초급 백엔드 스터디

2주차 – 스프링 빈 & 컨테이너

지난 주에는...

- 웹의 구조
- 백엔드의 역할, API
- API 명세서 작성

이번 주에는...

- **스프링 빈**과 **스프링 컨테이너**의 개념
- 스프링 컨테이너에 빈을 저장하는 방법
- 스프링 컨테이너에서 빈을 받아오는 방법

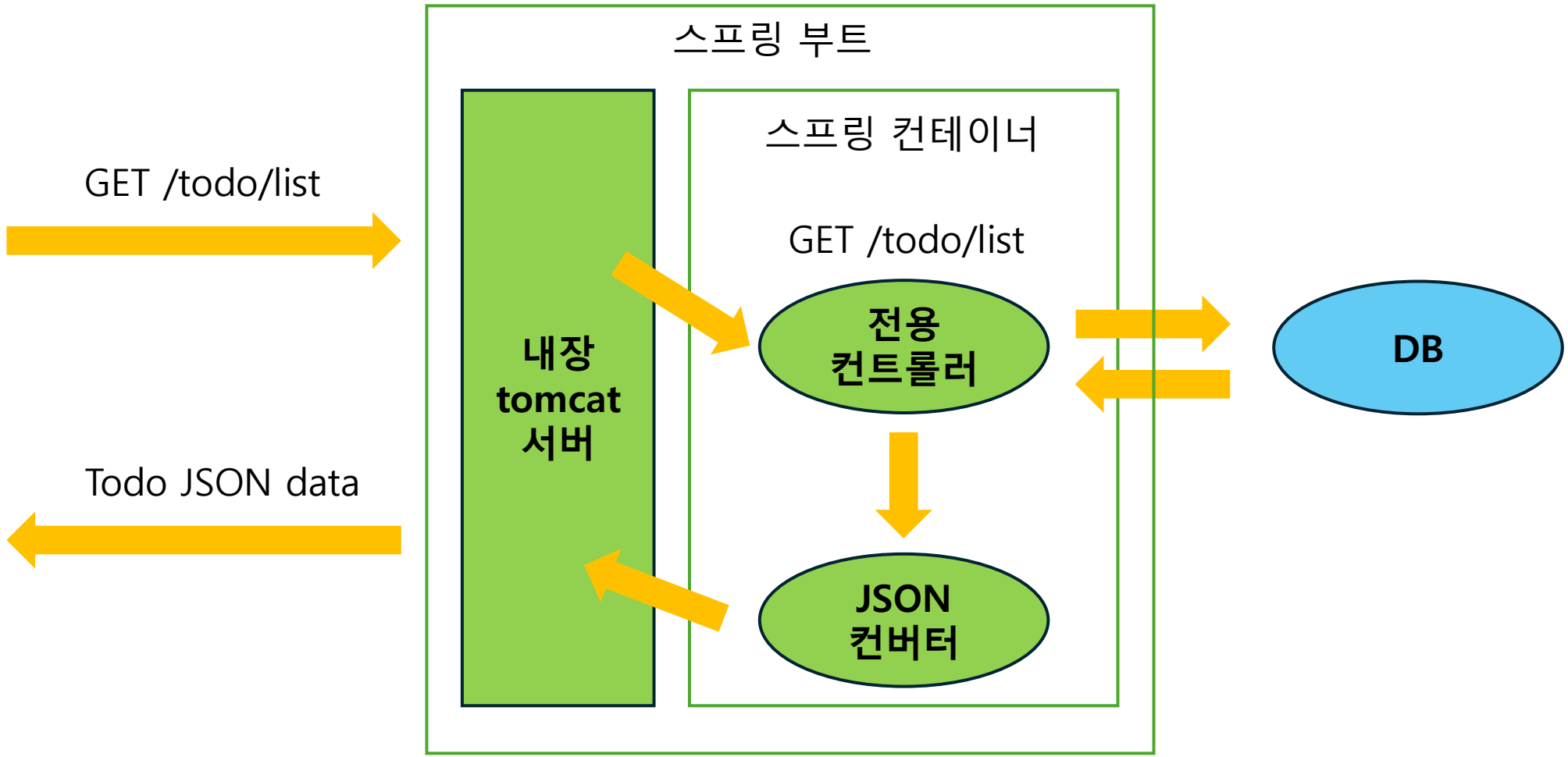
스프링

- JAVA 진영의 대표적인 백엔드 프레임워크
- **객체지향 원칙**을 지키면서 개발할 수 있도록 도와준다.

스프링 부트

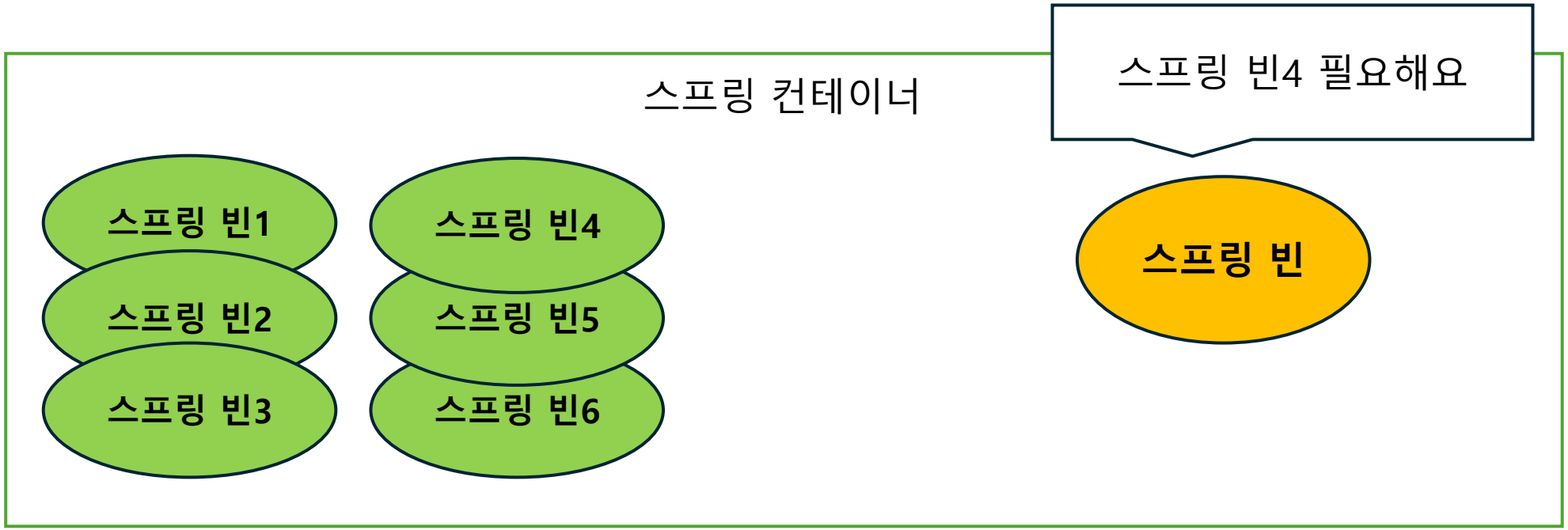
- 스프링 프레임워크를 사용하여 개발할 때, 편의를 더해주는 도구
- 스프링으로 개발할 때는 **스프링 부트**를 **함께** 사용한다.

