8. IoC와 DI (Inversion of Control / Dependency Injection)

IoC 컨테이너 개념

다음은 **Spring의 IoC 컨테이너 개념**에 대한 가장 깊이 있는 설명이다. 기본 정의부터 시작하여, Bean 생성 및 관리 원리, 컨테이너 종류, 내부 동작 흐름, 실제 코드 예시, 일반적인 오해까지 포함해 학습용으로 정리하였다.

IoC 컨테이너란?

1. 개념 정의

IoC(Inversion of Control, 제어의 역전)는 **객체의 생성과 생명주기를 개발자가 아닌 외부 컨테이너가 제어**하도록 하는 설계 원칙이다.

- 객체 간의 **의존성을 코드 내부에서 직접 new로 생성하지 않고**, 외부에서 주입 받는 방식
- **Spring Framework의 핵심 기반 철학**

Spring에서 이 원칙을 실현하는 **기술적 구현체**가 바로 **IoC 컨테이너**다.

---

2. 예제 비교: IoC 적용 전 vs 후

IoC 적용 전 (직접 의존성 주입)

```
class OrderService {
    private PaymentService payment = new CreditCardPaymentService();
}
```

IoC 적용 후 (컨테이너에서 주입)

```
class OrderService {
    private final PaymentService payment;

    public OrderService(PaymentService payment) {
        this.payment = payment;
    }
}
```

→ 이 객체를 생성하고 PaymentService를 주입하는 일은 **Spring IoC 컨테이너가 수행**

---

3. Spring에서 IoC 컨테이너의 역할

역할	설명
객체 생성	개발자가 아닌 컨테이너가 Bean 생성
의존성 주입 (Dependency Injection)	Bean 간 연결 관계를 자동으로 설정
Bean 스코프 관리	Singleton, Prototype 등 빈 생명주기 제어
초기화/소멸 처리	`@PostConstruct`, `@PreDestroy`
설정 파일 또는 어노테이션 기반 설정 지원	XML, JavaConfig, Annotation 모두 사용 가능

---

4. IoC 컨테이너의 종류 (Spring 기준)

컨테이너 이름	설명
`BeanFactory`	가장 기본적인 IoC 컨테이너 (지연 로딩)
`ApplicationContext`	확장된 컨테이너, 거의 항상 이걸 사용함
`WebApplicationContext`	Spring MVC 웹 전용 컨테이너
`AnnotationConfigApplicationContext`	자바 기반 설정을 사용하는 경우
`ClassPathXmlApplicationContext`	XML 기반 설정 사용 시

---

5. 동작 원리

1. **설정 파일 로드**
   XML, JavaConfig, `@ComponentScan`, `@Bean`, `@Configuration` 등
2. **빈 정의 분석 (BeanDefinition)**
   클래스, 생성자, 필드, 메서드 등 메타 정보 수집
3. **객체 생성 (Instantiation)**
   Reflection 또는 생성자 기반으로 객체 생성
4. **의존성 주입 (Dependency Injection)**
   `@Autowired`, `@Inject`, `@Resource` 등의 정보로 다른 빈을 주입
5. **초기화 (Initialization)**
   `@PostConstruct` 또는 InitializingBean
6. **사용 (Ready to use)**
   빈은 컨테이너에서 필요 시마다 사용됨
7. **소멸 처리 (Destroy)**
   `@PreDestroy` 또는 DisposableBean 등

---

6. 실전 코드 예시

자바 기반 설정 + 어노테이션 기반 DI

```
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
}

@Component
public class OrderService {
    @Autowired
    private PaymentService paymentService;
}

@Component
public class CreditCardPaymentService implements PaymentService {
    public void pay() {
        System.out.println("Pay with Credit Card");
    }
}

public interface PaymentService {
    void pay();
}
```

컨테이너 구동

```
ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
OrderService order = context.getBean(OrderService.class);
order.process();
```

---

7. Bean 관리 관련 어노테이션

어노테이션	설명
`@Component`	일반 컴포넌트 클래스
`@Service`	서비스 레이어용
`@Repository`	DAO 레이어용
`@Controller`	웹 컨트롤러
`@Autowired`	의존성 주입 요청
`@Bean`	수동 등록 빈
`@Configuration`	자바 설정 클래스
`@Scope`	빈의 범위 지정 (`singleton`, `prototype`)

어노테이션	설명
`@Qualifier`	이름으로 빈 지정

---

8. 싱글톤 vs 프로토타입

스코프	특징
`singleton`	컨테이너 내에서 오직 하나의 인스턴스만 생성 (기본값)
`prototype`	요청마다 새로운 인스턴스 생성
`request`	HTTP 요청마다 새로운 빈 생성 (웹)
`session`	세션별로 하나의 빈 생성 (웹)

---

9. IoC와 DI의 관계

개념	설명
IoC	객체 생성/관리를 외부로 위임하는 전체 개념 (제어의 역전)
DI	IoC의 구현 방식 중 하나. 객체에 필요한 의존성을 주입하는 방법

즉, **DI는 IoC를 실현하는 방법** 중 하나다. Spring은 DI를 통해 IoC를 구현한다.

---

10. 오해와 정리

잘못된 이해	올바른 설명
"Spring은 DI만 제공하는 프레임워크다"	Spring은 IoC, AOP, Data, Security 등 다양한 계층의 기능을 포함한다
"IoC는 DI와 동일하다"	DI는 IoC의 구현 방식 중 하나일 뿐
"IoC는 XML 기반으로만 동작한다"	어노테이션, JavaConfig 등 다양한 방식 존재
"빈은 new로 생성되지 않으면 쓸 수 없다"	컨테이너가 관리하는 객체만 `@Autowired`로 주입 가능

---

11. 요약

항목	설명
개념	객체의 생성과 연결을 외부 컨테이너가 제어하는 구조
구현 주체	Spring IoC 컨테이너 (`ApplicationContext`)
실현 방법	주로 DI (의존성 주입)

항목	설명
주요 기능	객체 생성, DI, 생명주기 관리, 스코프 관리
사용 방식	어노테이션 + 설정 파일 (JavaConfig, XML)
핵심 효과	결합도 감소, 유지보수 향상, 테스트 용이성 증가

BeanFactory vs ApplicationContext

다음은 **Spring의 IoC 컨테이너 핵심 인터페이스인 `BeanFactory` 와 `ApplicationContext`** 의 차이를 개념, 기능, 내부 구조, 사용 사례, 코드 예제, 실전에서의 선택 기준까지 포함하여 가장 자세히 정리한 설명이다.

항목	`BeanFactory`	`ApplicationContext`
역할	IoC 컨테이너의 **기본 기능만 제공**	IoC + 다양한 **부가 기능 제공**
포함 관계	최소 컨테이너, ApplicationContext의 상위 인터페이스	`BeanFactory` 를 확장한 하위 구현체
사용 목적	경량화, 리소스 절약 필요할 때	거의 모든 실무 Spring 애플리케이션에서 사용
빈 로딩 시점	**지연 로딩(Lazy Loading)**	**즉시 로딩(Eager Loading)**
국제화 (i18n)	❌ 없음	✅ `MessageSource` 내장
이벤트 처리	❌ 없음	✅ `ApplicationEventPublisher` 내장
AOP 지원	제한적	✅ 자동 프록시 생성 지원
환경 변수 처리	❌ 없음	✅ `Environment` 및 `PropertySource` 지원
통합 빈 생명주기 관리	제한적	✅ `Lifecycle`, `ApplicationListener` 등 완전 지원
주로 사용하는 구현 클래스	없음 (직접 사용 잘 안 함)	`ClassPathXmlApplicationContext`, `AnnotationConfigApplicationContext`, `WebApplicationContext` 등
실무 사용 빈도	낮음 (테스트나 성능 특수 목적용)	매우 높음 (99% 이상 실제 프로젝트에서 사용)

---

1. `BeanFactory`란?

