

# Spring DI & AOP

## 1-1. Spring의 등장 배경

- EJB는 자바 엔터프라이즈 애플리케이션을 개발하기 위한 표준 기술  
→ EJB에서 제공하는 많은 기능에도 불구하고 EJB를 적용하는 것은 너무 어려움
- Rod Johnson은 자신의 저서에서 EJB를 사용하지 않고 엔터프라이즈 애플리케이션을 개발하는 방법을 소개  
→ Spring Framework의 시초

### EJB 명세서에 기술된 EJB의 목표

EJB를 사용하면 애플리케이션 작성을 쉽게 할 수 있다. 저수준의 트랜잭션이나 상태관리, 멀티스레딩, 리소스 풀링과 같은 복잡한 저수준의 API를 이해하지 못하더라도 아무 문제 없이 애플리케이션을 개발 할 수 있다.

- *Enterprise JavaBean 1.0 Specification, Chapter 2 Goals*

어려움

### EJB의 대안 → Spring 프레임워크



## 1-1. Spring의 등장 배경



Rod Johnson

*EJB가 문제가 많아서  
"EJB 없이 J2EE 개발하기"  
출간*

*Good!  
Why don't you make an  
Open-source?*



Juergen Hoeller



Yann Caroff

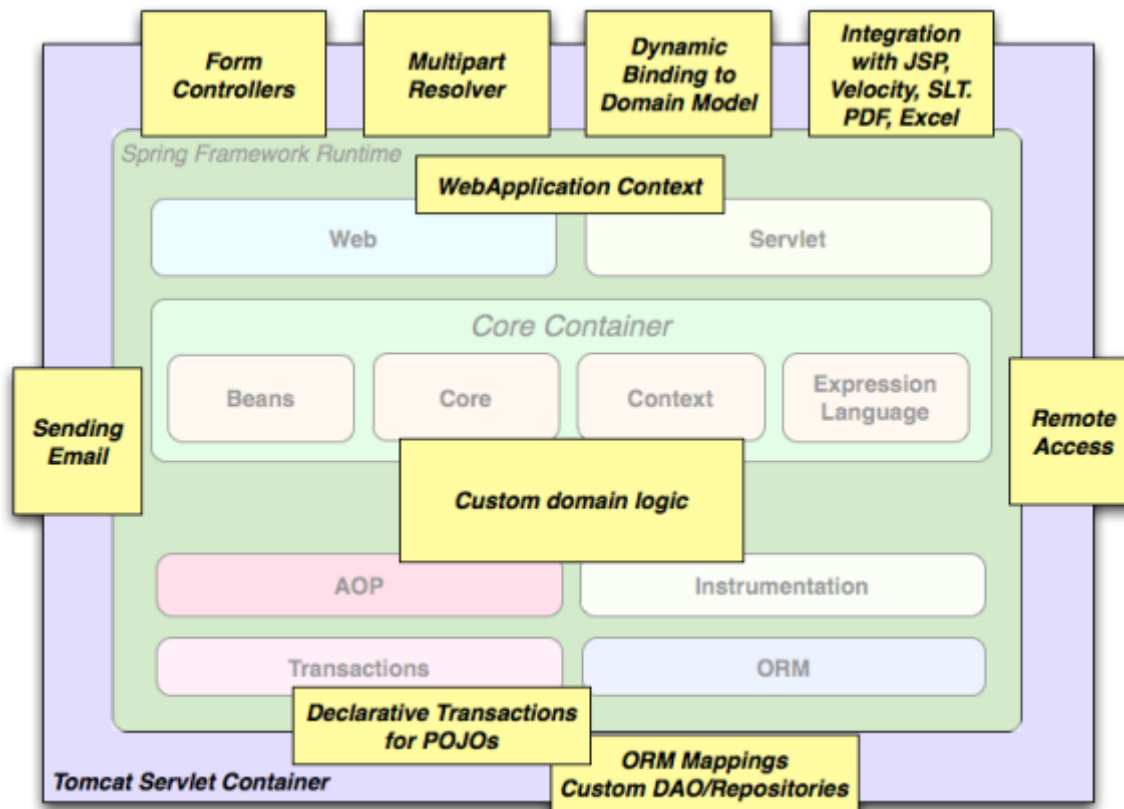
*EJB = **Winter!***



***Spring***

## 1-2. Spring 개요 – 애플리케이션 프레임워크

- Spring 프레임워크는 자바 엔터프라이즈 개발을 위한 오픈소스 경량 애플리케이션 프레임워크
- 애플리케이션 프레임워크는 애플리케이션 개발의 모든 계층을 지원
- 프레임워크가 애플리케이션 수준의 인프라 스택을 제공하므로 개발자는 업무 로직 개발에만 집중
- Spring 프레임워크는 공통 프로그래밍 모델 및 Configuration 모델을 제공



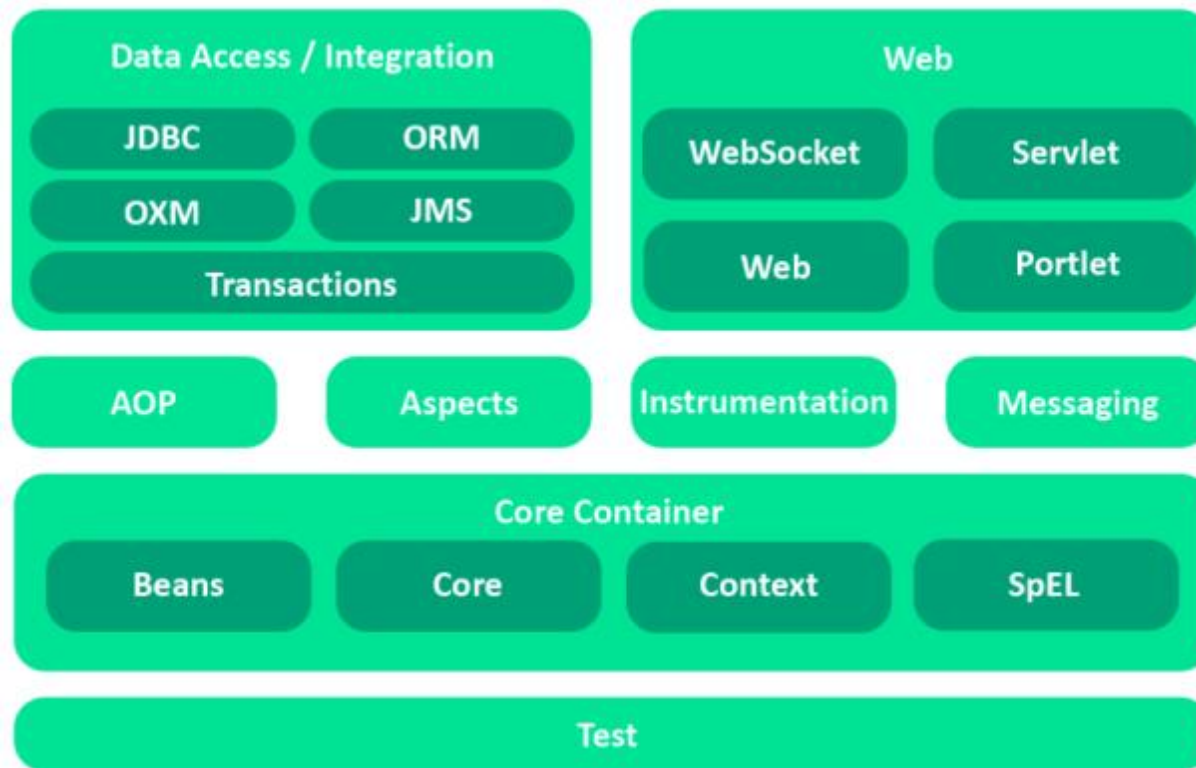
## 1-2. Spring 개요 – Release History

- 2003년 6월 아파치 2.0 라이선스로 공개
- Release History
  - 1.0: 2004년 4월
  - 2.0: 2006년 10월
  - 2.5: 2007년 11월
  - 3.0: 2009년 12월
  - 3.1: 2011년 12월
  - 3.1.4: 2013년 1월
  - 4.0.1: 2014년 1월
  - 4.0.5: 2014년 5월



