xml)

```
<bean id="speaker" class="polymorphism.SonySpeaker"
      init-method="initMethod"
      destroy-method="destroyMethod"
      lazy-init="true"
      scope="singleton"/>

<!-- 생성자 인젝션 방식 -->
<bean id="samsung" class="polymorphism.SamsungTV" >
    <constructor-arg ref="speaker" />
    <constructor-arg value="2000" />
</bean>

<!-- setter 인젝션 방식 -->
<bean id="samsung" class="polymorphism.SamsungTV">
    <property name="speaker" ref="speaker"> </property>
    <property name="price" value="4000"> </property>
</bean>
```

3.1 어노테이션 기반 설정

컴포넌트 스캔 설정

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

빈 등록

어노테이션	설명
@Component	클래스 선언부에 설정(빈 등록)
@Service	비즈니스 로직을 처리하는 service 클래스
@Repository	데이터베이스 연동을 처리하는 DAO 클래스
@Controller	사용자 요청을 처리하는 Controller 클래스

의존성 주입

어노테이션	설명
@Autowired	해당 타입의 객체를 찾아서 자동으로 할당
@Qualifier	특정 객체의 이름을 이용하여 의존성 주입
@inject	@Autowired와 동일한 기능 제공
@Resource	@Autowired 와 @Qualifier의 기능을 결합

3.1 어노테이션 기반 설정

컴포넌트 스캔 설정

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

빈 등록

```
@Component  
public class SonySpeaker implements Speaker {
```

의존성 주입

```
@Component  
public class SamsungTV implements TV {  
  
    @Autowired  
    private Speaker speaker;
```

3.1 어노테이션 기반 설정

컴포넌트 스캔 설정

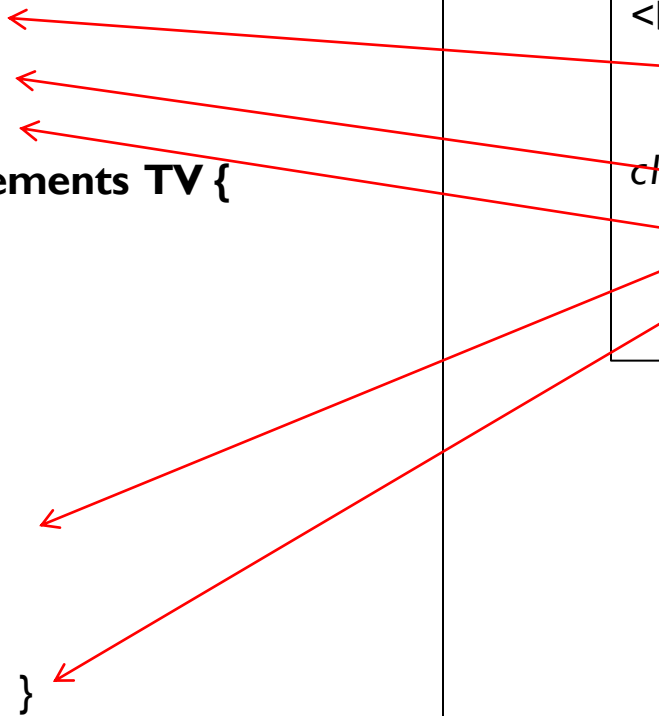
```
@Component("stv")  
@Scope("singleton")  
@Lazy("true")  
public class SamsungTV implements TV {
```

```
@Autowired  
private Speaker speaker;  
private int price;
```

```
@PostConstruct  
public void initMethod() { }
```

```
@PreDestroy  
public void destroyMethod() { }
```

```
<bean  
  id="speaker"  
  
  class="polymorphism.SonySpeaker"  
  scope="singleton"  
  lazy-init="true"  
  init-method="initMethod"  
  destroy-method="destroyMethod"/>
```