스프링 어플리케이션 구조



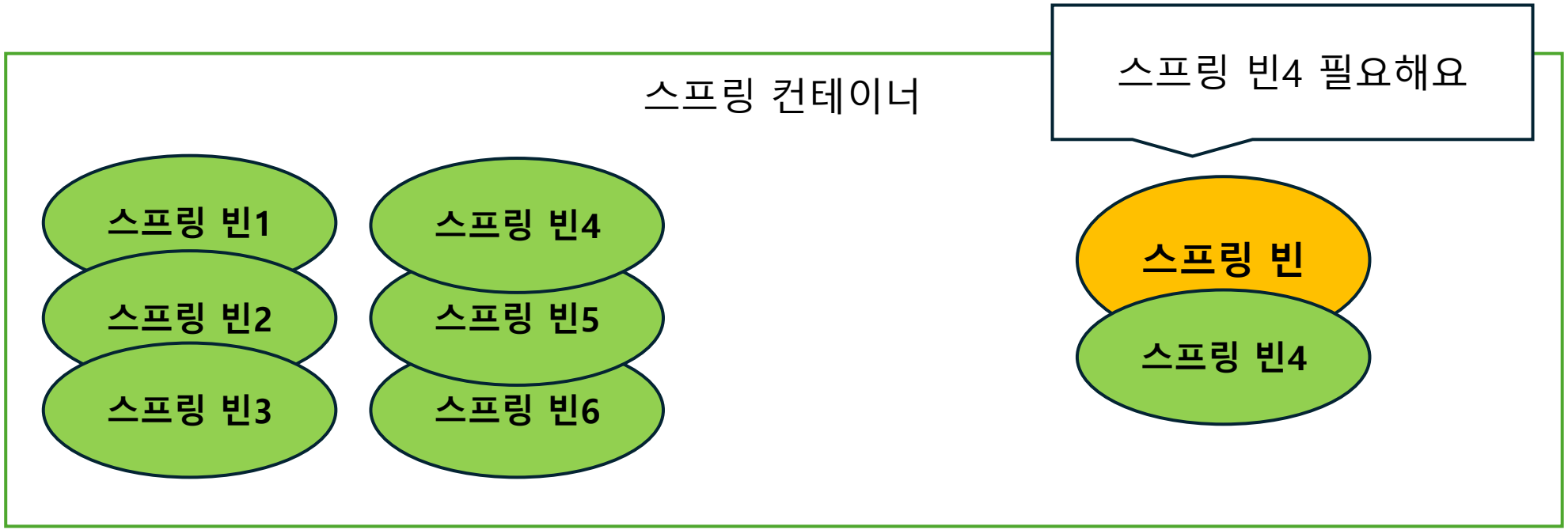
스프링 빈 (Spring Bean)

- 어플리케이션 전역에서 사용할 **공용** 객체
- **스프링 컨테이너**라고 하는 공용 창고에 **빈(객체)**을 저장해두고, 필요한 빈을 컨테이너에서 받아 사용한다.



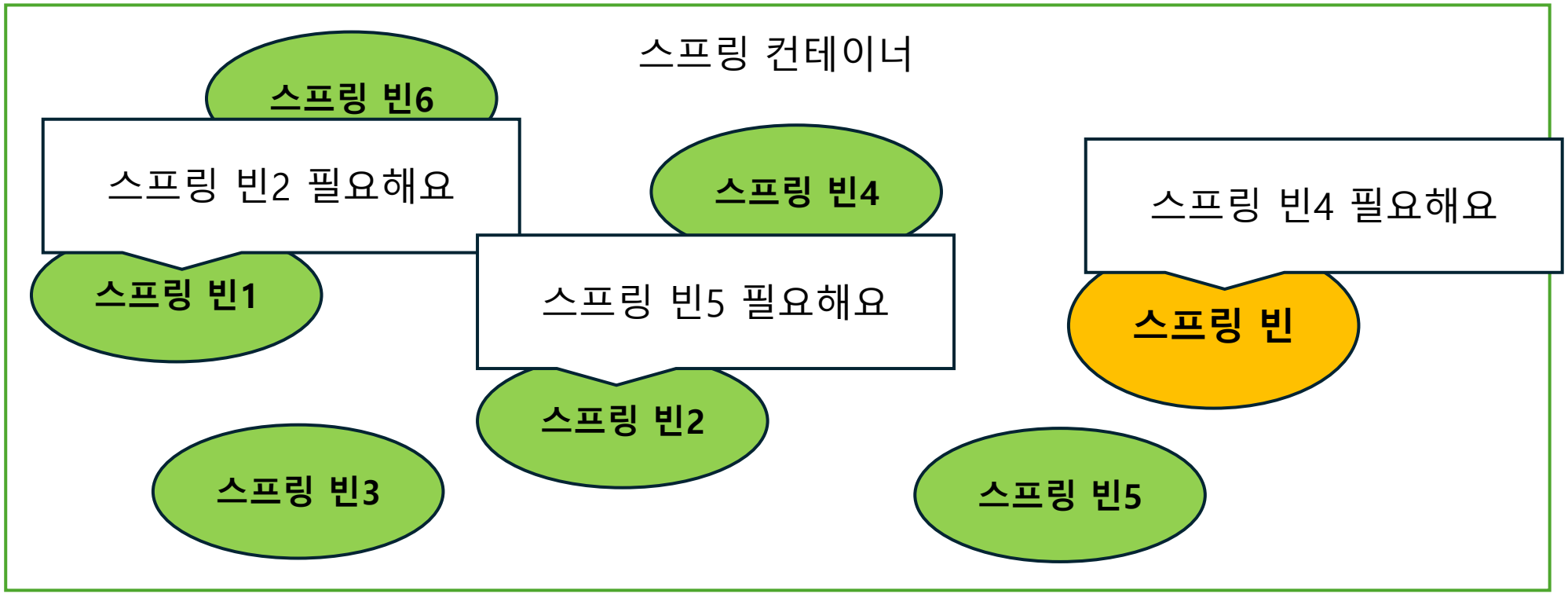
스프링 빈 (Spring Bean)