정의

- Spring IoC의 **최소 기능 인터페이스**
- `getBean()` 메서드를 통해 객체를 가져오는 컨테이너
- XML 또는 자바 설정을 읽어서 Bean 정보를 담고 있음

특징

- **빈을 필요할 때만 초기화 (lazy loading)**
- 이벤트, 메시지 처리 등의 부가 기능 없음
- 초경량 환경 또는 테스트용으로 적합

예시

```
BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));
MyBean obj = (MyBean) factory.getBean("myBean");
```

> `XmlBeanFactory`는 Spring 3.1부터 deprecated 됨 → 실무 사용 X

---

2. `ApplicationContext`란?

정의

- `BeanFactory`를 확장하여 **엔터프라이즈급 기능**을 추가한 IoC 컨테이너
- Spring 개발 시 **기본적으로 사용하는 컨테이너**

특징

- **빈을 컨테이너 초기화 시 모두 로딩 (eager loading)**
- 메시지 리소스 처리, 이벤트 발행, 환경 설정, 빈 후처리 등 다양한 기능 포함
- 웹, 배치, 데스크탑 등 모든 환경에서 사용 가능

주요 구현체

클래스	설명
`AnnotationConfigApplicationContext`	자바 기반 설정용
`ClassPathXmlApplicationContext`	XML 설정용
`WebApplicationContext`	Spring MVC 컨텍스트 전용
`GenericApplicationContext`	유연한 일반 컨텍스트

---

3. 핵심 메서드 비교

메서드	BeanFactory	ApplicationContext
`getBean(String)`	✅	✅
`containsBean(String)`	✅	✅
`getBeanDefinitionNames()`	✅	✅
`getMessage()`	❌	✅ (국제화 메시지 처리)
`publishEvent()`	❌	✅ (ApplicationEvent 처리)
`getEnvironment()`	❌	✅ (환경/프로파일 분리)

---

4. 코드 예제 비교

4.1 BeanFactory 사용 예 (비추천)

```
Resource resource = new ClassPathResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
UserService userService = factory.getBean(UserService.class);
```

4.2 ApplicationContext 사용 예 (표준)

```
ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
UserService userService = context.getBean(UserService.class);
```

```
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {}
```

---

5. 빈 로딩 시점 비교

동작	BeanFactory	ApplicationContext
`getBean()` 전까지 생성 안 됨	✅ (Lazy)	❌ (Eager)
애플리케이션 시작 시 검사	❌	✅

→ BeanFactory는 자원을 아낄 수 있지만, 오류를 **실행 시점**까지 발견하지 못함

---

6. 실전 사용 기준

상황	컨테이너 선택
스프링 부트 애플리케이션	ApplicationContext (자동)
일반 스프링 애플리케이션	ApplicationContext
매우 단순한, 성능 민감한 테스트	BeanFactory 가능 (특수 목적)
IoC 컨테이너 실습용	둘 다 가능하나 실무는 ApplicationContext 위주

---

7. 요약

항목	BeanFactory	ApplicationContext
정의	최소 IoC 컨테이너 인터페이스	Spring의 실무용 IoC 컨테이너 구현체
로딩 방식	지연 로딩	즉시 로딩
주요 기능	`getBean()` 등 최소 기능	이벤트, 메시지, 환경, 국제화 등 확장 포함
사용 용도	테스트, 경량 애플리케이션	모든 Spring 애플리케이션 (표준)
실무 사용	거의 없음	항상 사용됨

Bean 등록 방식

- # XML 기반 등록

  다음은 **Spring에서 XML 기반으로 Bean을 등록하는 방법**에 대해 가장 깊이 있게 설명한 내용이다. XML 설정 방식은 현대 Spring Boot에서는 주로 Java Config로 대체되었지만, **레거시 프로젝트 유지보수**, **기본 IoC 개념 이해**, **테스트 환경 구성**, 또는 **타 시스템 연동**에서는 여전히 유효하게 쓰이고 있다.

  ## 1. 개요

  Spring은 초창기부터 **XML 파일을 이용하여 IoC 컨테이너의 Bean 설정**을 지원해왔다.
  Java 코드와 분리된 설정 파일로 객체 생성, 의존성 주입(DI), 속성 설정, 라이프사이클 관리 등을 정의할 수 있다.

  ```
  <!-- beans.xml -->
  <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
           http://www.springframework.org/schema/beans/spring-beans.xsd">

      <bean id="userService" class="com.example.UserService"/>
  </beans>
  ```

  ---

2. XML 기반 Bean 등록 기본 형식

```
<bean id="bean이름" class="패키지.클래스이름" scope="singleton|prototype">
    <property name="필드명" value="값 또는 ref"/>
</bean>
```

속성	설명
`id`	컨테이너에서 사용할 Bean 이름
`class`	빈으로 등록할 클래스의 전체 경로
`scope`	Bean의 스코프 (`singleton`, `prototype`, `request`, `session` 등)
`<property>`	setter를 통한 의존성 주입 또는 값 주입

---

3. 생성자 기반 의존성 주입

```
<bean id="orderService" class="com.example.OrderService">
    <constructor-arg ref="userService"/>
</bean>

<bean id="userService" class="com.example.UserService"/>
```

→ `OrderService(UserService userService)` 생성자에 의존성 주입

---

4. Setter 기반 의존성 주입

```
<bean id="orderService" class="com.example.OrderService">
    <property name="userService" ref="userService"/>
</bean>
```

→ `orderService.setUserService(...)` 방식으로 주입됨

---

5. 값 주입 (`value`)

```
<bean id="config" class="com.example.AppConfig">
    <property name="appName" value="MyApp"/>
    <property name="port" value="8080"/>
</bean>
```

- 내부적으로 Spring은 `setAppName("MyApp")`, `setPort(8080)` 호출

---

6. 컬렉션 주입 (`list`, `map`, `set`, `props`)

```
<bean id="dataService" class="com.example.DataService">
    <property name="servers">
        <list>
            <value>server1</value>
            <value>server2</value>
        </list>
    </property>
</bean>
```

또는

```
<property name="settings">
    <map>
        <entry key="timeout" value="30"/>
        <entry key="mode" value="fast"/>
    </map>
</property>
```

---

7. 빈 참조 (`ref`)

```
<bean id="dao" class="com.example.MyDao"/>
<bean id="service" class="com.example.MyService">
    <property name="dao" ref="dao"/>
</bean>
```

---

8. 빈 라이프사이클

```
<bean id="networkClient" class="com.example.NetworkClient"
      init-method="connect" destroy-method="disconnect"/>
```

- `connect()` 는 초기화 시 호출됨
- `disconnect()` 는 컨테이너 종료 시 호출됨

---

9. 애노테이션과 XML 병행 사용

예시: `@ComponentScan` 없이 XML로 스캔 지정

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

> XML에서도 `context` 네임스페이스 추가 필요:

```
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="... http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd"
```

---

10. XML 파일을 사용하는 IoC 컨테이너 코드

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("beans.xml");

MyService service = context.getBean("myService", MyService.class);
```

또는

```
BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));
```

> `XmlBeanFactory`는 Spring 3.1부터 deprecated 되었음 → `ApplicationContext` 사용 권장

---

11. XML 설정 파일 위치

- 기본 위치: `src/main/resources/beans.xml`
- 패키지 구조 가능: `com/example/config/mybeans.xml`
- 상대 경로 지정 가능: `new ClassPathXmlApplicationContext("config/mybeans.xml")`

---

12. 실무에서 XML 설정이 쓰이는 경우

상황	이유
레거시 시스템 유지보수	이전 버전의 Spring은 XML만 지원
비표준 외부 시스템 연동	스프링 애노테이션 없이 구성
모듈화된 설정 파일 관리 필요	Spring Boot가 아닌 프로젝트에서 일부 사용
설정과 구현의 완전한 분리 필요	코드와 설정을 완전히 분리하고 싶은 경우

---

13. Spring Boot에서 XML 사용

Spring Boot는 기본적으로 **Java Config + 애노테이션 기반 설정**을 사용하지만, XML도 동시에 병용 가능하다.