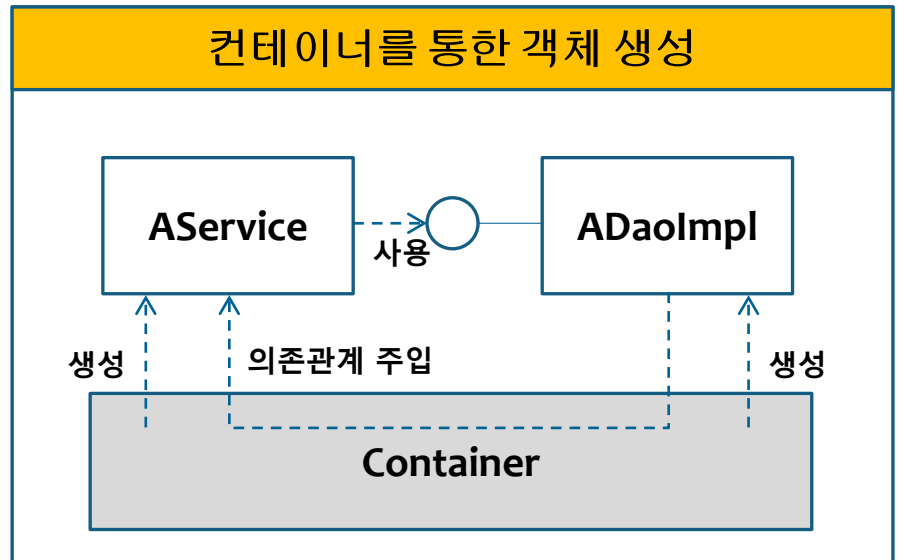
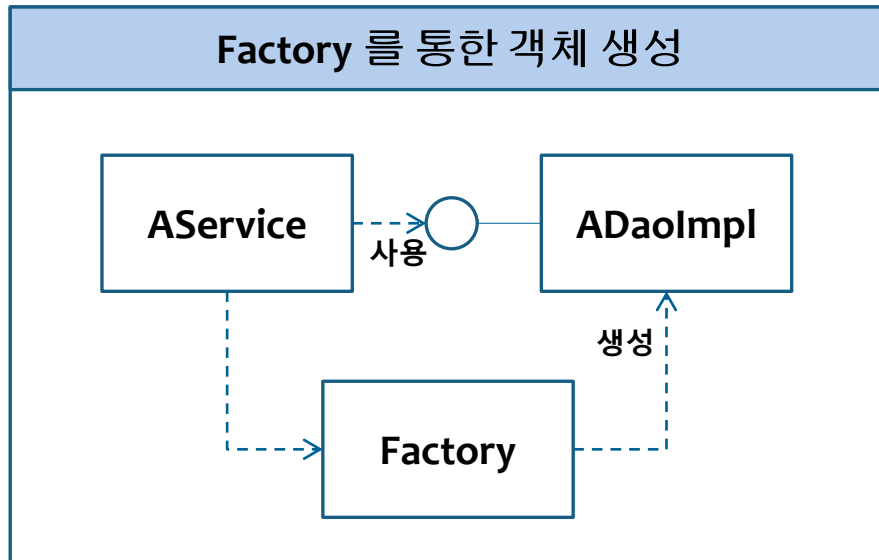
## 1-2. Spring 개요 – 경량 프레임워크

- EJB는 분산 환경을 위한 컴포넌트
- Spring은 EJB가 제공하는 서비스를 지원하는 경량 프레임워크  
→ EJB와 달리 기술과 환경에 의존적인 부분을 제거한 프레임워크
- Spring은 수십만 라인의 복잡한 코드로 되어 있으나, 20여 개의 모듈로 나누어진 경량 프레임워크