- 필요한 빈은 스프링 프레임워크가 자동으로 가져다 준다.
- 이때 **빈을 요구하는 객체도 스프링 빈**이다.
  (빈이 아닌 객체가 빈을 요구해도 프레임워크가 자동으로 가져다주지는 못한다.)



스프링 빈 (Spring Bean)

- 즉, 빈을 사용하는 주체 역시 스프링 빈이므로 서로가 서로를 필요로 하는 구조로 되어있다.



스프링 컨테이너

- 스프링 빈이 저장되는 공간
- **어플리케이션 컨텍스트(Application Context)** 라고도 한다.

우리가 알아야 하는 내용

- 스프링 컨테이너에 **스프링 빈을 저장하는 방법**
- 스프링 컨테이너에서 **스프링 빈을 받아오는 방법**

스프링 빈 저장

- 스프링 빈을 스프링 컨테이너에 저장하는 2가지 방법

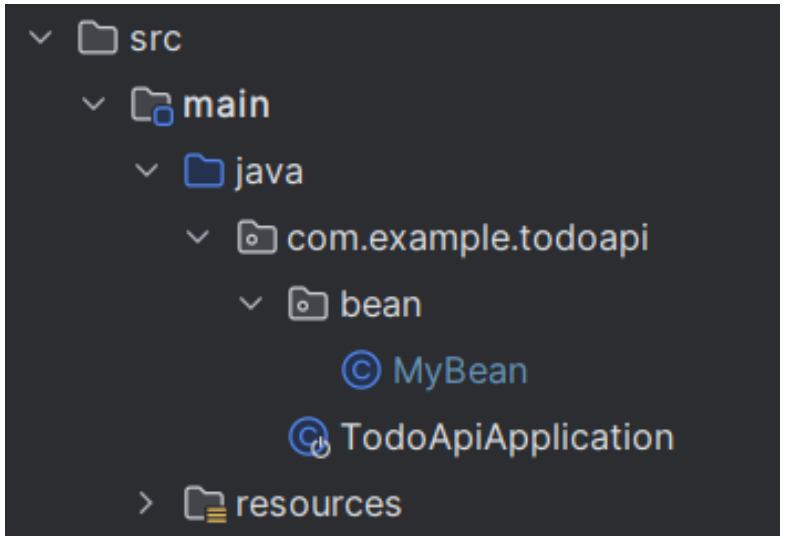
- **설정 파일 작성 (수동 등록)**
- 컴포넌트 스캔 (자동 등록)

설정 파일 작성

- 설정 파일은 **자바 클래스로 작성**한다.
- 이때 클래스에 **@Configuration** 으로 설정 파일임을 명시한다.

스프링 빈 클래스 생성

- 스프링 빈(객체)을 생성할 클래스를 만들자.
- **main** 폴더 밑에 **bean** 패키지를 만들고 **MyBean** 클래스 생성

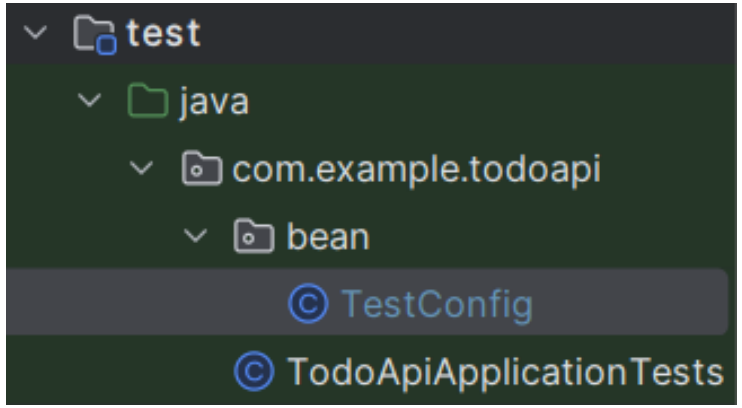



```
package com.example.todoapi.bean;


8 usages  new *
public class MyBean {}
```

설정 파일 작성

- **test** 폴더 밑에 **bean** 패키지 생성 – **TestConfig** 클래스 생성
- 클래스에 **@Configuration**, 메서드에 **@Bean** 어노테이션 사용