```
@ImportResource("classpath:beans.xml")
@SpringBootApplication
public class Application { ... }
```

---

14. 요약

항목	설명
장점	명확한 설정 분리, 코드 없는 구조화
단점	XML 문법 불편, 가독성 낮음, 중복
현재 사용 추세	Spring Boot 이후 JavaConfig 위주
레거시 유지 필요 여부	일부 환경에서는 여전히 유효

- # 자바 기반 등록 (`@Bean`, `@Configuration`)

다음은 **자바 기반 Bean 등록 방식**에 대한 가장 깊이 있는 설명이다.
`@Bean`과 `@Configuration`의 역할과 동작 방식, 내부 구조, 생성자 및 의존성 주입 패턴, XML 방식과의 비교, 실무 예시까지 포함하여 **Spring Core IoC 컨테이너의 핵심 개념**을 이해하는 데 필요한 모든 내용을 정리했다.

---

1. 개요

Spring은 XML 설정 없이도 Java 클래스를 이용해 **IoC 컨테이너에 Bean을 등록**할 수 있도록 `@Bean`, `@Configuration` 등의 어노테이션 기반 **자바 구성 방식(JavaConfig)**을 제공한다.

- **자바 코드로 설정을 선언**하는 방식 → 타입 안전성, IDE 지원, 리팩터링 용이
- **Spring Boot에서 기본 방식**

---

2. 핵심 어노테이션

어노테이션	역할
`@Configuration`	설정 클래스임을 명시 (Spring이 스캔 후 Bean 정의로 인식)
`@Bean`	반환 객체를 Spring IoC 컨테이너에 Bean으로 등록
`@Import`	다른 설정 클래스를 함께 로드
`@ComponentScan`	자동 컴포넌트 스캔과 함께 사용 가능

---

3. 기본 예제

```
@Configuration
public class AppConfig {

    @Bean
    public UserService userService() {
        return new UserService(userRepository());
    }

    @Bean
    public UserRepository userRepository() {
        return new InMemoryUserRepository();
    }
}
```

→ `userService()` 메서드의 반환 객체가 Spring Bean으로 등록됨
→ 다른 Bean(`userRepository`)도 메서드 호출로 주입 가능

---

4. 등록 방식의 특징

항목	설명
타입 안전성	컴파일 시점에 타입 검사 가능
리팩터링 용이	IDE에서 클래스 이름, 메서드 추적 가능
순서 및 의존성 관리	메서드 호출 순서, 명시적 주입으로 제어 가능
재사용성	설정 클래스를 `@Import`로 구성 가능
테스트 친화성	설정 클래스를 독립적으로 테스트 가능

---

5. 내부 동작 원리

`@Configuration`이 붙은 클래스는 CGLIB 프록시 객체로 변환된다.

- `userService()` 내부에서 `userRepository()`를 직접 호출해도,
  실제로는 Spring 컨테이너에 등록된 싱글톤 Bean을 반환함
- 즉, **메서드 간 호출도 컨테이너 관리 하에 있다**는 것

예:

```
@Bean
public A a() {
    return new A(b());  // 단순 호출처럼 보이지만 실제로는 싱글톤 B Bean 반환
}
```

---

6. @Bean 메서드 간 의존성 처리 방식

생성자 주입 방식

```
@Bean
public OrderService orderService() {
    return new OrderService(paymentService());
}
```

메서드 주입 방식

```
@Bean
public OrderService orderService(PaymentService paymentService) {
    return new OrderService(paymentService);
}
```

→ 둘 다 동일한 결과, 후자는 더 깔끔하고 테스트하기 좋음

---

7. 빈 이름과 이름 변경

```
@Bean(name = "customService")
public MyService myService() {
    return new MyService();
}
```

- 기본 이름: 메서드 이름 (`myService`)
- 커스텀 이름: `@Bean(name="...")` 또는 배열 가능

---

8. `@Component` 방식과 차이

항목	`@Bean` / `@Configuration`	`@Component`, `@Service` 등
위치	설정 클래스 내 메서드	클래스에 직접 어노테이션 부착
제어 수준	수동 등록 (명시적)	자동 등록 (스캔 기반)
주입 방식	메서드 반환값	클래스 전체
동적 조건 구성	용이 (`@Conditional`, 분기 가능)	어려움

---

9. 외부 라이브러리 Bean 등록 시 유용

예: 외부 클래스 `ObjectMapper`, `RestTemplate` 등은 어노테이션이 불가능하므로 `@Bean` 방식 사용

```java
@Configuration
public class InfraConfig {

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

---

10. 고급 기능 예시

10.1 `@Conditional`로 조건부 Bean 등록

```java
@Bean
@ConditionalOnProperty(name = "feature.payment", havingValue = "true")
public PaymentService paymentService() {
    return new RealPaymentService();
}
```

10.2 프로파일 기반 Bean

```java
@Bean
@Profile("dev")
public DataSource devDataSource() { ... }

@Bean
@Profile("prod")
public DataSource prodDataSource() { ... }
```

---

11. XML 기반 설정과의 비교

항목	Java Config (`@Bean`)	XML 설정
타입 안전성	✅	❌ (문자열 기반)
IDE 자동완성	✅	❌
설정과 코드 분리	❌ (같은 자바파일에 있음)	✅
테스트 가능성	✅ (POJO 방식으로 테스트 가능)	❌ (설정 자체 테스트 어려움)

---

12. 실전 프로젝트 예: 설정 분리

```java
@Configuration
public class AppConfig {
    @Bean
    public UserService userService(UserRepository repo) {
```

```
        return new UserService(repo);
    }
}

@Configuration
public class InfraConfig {
    @Bean
    public DataSource dataSource() {
        return DataSourceBuilder.create()
            .url("jdbc:mysql://localhost:3306/test")
            .username("root")
            .password("1234")
            .build();
    }
}
```

→ `@Import(AppConfig.class)` 또는 `@ComponentScan`과 함께 사용 가능

---

13. 요약

항목	설명
정의	자바 코드로 Bean을 등록하는 방식
대표 어노테이션	`@Configuration`, `@Bean`
장점	타입 안전, IDE 친화적, 리팩터링 편리
사용 용도	복잡한 빈 설정, 외부 객체 등록, 프로파일 분기
추천 여부	✅ Spring Boot 이후 기본 방식

- # 컴포넌트 스캔 (`@Component`, `@Service`, `@Repository`, `@Controller`)

다음은 **Spring의 컴포넌트 스캔(Component Scanning)** 개념과, 관련 어노테이션인 `@Component`, `@Service`, `@Repository`, `@Controller`의 역할, 차이점, 스캔 범위 지정, 내부 동작 원리, 실무 사용 전략까지 포함한 **가장 자세한 정리**다.

---

1. 컴포넌트 스캔(Component Scan)이란?

컴포넌트 스캔은 Spring이 애플리케이션 실행 시 **클래스를 자동으로 스캔하고**, 특정 어노테이션이 붙은 클래스를 **Bean으로 등록하는 메커니즘**이다.