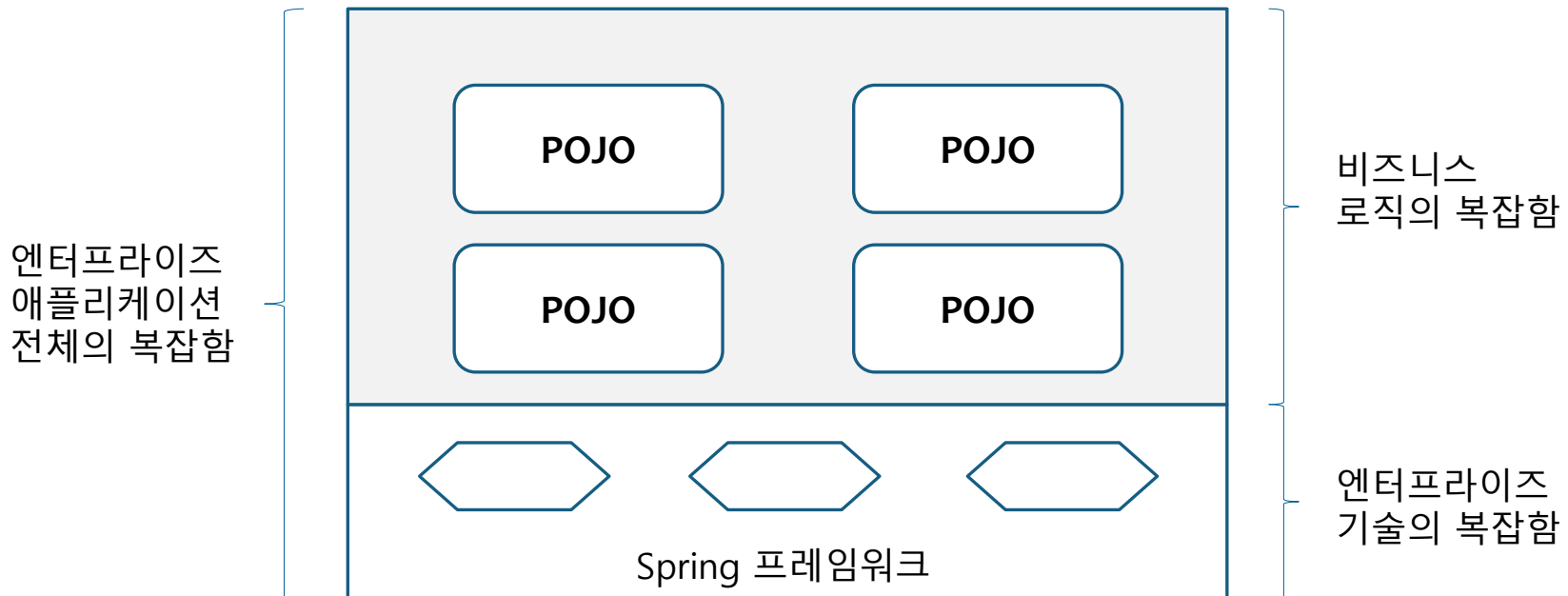
## 1-2. Spring 개요 – 경량 프레임워크 (Lightweight Container)

- Spring은 애플리케이션 객체의 생명주기와 설정(Configuration)을 관리하는 경량 컨테이너
- Spring은 객체를 생성하고, 의존관계 설정(Dependency Injection)을 통해 객체 간의 관계를 지정
- 애플리케이션 모든 계층의 객체가 관리 대상
  - 각 계층이나 서비스들간의 의존관계를 관리
- 의존관계 설정 (Dependency Injection)을 통해 인터페이스 기반의 컴포넌트로 업무 로직을 모듈화 할 수 있음