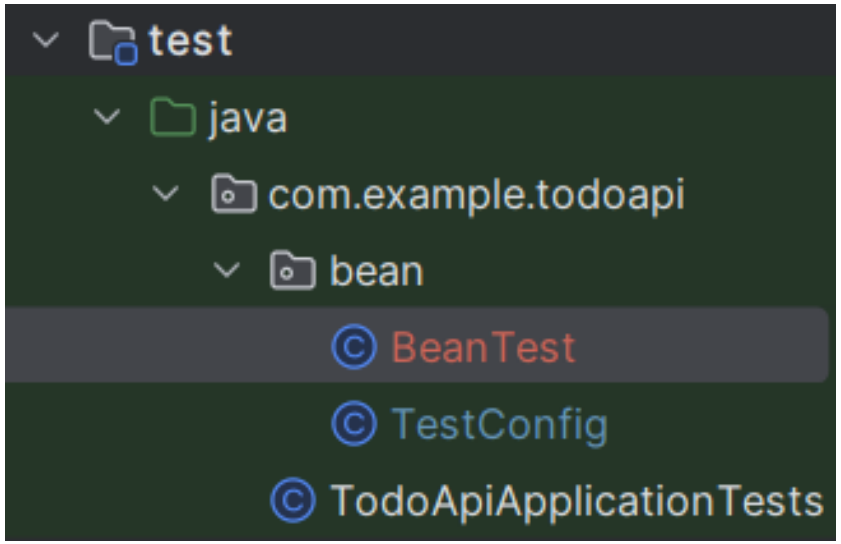



```java
@Configuration
public class TestConfig {

    new *
    @Bean
    MyBean myBean() {
        return new MyBean();
    }
}
```

설정 파일 작성

- 테스트용 컨테이너를 만들어서 빈을 등록하고 확인해보자.
- **bean** 패키지 안에 **BeanTest** 클래스 생성



설정 파일 작성 – 테스트

- **BeanTest** 클래스 안에 **ApplicationContext** 으로 컨테이너 생성

  → TestConfig를 이용하여 테스트 전용 스프링 컨테이너를 따로 만든다.

```
kckc0608 *
public class BeanTest {

    5 usages
    ApplicationContext context = new AnnotationConfigApplicationContext(TestConfig.class);
```

참고) ApplicationContext는 인터페이스, Annotation...Context는 그 구현체 클래스이다.
@Bean 과 같은 어노테이션을 사용하여 빈을 생성하므로 **어노테이션 컨테이너**를 사용한다.

스프링 컨테이너를 생성할 때, 컨테이너에 등록할 빈 정보가 담긴 Config 클래스를 넘길 수 있다.

설정 파일 작성 – 테스트

- 컨테이너 안에 등록된 모든 빈 조회

```java
@Test
void checkAllBeans() {
    for (String name : context.getBeanDefinitionNames()) {
        System.out.println(name);
    }
}
```

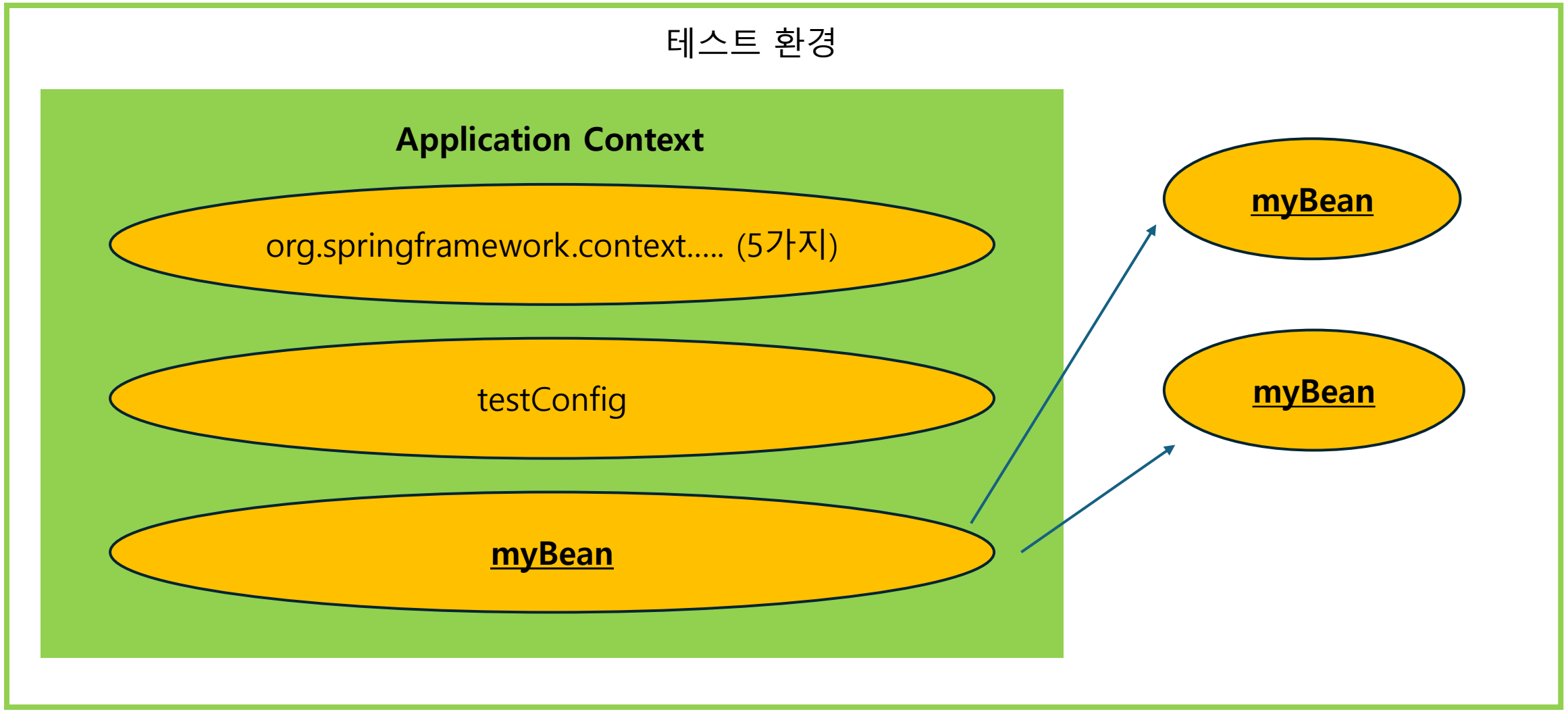
설정 파일 작성 – 테스트

- 컨테이너 안에 등록된 모든 빈 조회

```
@Test
void getAllBeans() {
    for (String name: context.getBeanDefinitionNames()) {
        System.out.println(name);
    }

    Assertions.assertThat(context.getBeanDefinitionNames()).contains("myBean");
}
```

설정 파일 작성 – 테스트


테스트 환경
Application Context
org.springframework.context..... (5가지)
testConfig
myBean
myBean
myBean

설정 파일 작성 – 테스트

- 컨테이너에서 원하는 빈을 선택해서 가져오기

```
@Test
void getOneBean() {
    MyBean bean = context.getBean(MyBean.class);
    System.out.println(bean);
}
```

```
"C:\Program Files\JetBrains\IntelliJ IDEA
com.example.todoapi.bean.MyBean@2f4205be

Process finished with exit code 0
```

설정 파일 작성 – 테스트

- 스프링 빈은 기본적으로 **1개의 객체**이다.