- 개발자가 `@Bean` **없이도 클래스를 자동 등록**할 수 있게 함
- 주로 **애노테이션 기반 DI** 구성에서 사용
- `@Component` 계열 어노테이션을 탐지함

---

2. 관련 어노테이션 종류

어노테이션	역할
`@Component`	모든 스프링 컴포넌트의 **기본형**, 일반적인 Bean 등록용
`@Service`	**서비스 계층** 용도, 비즈니스 로직 담당 클래스에 사용
`@Repository`	**데이터 접근 계층**(DAO)에 사용, 예외 변환(AOP) 지원
`@Controller`	**웹 컨트롤러**에 사용, Spring MVC에서 요청 처리 담당

> 모두 `@Component`를 메타 어노테이션으로 포함하고 있으며, **기능상 동일**하지만 **의도 구분** 및 **AOP 지원 차이**가 있음

---

3. 동작 방식

1. `@ComponentScan` 또는 `@SpringBootApplication` 어노테이션이 선언된 클래스에서 시작
2. 지정한 패키지(혹은 기본 동일 패키지 이하)를 순회
3. `@Component` 계열 어노테이션이 붙은 클래스를 찾음
4. Spring IoC 컨테이너가 Bean으로 등록 (`ApplicationContext`에 보관)

---

4. 기본 사용 예

```
@SpringBootApplication  // 내부적으로 @ComponentScan 포함
public class MyApp {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}
```

클래스 예:

```
@Component
public class MyComponent {}

@Service
public class MyService {}

@Repository
public class MyRepository {}

@Controller
public class MyController {}
```

---

5. 패키지 스캔 범위 지정

명시적 사용 예 (`@ComponentScan`):

```
@Configuration
@ComponentScan(basePackages = {"com.example.service", "com.example.dao"})
public class AppConfig {}
```

→ 지정한 패키지 이하의 클래스만 스캔됨

Spring Boot 기본 규칙:

- `@SpringBootApplication`이 붙은 클래스의 **하위 패키지부터 자동 스캔**
- 따라서 최상위 루트 패키지에 놓는 것이 권장

```
// com.example
@SpringBootApplication
public class MyApp {}
```

---

6. 등록된 Bean 이름

기본적으로 **클래스 이름을 camelCase로 변환**하여 Bean 이름으로 등록

예:

```
@Component
public class UserService {}  // Bean 이름: "userService"
```

커스텀 이름 지정:

```
@Component("customName")
public class UserService {}
```

---

7. 계층별 역할 구분 (개념적)

어노테이션	사용 위치	목적
`@Component`	공통 유틸, 헬퍼 클래스	범용 빈 등록
`@Service`	서비스 계층	비즈니스 로직
`@Repository`	DAO, JPA 등	DB 연동 + 예외 변환
`@Controller`	웹 요청 처리 클래스	MVC의 Controller 처리 담당

> 기능상 차이는 없으나, **계층 구조를 명확하게 표현**하고, **AOP 타겟 설정에 활용**된다

---

8. 예외 변환 기능 (@Repository 전용)

```
@Repository
public class UserRepository {
    public void save() {
        throw new SQLException(); // 런타임 시 DataAccessException으로 자동 변환
    }
}
```

Spring은 `@Repository`에 AOP 프록시를 붙여 JDBC 예외를 Spring 예외로 변환해 준다.

---

9. AOP 및 트랜잭션과의 연계

- `@Service`: `@Transactional` 적용 대상이 되는 클래스의 대표적 위치
- `@Controller`: `@ExceptionHandler`, `@RequestMapping`, `@RestController` 등 MVC 확장 대상
- `@Repository`: Spring Data JPA, 예외 변환 포인트

---

10. 주의사항 및 팁

항목	주의점
패키지 스캔 누락	상위 클래스 패키지에서 스캔하지 않으면 등록 안 됨
다중 빈 충돌	동일 타입 클래스가 여러 개일 경우 `@Primary` 또는 `@Qualifier` 필요
동적 등록 필요 시	`@Bean` 방식으로 등록해야 함
Spring Boot Starter에서도 사용됨	예: `spring-boot-starter-web`이 `@Controller`, `@RestController`를 자동 스캔

---

11. 요약

항목	설명
컴포넌트 스캔	특정 패키지 이하의 클래스 중 지정 어노테이션이 붙은 객체를 자동 Bean으로 등록
핵심 어노테이션	`@Component`, `@Service`, `@Repository`, `@Controller`
등록 위치 규칙	`@SpringBootApplication` 또는 `@ComponentScan` 기준 패키지 이하
기능 차이	내부적으로는 동일, AOP와 역할 구분에서 차이 발생
실무 권장 사용 방식	역할에 따라 어노테이션 명확히 구분하여 사용

의존성 주입 방식

- # 생성자 주입

다음은 **Spring에서 생성자 주입(Constructor Injection)**에 대한 가장 자세한 설명이다.
기본 개념부터 필드 주입과의 비교, 장단점, 순환 참조 방지 효과, 테스트 코드 작성, 실무 패턴 등까지 **객체 설계의 핵심이 되는 의존성 주입 방식**을 이해할 수 있도록 정리했다.

생성자 주입이란?

1. 정의

생성자 주입은 **객체가 생성될 때 필요한 의존성(Bean)을 생성자의 매개변수로 주입받는 방식**이다.
Spring IoC 컨테이너가 생성자에 필요한 의존 객체를 자동으로 찾아서 주입해준다.

```
@Component
public class OrderService {

    private final PaymentService paymentService;

    @Autowired  // 생략 가능 (Spring 4.3+부터 단일 생성자일 경우)
    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

---

2. 주요 특징

항목	설명
불변성 확보	`final` 필드로 주입 받기 적합 → 값 변경 불가
객체 완전성 보장	생성 시점에 모든 의존성이 주입되어야 함
테스트 용이성	테스트 시 명시적 생성자 호출로 의존성 설정 가능
순환 참조 감지	Spring이 **빨리 감지하고 예외 발생시켜줌**
IDE 지원	생성자 인자 자동 생성, 리팩터링 쉬움

---

3. 예제 코드

3.1 일반적인 생성자 주입

```
@Service
public class OrderService {

    private final UserService userService;
```

```
    private final PaymentService paymentService;

    @Autowired
    public OrderService(UserService userService, PaymentService paymentService) {
        this.userService = userService;
        this.paymentService = paymentService;
    }
}
```

3.2 생성자가 하나라면 `@Autowired` 생략 가능

```
@Component
public class EmailService {
    private final MailSender mailSender;

    public EmailService(MailSender mailSender) {
        this.mailSender = mailSender;
    }
}
```

---

4. 생성자 주입 vs 필드 주입 vs 세터 주입

비교 항목	생성자 주입 ✅	필드 주입 ❌	세터 주입 ⚠️
불변성 확보	✅	❌ (값 변경 가능)	❌ (public setter 필요)
테스트 용이성	✅ (Mock 주입 가능)	❌ (리플렉션 필요)	⚠️ (setter 호출 필요)
순환 참조 감지	✅ 컴파일 타임 또는 즉시	❌ 런타임 오류로 감지됨	❌ 늦게 오류 발견 가능
IDE 리팩터링 지원	✅	❌	⚠️
추천 여부	⭐⭐⭐⭐⭐ 강력 추천	❌ 지양	⚠️ 조건부 사용 가능

---

5. 실무에서 사용하는 패턴

5.1 `@RequiredArgsConstructor` + `final` 조합 (Lombok)

```
@Service
@RequiredArgsConstructor
public class UserService {
    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
}
```

→ 생성자를 자동 생성해주고 Spring이 자동으로 주입
→ **가장 간결하면서도 안전한 생성자 주입 방식**

---

6. 순환 참조(Circular Dependency) 감지

Spring은 생성자 주입 시 **순환 참조가 감지되면 즉시 예외를 발생**시킨다.

예:

```
@Component
class A {
    public A(B b) {}
}
@Component
class B {
    public B(A a) {}
}
```

→ `UnsatisfiedDependencyException: circular reference` 발생

> 필드 주입은 지연 생성되므로 순환 참조 오류를 **늦게 발견하거나 실행 도중에 문제 발생**

---

7. 생성자 주입 시 단일/다중 생성자 처리

단일 생성자일 경우

- `@Autowired` 생략 가능 (Spring 4.3+)

다중 생성자일 경우

- 명확히 하나에 `@Autowired` 명시해야 함

```
@Component
public class NotificationService {

    @Autowired
    public NotificationService(SmsSender sender) { ... }

    public NotificationService(EmailSender sender) { ... } // 사용되지 않음
}
```

---

8. 테스트 코드 예시