## 1-2. Spring 개요 – POJO 프레임워크

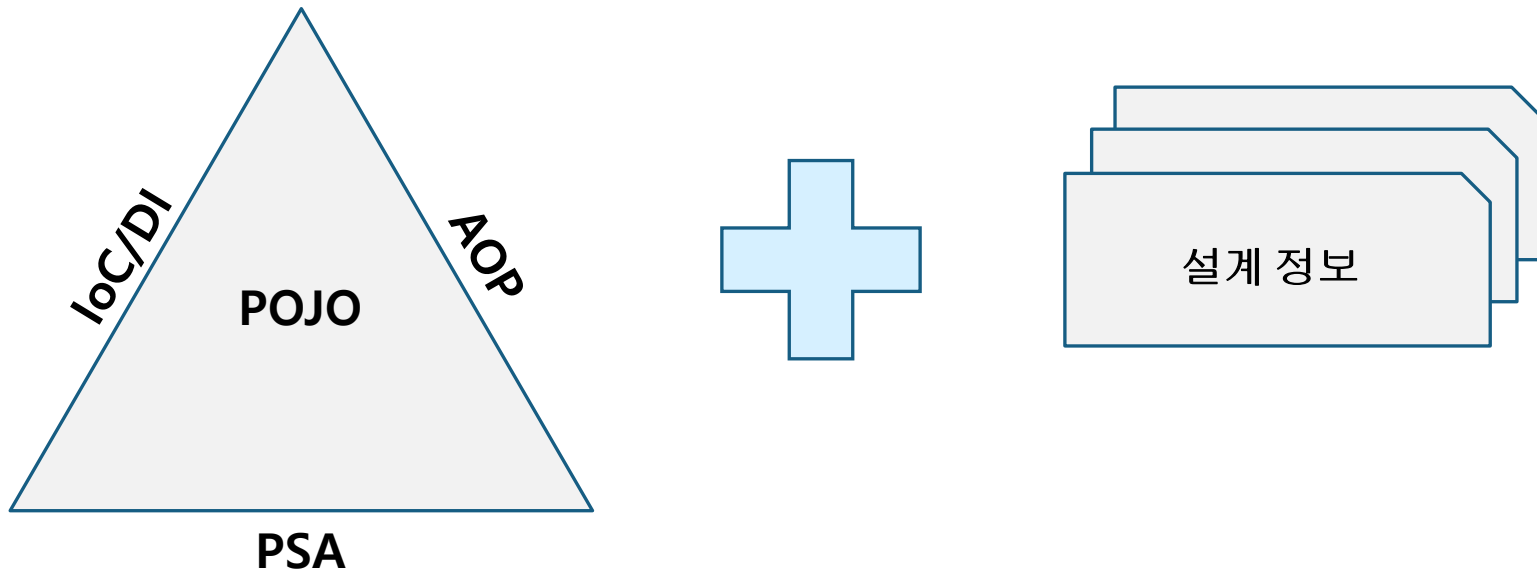
- Spring은 업무 로직과 엔터프라이즈 기술의 복잡함을 분리하여 처리 할 수 있도록 지원  
→ Spring은 특정 프레임워크나 기술에 얽매이지 않는 POJO로 업무 로직을 구성할 수 있음
- POJO (Plain Old Java Object)란 특별한 기술에 종속되지 않은 순수한 자바 객체를 말함
- POJO로 개발하면 테스트가 쉽고, 객체지향 설계 적용이 용이하여 개발 생산성 및 이식성이 향상





## 1-2. Spring 개요 – POJO 프레임워크

- Spring은 업무 로직을 POJO만으로 개발할 수 있는 POJO 프레임워크
- Spring은 POJO로 개발할 수 있도록 IoC/DI, AOP, PSA 기술을 지원
- Spring 애플리케이션은 POJO를 이용해서 만든 업무 코드와 설계정보로 구성
- 설계정보는 POJO 사이의 관계 및 동작방법을 정의



## 1-2. Spring 개요 – Spring 프로젝트



### SPRING FRAMEWORK

Provides core support for dependency injection, transaction management, web apps, data access, messaging



### SPRING DATA

Provides a consistent approach to data access – relational, non-relational, map-reduce, and beyond.



### SPRING BATCH

Simplifies and optimizes the work of processing high-volume batch operations.



### SPRING FOR ANDROID

Provides key Spring components for use in developing Android applications.