  따라서 컨테이너에서 빈을 가져올 때마다 같은 객체가 반환된다.

```java
@Test
void checkMyBean() {
    MyBean myBean1 = context.getBean(MyBean.class);
    MyBean myBean2 = context.getBean(MyBean.class);
    System.out.println(myBean1.getClass().getName());
    System.out.println(myBean1);
    System.out.println(myBean2);

    Assertions.assertThat(myBean1).isSameAs(myBean2);
}
```

```
com.example.todoapi.bean.MyBean
com.example.todoapi.bean.MyBean@3704122f
com.example.todoapi.bean.MyBean@3704122f
```

스프링 빈 저장

- 스프링 빈을 스프링 컨테이너에 저장하는 2가지 방법

- 설정 파일 작성 (수동 등록)
- **컴포넌트 스캔 (자동 등록)**

컴포넌트 스캔

- 빈을 생성할 클래스에 **@Component** 어노테이션 사용
- 어플리케이션을 시작할 때 **@Component**가 붙은 클래스를 찾아서 자동으로 빈 등록

```
@Component
public class MyBean {
}
```

컴포넌트 스캔 – 테스트

- 컴포넌트 스캔은 **@ComponentScan** 어노테이션 사용
- 기존 **TestConfig** 내용을 모두 지우고, **@ComponentScan** 추가 스캔해서 발견한 컴포넌트를 빈으로 등록한다.

```
@Configuration
@ComponentScan
public class TestConfig {
}
```

컴포넌트 스캔 – 테스트

- 기존에 작성한 테스트가 똑같이 동작하는 것을 알 수 있다.

컴포넌트 스캔 – 테스트

Q) 컴포넌트 스캔을 할 때도 Config 파일이 필요한 것 아닌가요?

```java
@SpringBootApplication
public class TodoApiApplication {

    kckc0608
    public static void main(String[] args) {
        SpringApplication.run(TodoApiApplication.class, args);
    }

}
```

컴포넌트 스캔 – 테스트

Q) 컴포넌트 스캔을 할 때도 Config 파일이 필요한 것 아닌가요?

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type =
        @Filter(type = FilterType.CUSTOM, classes
public @interface SpringBootApplication {
```

스프링 빈 등록 정리

- 설정 파일 작성 **@Configuration, @Bean**
- 컴포넌트 스캔 **@Component**, @ComponentScan

스프링 빈 등록 정리

- 내가 등록할 빈을 생성하는 클래스에 **@Component**를 붙이면 끝!

의존성 주입

- 빈을 사용할 때는 컨테이너에 직접 접근해서 빈을 꺼내지 않고, 프레임워크에게 **필요한 빈(의존성)을 요청**하고 받아서 사용한다.

- 구체적인 방법을 보기 전에 먼저 **'의존성'** 개념을 살펴보자.

의존성 주입

- 자동차가 움직이려면 반드시 바퀴가 필요하다.
  즉, 자동차는 바퀴에 **의존**한다.

```
no usages
public class Car {
    1 usage
    private Wheel wheel;

    no usages
    public void move() {
        wheel.roll();
    }
}
```

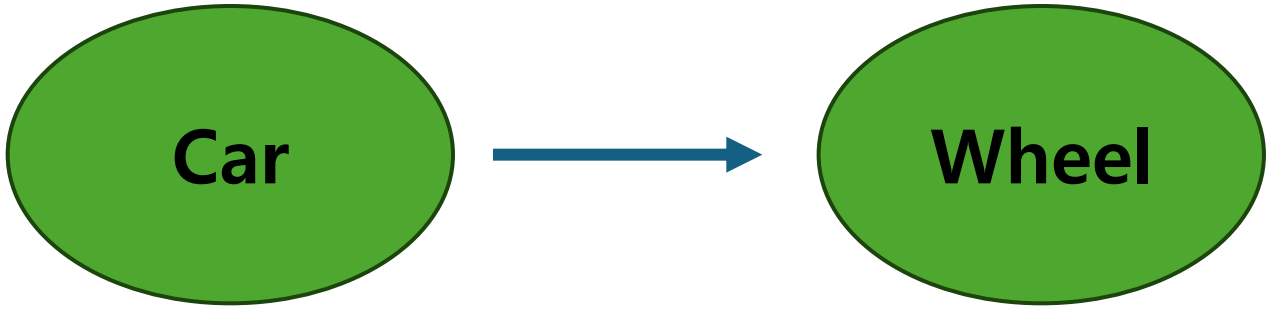
```
1 usage
public class Wheel {