4.1 jUnit을 사용한 테스트

- jUnit의 개요
 - Java에서 독립된 단위테스트(unit Test)를 지원하는 프레임워크
 - 단위테스트란 소스 코드의 특정 모듈이 의도된 대로 정확히 작동하는지 검증하는 절차, 즉 모든 함수와 메소드에 대한 테스트 케이스(Test case)를 작성하는 절차
- jUnit의 특징
 - TDD의 창시자인 Kent Beck과 디자인 패턴 책의 저자인 Eric Gamma가 작성
 - 단정(assert) 메서드로 테스트 케이스의 수행결과를 판별한다.
 - 예) assertEquals(예상값, 실제값)
 - jUnit4부터는 테스트를 지원하는 어노테이션을 제공한다.
 - @Test, @Before, @After
 - 각 @Test 메서드가 호출될 때마다 새로운 인스턴스를 생성하여 독립적인 테스트가 이루어지도록 한다.
 - 결과는 성공(녹색), 실패(붉은색)중 하나로 표시

4.1 jUnit을 사용한 테스트

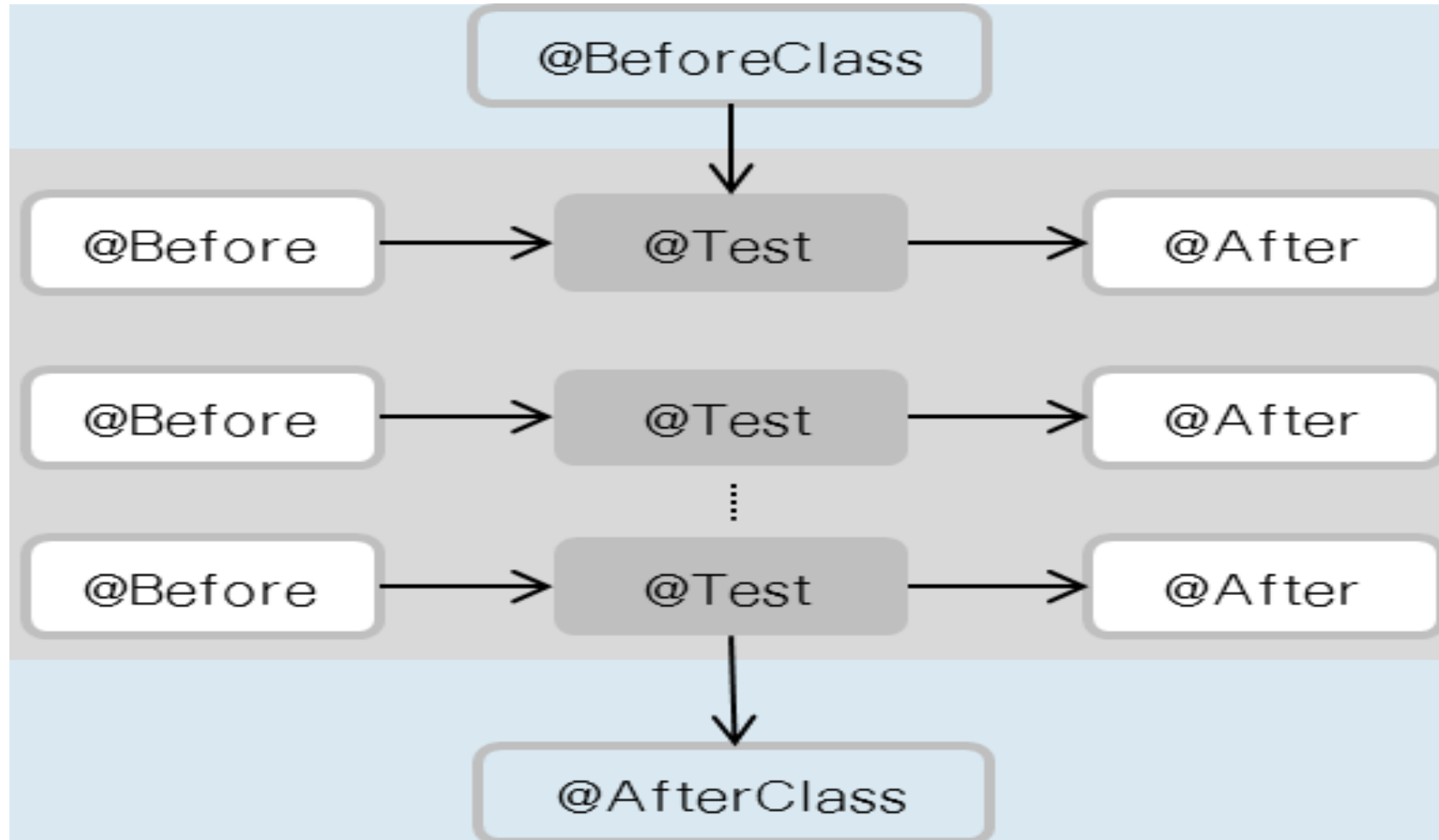
- jUnit에서 테스트를 지원하는 어노테이션
 - @Test
 - @Test가 선언된 메소드는 테스트를 수행하는 메소드가 된다.
 - Junit은 각각의 테스트가 서로 영향을 주지 않고 독립적으로 실행됨을 원칙으로 하므로 @Test마다 객체를 생성한다.
 - @Ignore
 - @Ignore가 선언된 메소드는 테스트를 실행하지 않게 한다.
 - @Before
 - Before가 선언된 메소드는 @Test 메소드가 실행되기 전에 먼저 실행
 - @Test 메소드가 공통으로 사용하는 코드를 @Before 메소드에 선언하여 사용