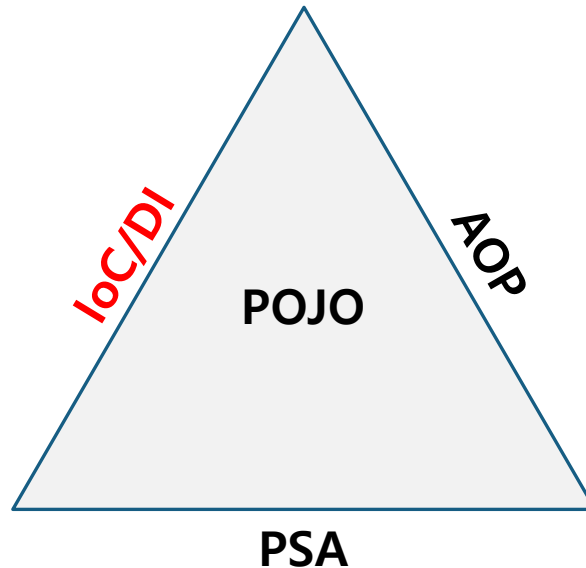


### SPRING SECURITY

Protects your application with comprehensive and extensible authentication and authorization support.

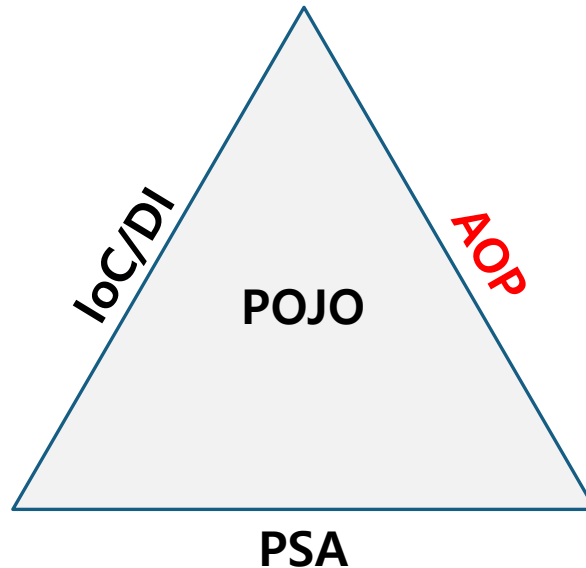
### 1-3. Spring 핵심기술

- Inversion Of Control를 줄여서 IoC
- IoC는 객체지향 언어에서 객체 간의 연결 관계를 런타임 시에 결정하게 하는 방법
  - 객체 간의 관계를 느슨하게 연결하도록 구현 (Loose coupling)
- DI는 Dependency Injection의 약자로 IoC의 구현 방법의 하나



### 1-3. Spring 핵심기술

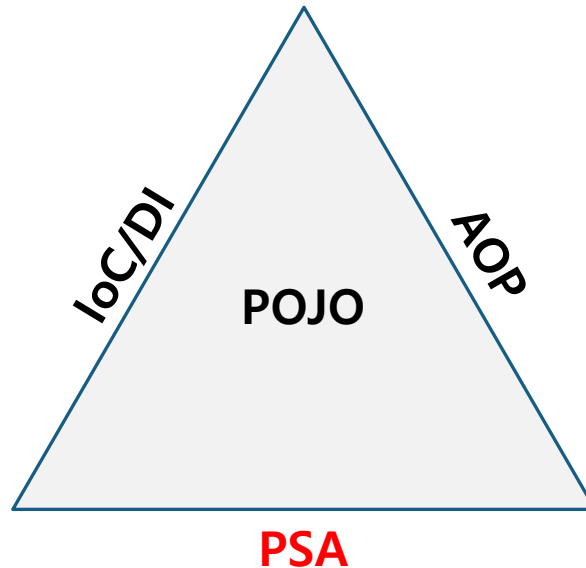
- Aspect Oriented Programming를 줄여서 AOP
- AOP는 관심사의 분리(SoC)를 통해서 소프트웨어 모듈성을 향상 시키고자 하는 프로그래밍 패러다임
- 소스코드 레벨에서 관심사의 모듈화를 지향하는 프로그래밍 방법이나 도구를 포함