    1 usage
    public void roll() {}
}
```

의존성 주입

- Car클래스의 move() 메서드를 실행하려면 Wheel 객체가 필요
- A의 기능을 실행하는데 B의 기능이 필요하다면,

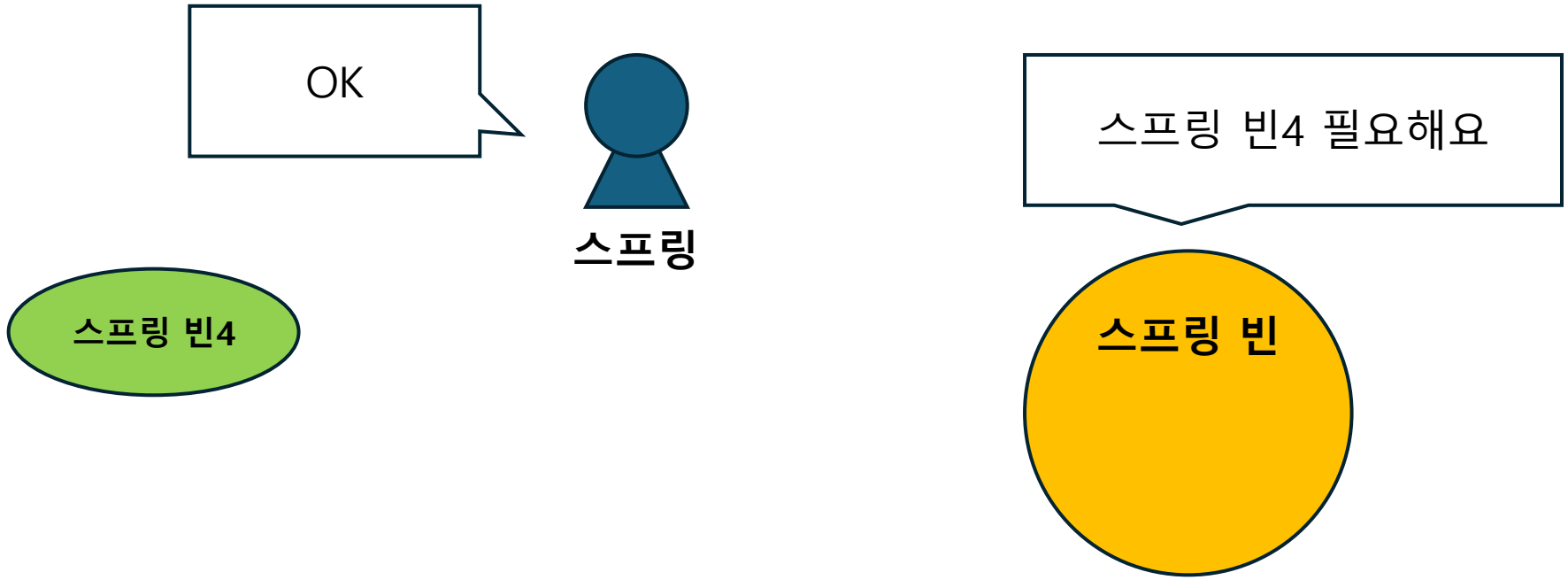
  'A는 B에 의존한다'고 한다.



의존성 주입

- **의존성 주입 (Dependency Injection, DI)**

  : 내가 의존하는 객체를 직접 생성하지 않고 밖에서 주입 받는 것

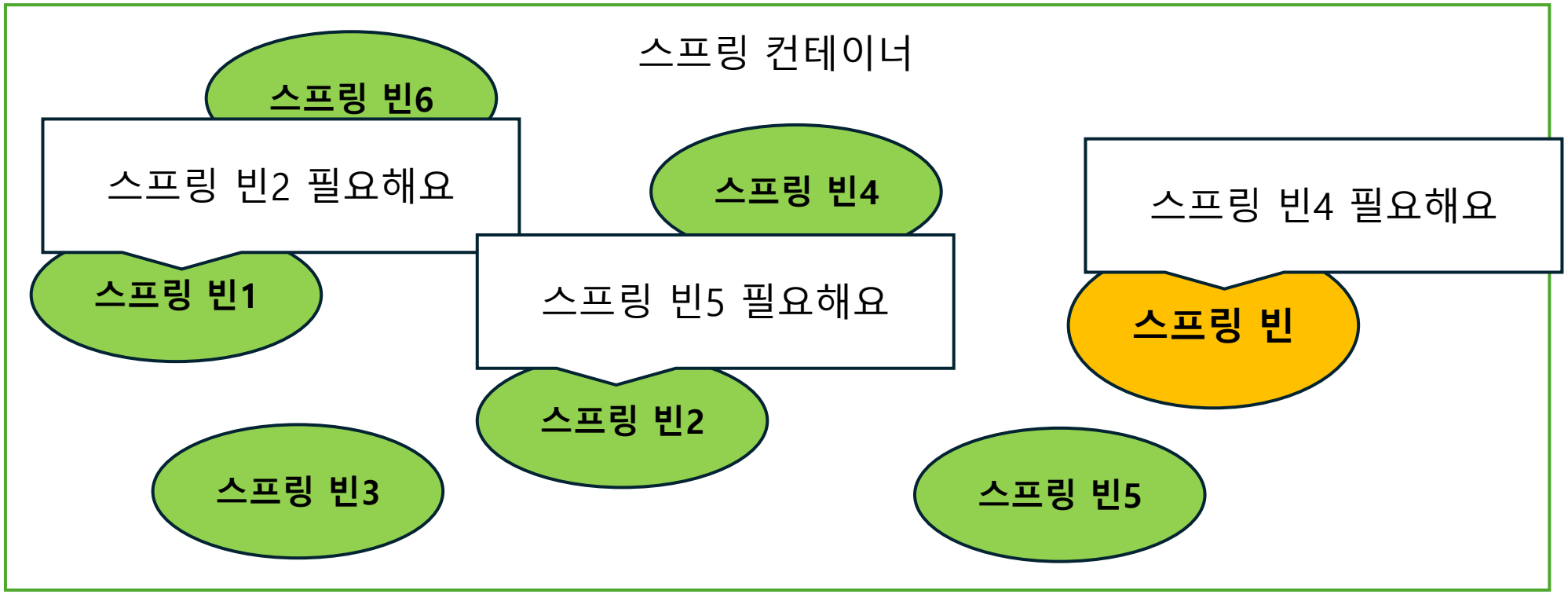


의존성 주입

- 스프링에서는 컨테이너에 저장된 빈(객체)과 빈(객체)사이의 의존성을 프레임워크가 주입하는 것을 말한다.

Cf) 빈이 아닌 객체에 빈을 자동으로 주입할 수는 없다. 스프링이 빈을 주입하려면, 두 객체 모두 스프링에 의해 관리 (=빈으로 등록) 되어야 하기 때문이다.

의존성 주입

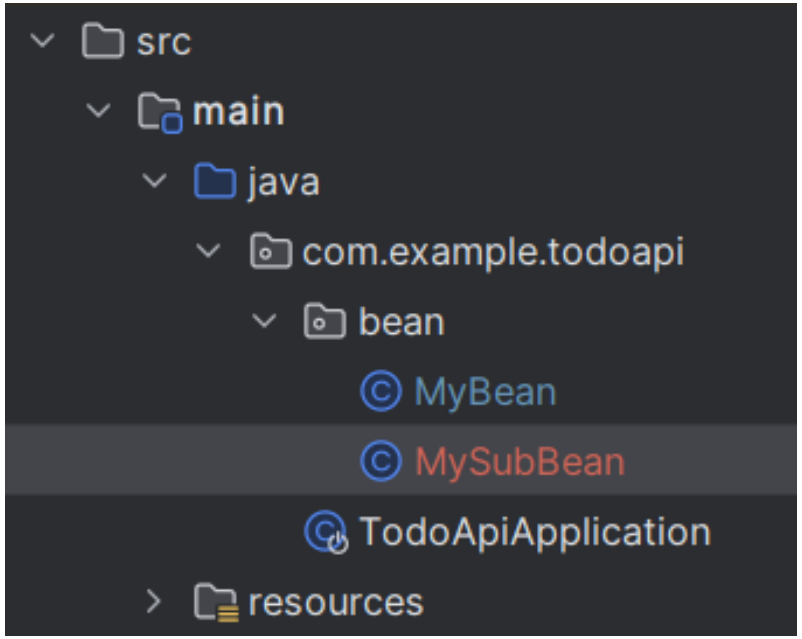

스프링 컨테이너
스프링 빈6
스프링 빈2 필요해요
스프링 빈4
스프링 빈4 필요해요
스프링 빈1
스프링 빈5 필요해요
스프링 빈
스프링 빈2
스프링 빈3
스프링 빈5

의존성을 주입 받는 이유

- 객체 지향 원칙 중 하나인 **OCP(Open Close Principle) 원칙**을 준수한다.
  (필요한 객체를 내가 하드코딩하지 않기 때문에 **유지보수**하기 좋아진다)
- 매번 필요한 객체를 생성하는 대신, 생성해둔 객체를 사용하므로 **메모리를 효율적으로 사용**할 수 있다.

의존성 클래스 추가

- **bean** 패키지에 **MySubBean** 클래스를 생성 후 빈으로 등록한다.