4.1 jUnit을 사용한 테스트

- jUnit에서 테스트를 지원하는 어노테이션
 - @After
 - @After가 선언된 메소드는 @Test 메소드가 실행된 후 실행
 - @BeforeClass
 - @BeforeClass 어노테이션은 @Test 메소드보다 먼저 한번만 수행되어야 할 경우에 사용하면 된다.
 - @AfterClass
 - AfterClass 어노테이션은 @Test 메소드보다 나중에 한번만 수행되어야 할 경우에 사용하면 된다.

4.1 jUnit을 사용한 테스트

- jUnit에서 테스트를 지원하는 어노테이션



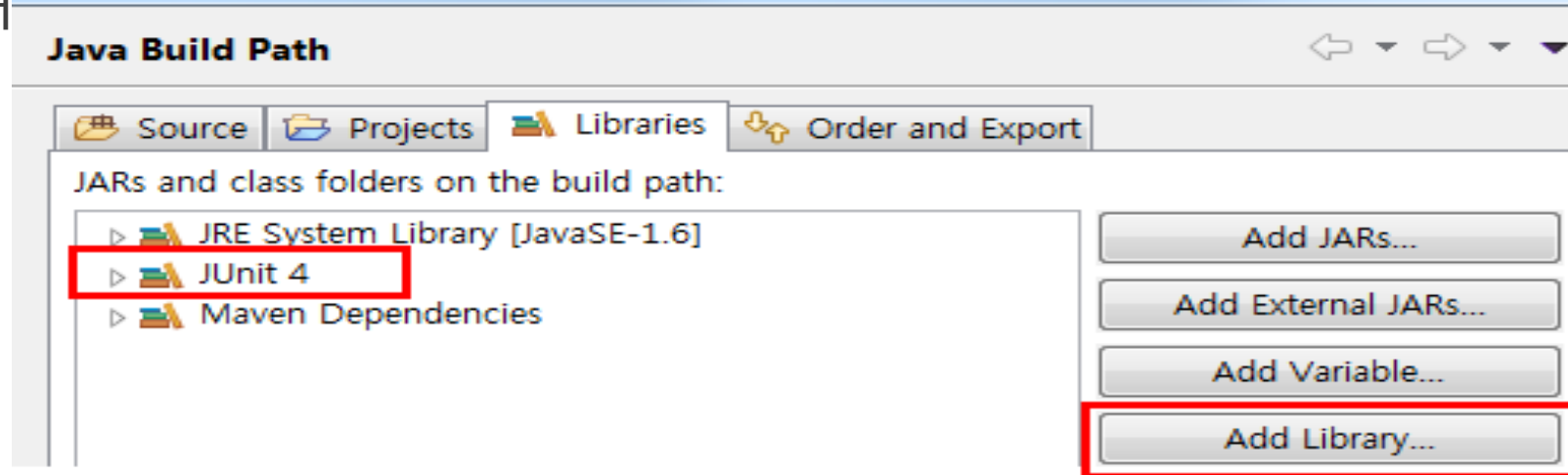
4.1 jUnit을 사용한 테스트

- 테스트 결과를 확인하는 단정(assert) 메서드
 - `assertArrayEquals(a,b)`
 - 배열 a와b가 일치함을 확인
 - `assertEquals(a,b)`
 - 객체 a와b의 값이 같은지 확인
 - `assertSame(a,b)`
 - 객체 a와b가 같은 객체임을 확인
 - `assertTrue(a)`
 - a가 참인지 확인
 - `assertNotNull(a)`
 - a객체가 null이 아님을 확인

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

4.1 jUnit을 사용한 테스트

- 이클립스에서 jUnit 라이브러리 추가
 - Project 이름에서 오른쪽 마우스를 클릭하고 Properties를 선택
 - Java BuildPath를 선택
 - Libraries 탭을 선택하고, Add Library를 선택
 - JUnit을 선택하고, Next 버튼 선택
 - 버튼 선택하고, Next 버튼 선택



4.1 jUnit을 사용한 테스트



```
<dependency>  
  <groupId>junit</groupId>  
  <artifactId>junit</artifactId>  
  <version>4.12</version>  
</dependency>
```

```
<dependency>  
<groupId>org.springframework</groupId>  
  <artifactId>spring-test</artifactId>  
  <version>${org.springframework-version}</version>  
</dependency>