```
@Test
void orderServiceTest() {
    UserService mockUserService = mock(UserService.class);
    PaymentService mockPayment = mock(PaymentService.class);
    OrderService service = new OrderService(mockUserService, mockPayment);

    // 테스트 가능
}
```

→ **생성자 주입은 테스트 시 mocking 용이**하고, 가짜 객체 주입 시 에러 발생 가능성 줄어듦

---

9. 요약

항목	설명
정의	객체 생성 시점에 의존성을 생성자로 주입
가장 이상적인 방식	✅ 불변성, 테스트성, 순환 감지 모두 우수
Lombok 패턴	`@RequiredArgsConstructor` + `final` 필드
Spring Boot 기본 스타일	생성자 주입 → 필드 주입 지양 권장
실무 적용 추천 수준	⭐ ⭐ ⭐ ⭐ ⭐ 매우 강력 추천

- # 세터 주입

다음은 **Spring에서의 세터 주입(Setter Injection)**에 대한 가장 자세하고 깊이 있는 설명이다.
기본 개념부터, 생성자 주입과의 차이점, 테스트와의 연계, 실무에서의 사용 위치, Lombok과의 병행 방식, 의존성 주입 순서까지 포함해 **Spring IoC 컨테이너의 DI 전략 중 하나**로서의 세터 주입을 완벽하게 이해할 수 있도록 정리했다.

세터 주입(Setter Injection)이란?

1. 정의

세터 주입이란, **Spring IoC 컨테이너가 빈 생성 후 `setXxx()` 메서드를 통해 의존 객체를 주입하는 방식**이다.

- 객체 생성 → Spring이 빈 초기화 과정에서 `@Autowired` 또는 XML `<property>` 등을 통해 **세터 메서드 호출**
- 주입 대상 필드는 `private`이어도 상관없으며, **세터만 public이면 가능**

---

2. 기본 예제

```
@Component
public class OrderService {

    private PaymentService paymentService;

    @Autowired
    public void setPaymentService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

→ Spring이 `OrderService`를 생성한 후, `setPaymentService()` 호출

---

3. 특징

항목	설명
주입 시점	**빈 생성 이후, 초기화 단계에 실행**
필수/선택 여부	선택적 의존성 주입 가능 (`required=false`)
테스트 용이성	Mock 객체 세터로 주입 가능
변경 가능성	`final` 불가 → 불변성 확보 어려움
순환 참조 문제	**회피 가능** (Spring은 지연 참조로 처리함)

---

4. `@Autowired(required = false)`

세터 주입은 **선택적 의존성**을 주입하는 데 적합하다.

```
@Autowired(required = false)
public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
}
```

→ `EmailService`가 없더라도 애플리케이션이 정상 실행됨
→ 생성자 주입에서는 이와 같은 "선택적 의존성" 처리 어려움

---

5. XML 설정 예 (전통적인 방식)

```
<bean id="orderService" class="com.example.OrderService">
    <property name="paymentService" ref="paymentService"/>
</bean>
```

→ `setPaymentService(...)`가 호출됨

---

6. 세터 주입 vs 생성자 주입

항목	세터 주입	생성자 주입
주입 방식	`setXxx()` 메서드를 통해 주입	생성자를 통해 주입
불변성	❌ 변경 가능 (final 불가)	✅ 객체 불변성 확보 가능
선택적 주입	✅ `required=false`로 가능	❌ 모든 인자를 필수로 요구
테스트 용이성	⚠️ setter를 수동 호출해야 함	✅ 생성자 인자로 전달하면 됨
순환 참조 대응	✅ 상대적으로 유연함	❌ 즉시 참조 필요 시 예외 발생

항목	세터 주입	생성자 주입
코드 간결성	❌ setter가 여러 개 필요할 수 있음	✅ 간결한 선언 가능
IDE 리팩터링, 추적성	❌ getter/setter 오염 우려	✅ 명확한 의존성 표시 가능

---

7. 실무에서의 사용 위치

상황	세터 주입 적합 여부
필수 의존성이 아님	✅ 사용 권장
컴포넌트 초기화 순서 필요	✅ 필요 시 설정
Lombok `@Setter`로 주입하고자 할 때	⚠️ 가능하지만 추천되지 않음
테스트 중 Mock 객체 주입	✅ 수동 주입 가능
외부 설정값(`@Value`) 주입	✅ Setter 방식 가능

---

8. Lombok으로 세터 주입 처리 (⚠️ 주의)

```
@Component
@Getter
@Setter  // 세터 주입 가능하나 권장되지 않음
public class AlertService {
    private EmailService emailService;
}
```

→ **Lombok으로 세터를 자동 생성하면 public이므로 주입 가능**
→ 그러나 불변성을 해치므로 **@RequiredArgsConstructor 방식이 더 권장됨**

---

9. 세터 주입의 순서 (Spring Lifecycle 중)

1. 객체 인스턴스 생성 (생성자 호출)
2. 의존성 주입 (세터 호출)
3. `@PostConstruct` 메서드 실행
4. ApplicationContext에 등록 완료

→ 즉, 세터는 **생성 이후에 호출되므로 순서 제어가 가능**

---

10. 요약

항목	설명
정의	Spring이 객체 생성 후 `setXxx()` 메서드를 통해 주입
특징	선택적 주입 가능, 순환 참조 회피에 유리
단점	객체 불완전 상태로 생성될 수 있음, 필수 의존성 보장 불가
추천 여부	⚠️ 조건부로 사용 (선택적, 외부 설정값 등)
실무 사용 권장 상황	일부 보조 빈, 설정 주입, 테스트 환경

---

11. 결론

- **Spring의 기본 DI 방식은 생성자 주입**을 우선 사용하고,
- **세터 주입은 선택적/후처리/환경 설정 목적일 때에만 제한적으로 사용**하는 것이 가장 권장되는 전략이다.

• 필드 주입

다음은 **Spring의 필드 주입(Field Injection)**에 대한 가장 깊이 있는 설명이다.
기본 개념, 작동 원리, 생성자/세터 주입과의 비교, 실무에서의 주의점, 장단점, 테스트 시의 문제점, 그리고 불가피한 사용 시 대안까지 포함해 **필드 주입을 언제 어떻게 써야 하는지 명확히 정리**하였다.

필드 주입(Field Injection)이란?

1. 정의

필드 주입은 **클래스의 필드(멤버 변수)에 직접 `@Autowired` 어노테이션을 붙여서 의존성(Bean)을 주입**하는 방식이다.
Spring이 **리플렉션(Reflection)**을 사용해 private 필드에 값을 주입한다.

```
@Component
public class OrderService {

    @Autowired
    private PaymentService paymentService;
}
```

→ `OrderService` 인스턴스를 생성한 후, `paymentService` 필드에 값을 **직접 삽입(setXxx 호출 없이)**

---

2. 특징 요약

항목	설명
접근 방식	**필드에 직접 주입** (`private`, `protected`, `public` 모두 가능)
setter/constructor 필요	❌ 필요 없음

항목	설명
코드 간결성	✅ 가장 짧고 간단
테스트 용이성	❌ 매우 떨어짐 (리플렉션 필요)
객체 불변성	❌ 보장 안 됨
의존성 명시성	❌ IDE에서 추적 어려움
실무 권장도	❌ 거의 모든 경우 **비권장**

---

3. 예제 코드

```
@Component
public class EmailService {

    @Autowired
    private MailSender mailSender;

    public void send(String to, String content) {
        mailSender.sendMail(to, content);
    }
}
```

---

4. 동작 원리

- Spring은 해당 클래스가 Bean으로 등록된 이후,
  `@Autowired`가 붙은 필드를 **리플렉션을 통해 강제로 접근 → 주입**
- 필드가 `private`이라도 접근 가능
- setter나 생성자를 쓰지 않기 때문에 코드가 짧지만, **의존성이 외부에서 보이지 않음**