```
@Component
public class MySubBean {}
```

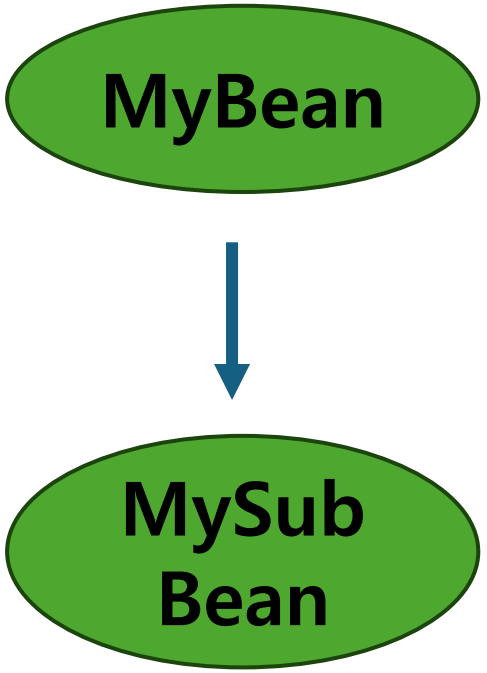
의존성 클래스 추가

- **MyBean**에 **MySubBean** 의존성과 **@Getter**를 추가한다.

```
@Getter
@Component
public class MyBean {

    private MySubBean mySubBean;

}
```



의존성 주입 방법

- 어떤 객체에 다른 객체를 주입하려면, 주입할 **통로**가 필요하다.
- 우리는 통로를 만들고, 이 통로를 통해 주입해 달라고 **표시**해두면 프레임워크가 알아서 객체를 주입해준다.

- 통로는 크게 생성자, 필드, 메서드가 존재한다.
- 표시를 남길 때는 **@Autowired** 어노테이션으로 표시를 남긴다.

의존성 주입 방법

- **생성자 주입**
- 필드 주입
- 세터 주입 (메서드 주입)

생성자 주입

- 의존성이 바뀔 일이 없는 경우 안전하게 **final**로 선언한다.
- 이때 final 필드는 생성자를 통해 초기화되어야 한다.

```java
@Getter
@Component
public class MyBean {

    private final MySubBean mySubBean;

    new *
    public MyBean(MySubBean mySubBean) {
        this.mySubBean = mySubBean;
    }
}
```

생성자 주입

- 생성자에 **@Autowired** 을 사용하면, 생성자를 통해 빈을 주입한다.
- 만약 **생성자가 하나**만 있다면, **@Autowired를 생략**할 수 있다.

생성자 주입

- **@RequiredArgsConstructor**를 사용하면
  모든 final 필드에 대한 생성자를 자동으로 만들어주어
  생성자 코드까지 생략할 수 있다.

```
6 usages   kckc0608 *
@Getter
@Component
@RequiredArgsConstructor
public class MyBean {

    private final MySubBean mySubBean;
}
```

생성자 주입

- 생성자 주입 방법을 정리하면 다음과 같다.
  1. 필요한 의존성을 **final** 키워드를 사용해 추가한다.
  2. **@RequiredArgsConstructor**를 사용해 생성자를 추가한다.