### 1-3. Spring 핵심기술

- **Portable Service Abstraction**을 줄여서 **PSA**
- **PSA**는 환경과 세부 기술의 변화에 관계없이 일괄된 방식으로 기술에 접근할 수 있게 해주는 설계 원칙

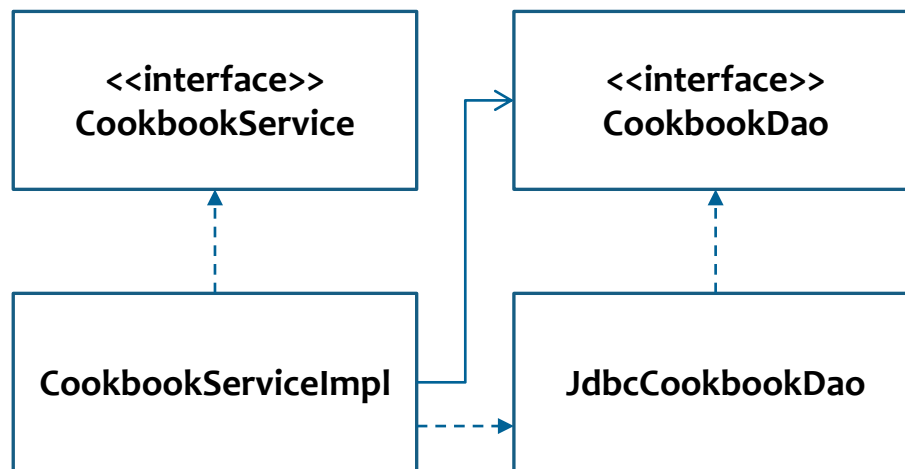
ex. 데이터베이스에 관계 없이 동일하게 적용할 수 있는 트랜잭션 처리 방식



## 2-1. IoC

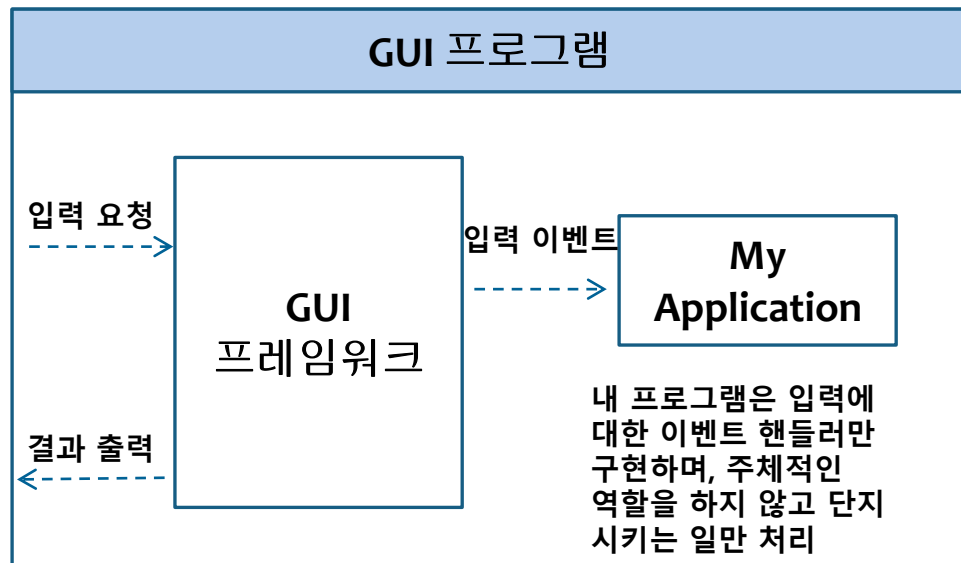
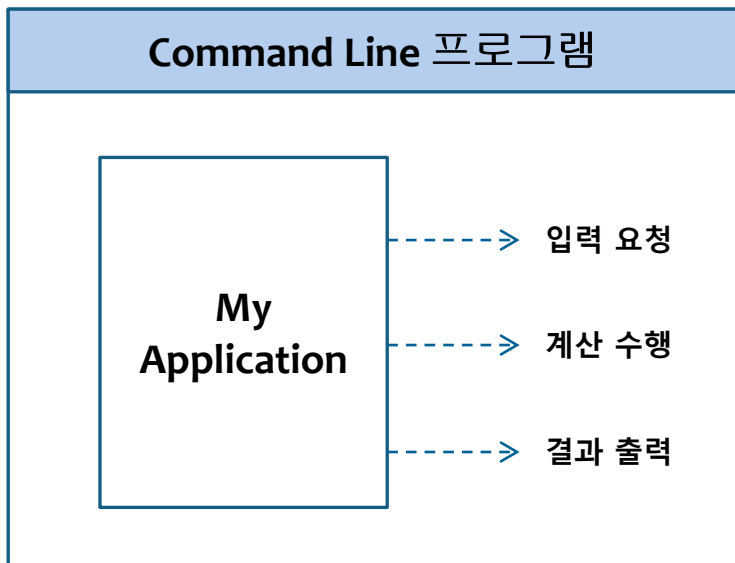
- 일반적인 프로그램들은 필요한 객체를 사용하는 위치에서 생성
- OOP에서는 인터페이스와 추상 클래스를 사용하여 객체간의 의존성을 떨어뜨림
  - 소스코드에 하드코딩 된 객체 생성 로직은 객체간의 의존성을 강제하게 됨
  - **의도하지 않았지만 객체 간의 결합도는 높아짐**

```
public class CookbookServiceImpl implements CookbookService {  
    private CookbookDao cookbookDao;  
  
    public CookbookServiceImpl() {  
        this.cookbookDao = new JdbcCookbookDao();  
    }  
  
    @Override  
    public List<Cookbook> findAllCookbooks() {  
        //  
        return cookbookDao.retrieveAll();  
    }  
    ...  
}
```