```

4.2 Spring-test를 사용한 테스트

- Spring-test에서 테스트를 지원하는 어노테이션
 - `@RunWith(SpringJUnit4ClassRunner.class)`
 - `@RunWith`는 junit 프레임워크의 테스트 실행방법을 확장할 때 사용하는 어노테이션이다.
 - `SpringJUnit4ClassRunner`라는 클래스를 지정해주면 `ApplicationContext`를 만들고 관리하는 작업을 진행해준다.
 - `@RunWith` 어노테이션은 각각의 테스트별로 객체가 생성되더라도 싱글톤(singleton)의 `Application Context`를 보장한다.
 - `@ContextConfiguration`
 - 스프링 빈(Beans) 설정 파일의 위치를 지정할 때 사용되는 어노테이션이다.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:spring/*-context.xml")
public class BoardClient {
    @Autowired ApplicationContext context;

    @Test
    public void dataSourceTest() throws SQLException {
        DataSource ds = (DataSource) context.getBean("dataSource");
        System.out.println(ds.getConnection());
    }
}
```

5.1 Dynamic Web Project 를 Spring 프로젝트로 변경

- Maven 프로젝트로 변경
 - Configure 컨텍스트메뉴 -> [convert to maven project](#)
- Spring 프로젝트로 변경
 - Spring 컨텍스트메뉴 -> [add Spring Project Nature](#)
- Spring 라이브러리 설치
 - <https://mvnrepository.com/> Spring context 검색
 - 5.3.16 버전 선택하여 pom.xml 에 복사
 - Maven Dependencies에서 jar 파일이 추가되었는지 확인
- Spring 설정파일 추가
 - File 메뉴 -> new -> Spring Bean Configuration File
 - XSD namespace 정의에서 **context** 선택