---

5. 장점 vs 단점

장점	단점 (치명적)
코드가 간결하다	객체 생성 시점에 의존성이 보장되지 않음
setter/constructor 없이도 주입 가능	테스트 시 Mock 객체를 직접 주입할 수 없음
자바 코드가 짧아 보인다	불변성, 명확성, 추적성 모두 부족
	리플렉션 사용 → 성능 및 안전성 문제 가능

---

6. 필드 주입 vs 생성자/세터 주입 비교

항목	필드 주입 ❌	세터 주입 ⚠️	생성자 주입 ✅
코드 길이	짧다	중간	길어도 명확함
주입 시점	객체 생성 후	객체 생성 후	객체 생성 시점
테스트 용이성	❌ 불편	⚠️ setter로 주입 가능	✅ 생성자로 명시 가능
순환 참조 감지	늦게 오류 발생 가능	중간	빨리 오류 감지 가능
명시적 의존성 표현	❌ 불명확	⚠️ setter로 추적 가능	✅ IDE/리팩터링 편함
불변성	❌ 불가능	❌	✅ `final` 가능

---

7. 테스트 문제점

- 필드에 직접 접근해야 하기 때문에 **Mockito나 JUnit에서 Mock 객체를 강제로 주입해야 함**

```
ReflectionTestUtils.setField(service, "emailSender", mockEmailSender);
```

→ 코드 가독성과 유지보수성, 테스트 안정성 **모두 나빠짐**

---

8. Spring 공식 권장 방식

> Spring 팀은 공식적으로 **생성자 주입을 우선 권장**하며,
> **필드 주입은 오직 "불가피한 경우"에만 제한적으로 사용할 것**을 제안한다.

---

9. 필드 주입이 제한적으로 허용되는 상황

상황	설명
설정 클래스 (`@Configuration`)	순환 참조를 회피하기 위해 제한적으로 사용 가능
외부 라이브러리 클래스 Wrapping	외부 클래스를 감싸면서 DI 필요할 때 간단히 주입
테스트 전용 코드	간결성을 위해 제한적으로 허용되기도 함

---

10. 대안: 생성자 주입으로 변경하는 예

Before: 필드 주입

```
@Component
public class MyService {
    @Autowired
    private MyRepository repository;
```

```
}
```

After: 생성자 주입

```
@Component
@RequiredArgsConstructor
public class MyService {
    private final MyRepository repository;
}
```

→ 명시적이고 테스트 가능한 구조로 변경

---

11. 요약

항목	설명
정의	`@Autowired`를 필드에 붙여 직접 주입
장점	코드가 간결함
단점	테스트 어려움, 불변성 없음, 의존성 명시성 부족
권장 여부	❌ 거의 모든 상황에서 지양됨
대안	`@RequiredArgsConstructor` + `final` 필드 (생성자 주입 방식)

---

12. 결론

- 필드 주입은 간편하지만, **유지보수성과 확장성을 심각하게 해친다**
- 실무에서는 **명확한 의존성 구조와 테스트 가능한 코드**를 위해 반드시 **생성자 주입을 사용할 것**

빈 스코프

- # singleton, prototype, request, session, application

Bean의 **Scope(범위)**란, Spring IoC 컨테이너가 **빈을 생성하고 유지하는 생명주기와 범위**를 의미합니다.
스프링에서 지원하는 기본 스코프는 아래와 같습니다.

---

1. `singleton` (기본값)

✅ 정의

- Spring IoC 컨테이너에 **단 하나의 인스턴스만 생성**하여 공유
- **모든 요청마다 같은 객체를 주입**받음

✅ 특징

- 가장 많이 사용됨
- 메모리 효율 좋음
- 공유 상태 주의 (동기화 문제 발생 가능)

✅ 사용 예

```
@Component
@Scope("singleton") // 생략 가능
public class UserService { }
```

✅ 라이프사이클

- 컨테이너 시작 시 생성 → 컨테이너 종료 시까지 유지

---

2. `prototype`

🔁 정의

- 요청할 때마다 **항상 새로운 인스턴스** 생성
- DI 받을 때마다 다른 객체가 주입됨

✅ 특징

- 상태를 가진 객체, 쓰레드마다 다른 인스턴스를 원할 때 사용
- 컨테이너는 **객체 생성까지만** 책임지고, 소멸은 **직접 관리**

✅ 사용 예

```
@Component
@Scope("prototype")
public class ChatMessage { }
```

⚠️ 주의

- `@PreDestroy` 등 라이프사이클 콜백이 **호출되지 않음**
- `ApplicationContext.getBean()` 호출마다 새 객체 반환

---

3. `request` (웹 전용)

🌐 정의

- **HTTP 요청 단위로 하나의 Bean 인스턴스**가 생성됨
- 요청마다 새로 생성되며, 응답 후 제거됨

✅ 사용 예

```
@Component
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode =
ScopedProxyMode.TARGET_CLASS)
public class RequestScopedBean { }
```

> proxyMode는 스프링 MVC 컨트롤러 등의 싱글톤 빈에서 주입되도록 **프록시 객체로 감쌈**

✅ 라이프사이클

- HTTP 요청이 시작될 때 생성 → 요청 완료 후 제거

---

4. `session` (웹 전용)

👤 정의

- **HTTP 세션 하나당 하나의 Bean 인스턴스** 생성
- 사용자 로그인 상태 관리 등에 활용 가능

✅ 사용 예

```
@Component
@Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
public class SessionScopedBean { }
```

✅ 라이프사이클

- HTTP 세션 시작 시 생성 → 세션 만료 시 제거

---

5. `application` (웹 전용)

🌍 정의

- **ServletContext(application scope)** 수준의 범위를 갖는 Bean
- 애플리케이션 단위로 단 하나만 존재 (웹 어플리케이션 전체에서 공유)

✅ 사용 예

```
@Component
@Scope(value = WebApplicationContext.SCOPE_APPLICATION)
public class ApplicationScopedBean { }
```

✅ 라이프사이클

- 웹 애플리케이션 시작 시 생성 → 종료 시까지 유지

---

6. 요약 비교표

스코프	생성 시점	소멸 시점	대상 환경	특징
`singleton`	컨테이너 초기화 시	컨테이너 종료 시	전체 공통	기본값, 공유됨
`prototype`	요청 시	개발자가 직접 관리	전체 공통	매번 새 인스턴스
`request`	HTTP 요청 시	요청 종료 시	웹 전용	요청별 인스턴스
`session`	세션 시작 시	세션 만료 시	웹 전용	사용자 단위 관리
`application`	서블릿 컨텍스트 시작 시	웹 앱 종료 시	웹 전용	애플리케이션 전체 공유

---

7. 실무 적용 전략

목적	추천 Scope
대부분의 서비스/DAO/비즈니스 로직	`singleton` (기본)
사용자 요청별 상태 정보 저장	`request`
로그인 정보 또는 사용자 세션 관리	`session`
반복 계산용 상태 캐싱	`application`
쓰레드 또는 사용자 고유 객체	`prototype`

---

8. 스코프 충돌 해결: 프록시 사용

`singleton` Bean 안에 `request`, `session`, `prototype` Bean을 주입할 때는 **프록시(proxy)**가 필요하다:

```
@Scope(value = "request", proxyMode = ScopedProxyMode.TARGET_CLASS)
```

→ Spring이 **지연 주입과 프록시 객체를 자동 생성**하여 문제 해결

---

9. 코드 예시: Controller에서 request-scope Bean 사용

```
@RestController
public class MyController {

    private final RequestScopedBean bean;

```

```
    public MyController(RequestScopedBean bean) {
        this.bean = bean;
    }

    @GetMapping("/")
    public String hello() {
        return bean.toString(); // 요청마다 다른 객체
    }
}
```