```
6 usages   kckc0608 *
@Getter
@Component
@RequiredArgsConstructor
public class MyBean {

    private final MySubBean mySubBean;
}
```

생성자 주입 – 테스트

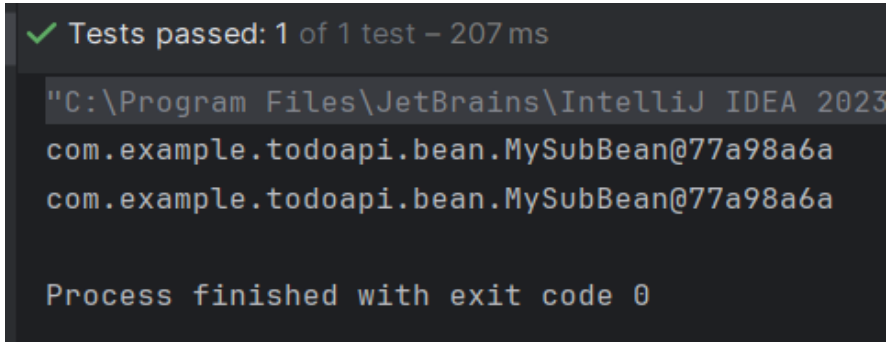
- 생성자를 통해 의존성이 주입되는지 **테스트**해보자.
- 이때 스프링 빈을 의존성으로 주입했다면,
  MyBean 객체가 갖고 있는 **MySubBean** 객체와
  스프링 컨테이너에 들어있는 **MySubBean** 객체는 동일할 것이다.

생성자 주입 – 테스트

- **BeanTest** 클래스에 의존성 테스트를 작성한다.

```
@Test
void checkMyBeanHasMySubBean() {
    MyBean myBean = context.getBean(MyBean.class);
    MySubBean mySubBean = context.getBean(MySubBean.class);

    System.out.println(myBean.getMySubBean());
    System.out.println(mySubBean);
```



의존성 주입 방법

- 생성자 주입
- **필드 주입**
- 세터 주입 (메서드 주입)

필드 주입

- 필드에 바로 **@Autowired** 어노테이션을 사용한다. (final은 사용 불가)
- 이 방식은 주로 **테스트 코드에서 사용**한다.
  (운영 코드에서 사용하면 IDE에서 경고를 띄운다.)

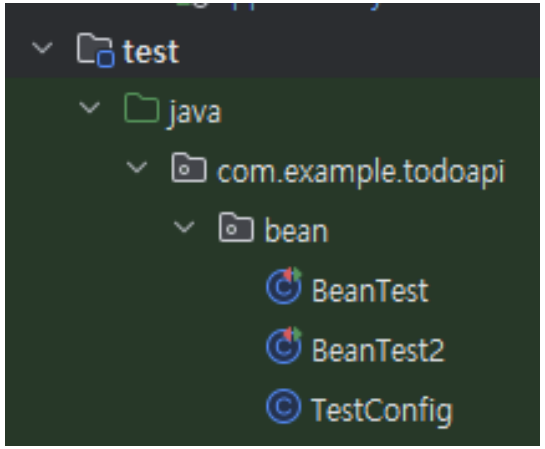
```
@Getter
@Component
public class MyBean {

    @Autowired
    private MySubBean mySubBean;
}
```

Field injection is not recommended

필드 주입 – 테스트

- 테스트에서 필드 주입을 하려면, 테스트를 실행할 때
  이미 스프링 컨테이너가 존재해야 한다.
- 이를 위해 **bean** 패키지에 새롭게 **BeanTest2** 클래스를 생성한다.



필드 주입 – 테스트

- 클래스에 **@SpringBootTest** 어노테이션을 사용하면
  어플리케이션에 있는 **모든 빈을 컨테이너에 등록**한 후 테스트한다.
- 스프링 부트까지 실행시키므로, **통합 테스트에도 사용**할 수 있다.

```
no usages
@SpringBootTest
public class BeanTest2 {

}
```

필드 주입 – 테스트

- 모든 빈이 컨테이너에 들어있는 상태에서 테스트를 진행하므로 원하는 빈을 **필드 주입** 받아서 테스트할 수 있다.

```java
@SpringBootTest
public class BeanTest2 {

    @Autowired
    MyBean myBean;

    @Autowired
    MySubBean mySubBean;

    @Test
    void checkMySubBean() {
        System.out.println(myBean);
        System.out.println(mySubBean);

        Assertions.assertThat(myBean.getMySubBean()).isSameAs(mySubBean);
    }
}
```

스프링 빈 활용

- 스프링 빈과 컨테이너를 실제 개발에서는 어떻게 활용할까?

스프링 Layered Architecture

스프링 Layered Architecture

컨트롤러

- 클라이언트의 요청을 받고, 응답을 보내는 계층
- **DTO** (Data Transfer Object)를 사용하여
  서비스 계층과 데이터를 주고받는다.

스프링 Layered Architecture

서비스

- 어플리케이션의 비즈니스 로직이 담기는 계층
- **레포지토리 계층**과 소통하며 **엔티티, 또는 DTO**로 소통한다.

스프링 Layered Architecture

레포지토리

- **DB**와 소통하며 데이터를 조작하는 계층
- 서비스 계층이 결정한 비즈니스 로직을 실제 DB에 적용한다.

스프링 빈 활용

- **컨트롤러, 서비스, 레포지토리**는 스프링 빈으로 등록한다.
- 매번 새로운 객체를 생성할 필요가 없고,
  객체지향 원칙을 준수하며 의존성을 관리할 수 있기 때문이다.

정리

- 스프링 빈 = 공동으로 사용할 **하나의 객체**
- 스프링 컨테이너 = 빈을 저장하는 공용 **공간**
- 의존성 주입 = 프레임워크가 필요한 빈을 주입하는 것

- 컨테이너에 빈을 **저장**하는 방법 : 설정 파일 / **컴포넌트 스캔**
- 컨테이너에서 빈을 **주입**받는 방법 : **생성자 주입** / 필드 주입

과제

- 스프링 계층 구조를 구성하는 컨트롤러, 서비스, 레포지토리를 직접 빈으로 등록하고 의존성을 주입해봅니다.

- 자세한 내용은 노션 과제 명세를 참고해주세요.

수고하셨습니다 ☺

- 축제 재밌게 즐기세요~!