→ `RequestScopedBean`은 매 요청마다 새로운 객체이지만, `MyController`는 singleton이므로 **프록시로 감싸져 주입됨**

---

10. 결론 요약

항목	설명
기본 스코프	`singleton`
상태 객체, 매 요청 객체	`prototype`, `request`
사용자 단위 데이터 관리	`session`
전체 애플리케이션 캐시	`application`
주의 사항	프로토타입은 소멸 관리 직접 필요, 프록시 필요 상황 주의

빈 라이프사이클

- # 초기화, 소멸

1. 개요

Spring IoC 컨테이너는 Bean을 생성한 후 다음과 같은 과정을 통해 **초기화 및 소멸 처리를 자동으로 수행**합니다:

```
객체 생성 → 의존성 주입 → 초기화 → 사용 → 소멸
```

---

2. 초기화(Initialization)

Bean이 컨테이너에 의해 생성되고 의존성 주입까지 완료된 후 실행되는 **후처리 작업**입니다.
예: 데이터 로딩, 네트워크 연결, 리소스 준비 등

---

3. 소멸(Destroy)

Spring 컨테이너가 종료될 때, 또는 스코프에 따라 Bean이 제거될 때 실행되는 **정리 작업**입니다.
예: 파일 닫기, DB 연결 종료, 버퍼 해제 등

---

4. 초기화/소멸 방식 3가지

방식	초기화	소멸
1. 어노테이션 기반	`@PostConstruct`	`@PreDestroy`
2. 인터페이스 구현	`InitializingBean`	`DisposableBean`
3. XML or Java 설정 지정	`init-method`	`destroy-method`

---

5. 어노테이션 방식 (표준, 실무 권장 ✅)

```
@Component
public class NetworkClient {

    @PostConstruct
    public void init() {
        System.out.println("초기화 작업: 서버 연결 시작");
    }

    @PreDestroy
    public void close() {
        System.out.println("소멸 작업: 서버 연결 종료");
    }
}
```

- `@PostConstruct`: Bean 생성 및 DI 완료 직후 실행
- `@PreDestroy`: 컨테이너 종료 직전에 실행
- **JSR-250 표준 어노테이션**
- `@Configuration`, `@Component`, `@Bean` 모두 가능

---

6. 인터페이스 방식

```
@Component
public class CacheManager implements InitializingBean, DisposableBean {

    @Override
    public void afterPropertiesSet() {
        System.out.println("초기화 로직 실행");
    }

    @Override
    public void destroy() {
        System.out.println("종료 로직 실행");
    }
}
```

인터페이스	역할
`InitializingBean`	초기화 후 `afterPropertiesSet()` 실행
`DisposableBean`	소멸 시 `destroy()` 실행

> 단점: Spring 전용 코드에 의존적 → **테스트, 재사용성 낮음**
> 현재는 잘 쓰이지 않음 (대체로 `@PostConstruct` 사용)

---

7. Java 설정 기반 (`@Bean`)

```
@Configuration
public class AppConfig {

    @Bean(initMethod = "connect", destroyMethod = "disconnect")
    public NetworkClient networkClient() {
        return new NetworkClient();
    }
}
```

- **해당 메서드 이름의 public 메서드가 존재해야 함**
- `@PostConstruct`, `@PreDestroy` 보다 우선순위는 낮음
- 실무에서는 외부 라이브러리 Bean 등록 시 유용

---

8. XML 기반 설정

```
<bean id="networkClient" class="com.example.NetworkClient"
      init-method="connect" destroy-method="disconnect"/>
```

- 오래된 방식이지만 유지보수 중인 프로젝트에서는 여전히 존재

---

9. 스코프별 소멸 주의점

스코프	소멸 시점	`@PreDestroy` 호출 여부
`singleton`	컨테이너 종료 시	✅ 호출됨
`prototype`	컨테이너는 생성만 함	❌ 호출 안 됨 (직접 관리)
`request` / `session`	요청 또는 세션 종료 시	✅ 호출됨 (웹 컨테이너 관리)

> **`prototype` 스코프는 destroy 호출되지 않으므로 수동 정리 필요**
> → 예: `@PreDestroy`는 작동하지 않음

---

10. 정리된 초기화/소멸 흐름 (singleton 기준)

```
1. 객체 생성
2. 의존성 주입
3. @PostConstruct (또는 afterPropertiesSet)
4. 애플리케이션 사용
5. @PreDestroy (또는 destroy)
6. 컨테이너 종료
```

---

11. 실무 추천 전략

목적	추천 방식
일반 Bean 초기화/정리	✅ `@PostConstruct`, `@PreDestroy`
외부 라이브러리 Bean	`@Bean(initMethod, destroyMethod)`
XML 유지보수 프로젝트	XML `init-method`, `destroy-method`
절대 피해야 할 방식	`System.exit()`, Thread 종료 직접 호출

---

12. 예외 처리 시 유의사항

- `@PostConstruct` 내부에서 예외 발생 → **빈 등록 실패 → 컨테이너 전체 부팅 실패**
- 로그, 외부 API 호출, Redis 초기 연결 등에서 예외 처리 필수

```
@PostConstruct
public void init() {
    try {
        connectToRedis();
    } catch (Exception e) {
        log.error("Redis 연결 실패", e);
    }
}
```

13. 요약 비교표

항목	초기화 방법	소멸 방법	특징
어노테이션 방식	`@PostConstruct`	`@PreDestroy`	✅ 실무 기본
인터페이스 방식	`afterPropertiesSet()`	`destroy()`	❌ Spring 의존
설정 방식 (`@Bean`)	`initMethod` 지정	`destroyMethod` 지정	외부 라이브러리에 유리
XML 방식	`init-method` XML 속성	`destroy-method` 속성	레거시 프로젝트

- `@PostConstruct`, `@PreDestroy`

1. 개요

어노테이션	설명
`@PostConstruct`	Bean이 **생성되고 의존성 주입이 끝난 직후** 자동으로 실행되는 메서드에 사용
`@PreDestroy`	Bean이 **컨테이너에서 제거되기 직전** 자동으로 실행되는 메서드에 사용

> 두 어노테이션은 모두 **JSR-250 (Java EE)** 표준이며,
> Spring뿐만 아니라 Jakarta EE, Java 기반 프레임워크 전반에서 사용 가능

2. 사용 예제

```
@Component
public class NetworkClient {

    @PostConstruct
    public void init() {
        System.out.println("초기화: 서버 연결 시작");
    }

    @PreDestroy
    public void cleanup() {
        System.out.println("소멸: 서버 연결 종료");
    }
}
```

3. 동작 시점

`@PostConstruct`

- **객체 생성 + 의존성 주입 → 이후 실행**
- Spring이 `BeanFactory` 또는 `ApplicationContext`로 Bean 등록 후 실행

`@PreDestroy`

- 컨테이너가 종료되거나, `@Scope("request")` / `@Scope("session")` Bean의 생명주기가 끝날 때 실행

---

4. 조건 및 제약

조건	설명
메서드는 `void` 여야 함	반환값이 없어야 함
매개변수가 없어야 함	인자 있는 메서드는 실행되지 않음
예외 선언 가능	`throws Exception` 가능
여러 개 선언 시	모두 실행되지만 순서는 보장되지 않음
`private` 사용 가능	Spring은 리플렉션으로 접근 가능

---

5. 빈 라이프사이클 내 위치

```
1. 생성자 호출
2. 필드 및 생성자 주입 완료
3. @PostConstruct 실행 ← 여기!
4. 컨테이너에서 사용됨
5. @PreDestroy 실행   ← 여기!
6. 컨테이너 종료
```

---

6. 실무 사용 패턴

✅ 주로 사용하는 용도

용도	설명
외부 서버 연결	DB, Redis, 메시지 브로커 초기화
캐시 초기화	LocalCache, LoadingCache 등 초기 로드
리소스 할당	파일, 네트워크 소켓 등
연결 종료, 자원 해제	close, shutdown, disconnect 등

---

7. 생성자 vs `@PostConstruct`

항목	생성자	`@PostConstruct`
실행 시점	객체 생성 시	객체 생성 + 의존성 주입 완료 후
DI 객체 접근	❌ 불가 (주입 전)	✅ 가능
실무 용도	필드 초기화, 불변성 유지	외부 연결, 초기화 로직 수행에 적합

→ 대부분의 초기화는 `@PostConstruct`**가 적합**
→ 생성자에는 오직 **필드 값 검증 또는 필수 주입 체크만** 수행해야 함

---

8. Bean 스코프별 동작

스코프	`@PostConstruct`	`@PreDestroy`
`singleton`	✅ 실행됨	✅ 실행됨
`prototype`	✅ 실행됨	❌ 실행 안 됨
`request` / `session`	✅ 실행됨	✅ 실행됨

> ⚠️ `prototype`은 Spring이 소멸 관리하지 않음 → `@PreDestroy` 호출되지 않음

---

9. Spring Boot / Java Config에서도 사용 가능

```
@Configuration
public class AppConfig {

    @Bean
    public SomeService someService() {
        return new SomeService();
    }
}

public class SomeService {

    @PostConstruct
    public void init() { ... }

    @PreDestroy
    public void destroy() { ... }
}
```

---

10. 대체 방식 비교

방식	초기화	소멸	특징
어노테이션 (`@PostConstruct`)	✅ 권장	✅ 권장	✅ 가장 많이 사용
인터페이스 (`InitializingBean`)	`afterPropertiesSet()`	`destroy()` (DisposableBean)	❌ Spring 의존성 높음
설정 지정 (`@Bean(initMethod)`)	`initMethod="..."`	`destroyMethod="..."`	외부 클래스에 유용

---

11. 실무 팁

- Spring Boot에서는 `@PostConstruct`로 설정된 메서드에서 예외 발생 시, **전체 애플리케이션 구동 실패**
- 따라서 초기화 로직에서 반드시 **예외 처리**할 것:

```
@PostConstruct
public void init() {
    try {
        connectToRedis();
    } catch (Exception e) {
        log.error("Redis 연결 실패", e);
    }
}
```

---

12. 요약

항목	설명
`@PostConstruct`	Bean 초기화 후 실행되는 메서드 (자원 연결, 로딩 등)
`@PreDestroy`	Bean 소멸 직전 실행되는 메서드 (해제, 정리 등)
위치	클래스 내 `void`, 파라미터 없는 메서드에 적용
권장 여부	✅ Spring에서 가장 권장되는 초기화/종료 방식
스코프 제한	`prototype`에서는 `@PreDestroy` 동작하지 않음

조건부 Bean 등록 (`@Conditional`, `@Profile`)

1. 개요

Spring에서는 특정 **조건에 따라 Bean을 등록하거나 등록하지 않는 기능**을 제공한다.
이는 다음과 같은 상황에서 유용하다:

- 개발 vs 운영 환경 구분

- 특정 라이브러리 존재 여부에 따라 기능 활성화
- 운영체제에 따라 Bean 다르게 등록
- 커스텀 조건 클래스 적용

---

2. `@Profile`

✅ 개념

`@Profile`은 **지정한 환경(profile)에 따라 Bean을 선택적으로 등록**하는 기능이다.

```
@Profile("dev")
@Bean
public DataSource devDataSource() { ... }

@Profile("prod")
@Bean
public DataSource prodDataSource() { ... }
```

✅ 프로파일 활성화 방법

1) `application.yml`

```
spring:
  profiles:
    active: dev
```

2) 실행 파라미터

```
--spring.profiles.active=prod
```

3) 어노테이션 설정 클래스에 적용

```
@Configuration
@Profile("test")
public class TestConfig { ... }
```

✅ 주요 특징

항목	설명
선언 대상	클래스 or 메서드 레벨 모두 가능
활성화 조건	`spring.profiles.active` 값과 일치
AND/OR 조합	배열 형태로 다중 선언 가능: `@Profile({"dev", "test"})`

항목	설명
기본 프로파일	아무 프로파일도 지정하지 않았을 때 활성화: `@Profile("default")`

---

3. `@Conditional`

✅ 개념

`@Conditional`은 **보다 세밀하게 조건을 정의하고, 그 조건이 참일 때만 Bean을 등록**하는 어노테이션이다.
내부적으로는 `Condition` 인터페이스 구현체로 판단된다.

```
@Bean
@Conditional(WindowsCondition.class)
public MyBean windowsBean() {
    return new MyBean();
}
```

✅ `Condition` 인터페이스

```
public class WindowsCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        return System.getProperty("os.name").contains("Windows");
    }
}
```

- `ConditionContext`: 환경, 빈 팩토리, 레지스트리 접근 가능
- `AnnotatedTypeMetadata`: 어노테이션 정보 접근 가능

✅ 실전 조건 예

조건 클래스	설명
`OnClassCondition`	클래스가 존재할 때 등록 (예: 특정 라이브러리 존재 여부)
`OnPropertyCondition`	특정 설정 값이 존재할 때 등록
`OnWebApplicationCondition`	웹 애플리케이션일 경우 등록
직접 정의한 `MyCustomCondition`	사용자 정의 로직 가능

✅ 조건 작성 예 (환경 변수 기반)

```
public class RedisEnabledCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        String enabled = context.getEnvironment().getProperty("feature.redis.enabled");
        return "true".equalsIgnoreCase(enabled);
    }
}
```

```
@Bean
@Conditional(RedisEnabledCondition.class)
public RedisClient redisClient() {
    return new RedisClient();
}
```

---

4. `@ConditionalOn...` 시리즈 (Spring Boot)

Spring Boot에서는 `@Conditional`을 기반으로 한 여러 어노테이션을 **자동 설정에 활용**한다.

어노테이션	설명
`@ConditionalOnProperty`	설정값 존재 여부로 조건 제어
`@ConditionalOnMissingBean`	특정 Bean이 없을 때만 등록
`@ConditionalOnClass`	클래스(Classpath) 존재 시 등록
`@ConditionalOnWebApplication`	웹 애플리케이션일 경우 등록

예:

```
@Bean
@ConditionalOnProperty(name = "feature.mail.enabled", havingValue = "true")
public MailSender mailSender() {
    return new MailSender();
}
```

---

5. `@Profile` vs `@Conditional` 비교

항목	`@Profile`	`@Conditional`
주요 용도	환경(dev/prod/test)에 따라 분기	커스텀 조건, 라이브러리, OS 등 유연한 제어
선언 위치	클래스 또는 메서드	메서드 또는 클래스
조건 표현	문자열(프로파일 이름)	자바 코드로 직접 제어 가능

항목	`@Profile`	`@Conditional`
활용도	일반 설정 분리	고급 자동 설정, 라이브러리 조건
Spring Boot에서 사용	Spring Core	Spring Core + Boot 자동 설정에 광범위 활용

---

6. 실무 적용 전략

상황	사용 방법
개발/운영 환경 분기	`@Profile`
특정 설정 유무에 따라 기능 활성화	`@ConditionalOnProperty`
특정 클래스가 classpath에 존재할 경우	`@ConditionalOnClass`
자동 설정 제공 시 Bean 중복 방지	`@ConditionalOnMissingBean`
커스텀 로직 기반 조건 판단	직접 `Condition` 구현 후 `@Conditional` 사용

---

7. 정리 요약

항목	설명
`@Profile`	Spring 환경 분기용으로 간단하게 Bean 활성화
`@Conditional`	자바 코드 기반의 복잡한 조건 로직 처리
Spring Boot 확장	`@ConditionalOn...` 시리즈로 자동 설정 구현
우선순위	`@Conditional`이 더 강력하고 범용적임
실무 사용 전략	프로파일 → 기본 분기, 조건부 → 기능 활성화 및 자동 설정에 활용