## 2-1. IoC

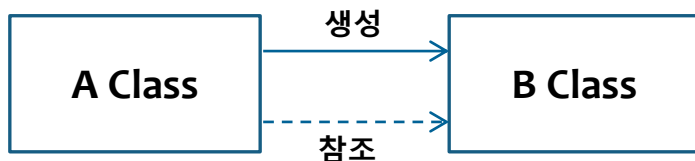
- IoC(Inversion of Control)는 통제 방향의 변경을 의미
  - 프레임워크의 일반적인 속성으로 어떤 일을 하는 주체를 변경하는 것
- 최초의 Command Line 프로그램에서는 프로그램이 모든 것을 통제
  - GUI 프로그램은 프레임워크가 통제하며 프로그램에서는 이벤트 핸들러만 구현



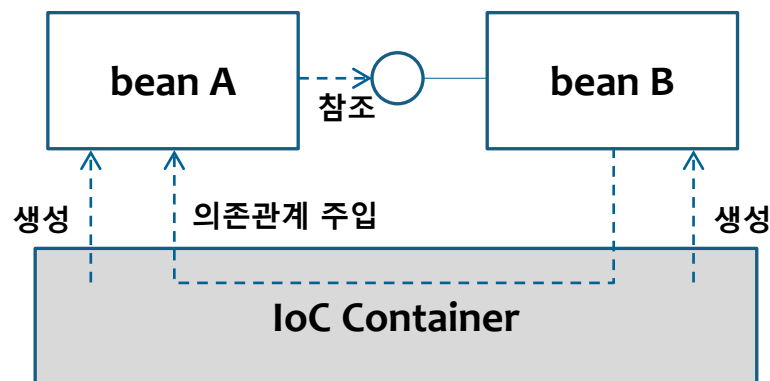
## 2-1. IoC

- 객체 간 결합도가 높으면, 해당 클래스가 변경될 때 결합된 다른 클래스도 같이 수정될 가능성이 있음
- IoC는 객체 생성 책임을 컨테이너에게 위임  
→ 객체 간의 결합도를 낮춤 (Loose coupling)
- IoC를 통한 객체 제어 방식은 기존 로직에서 객체를 생성하는 로직을 제거
- IoC는 구현하는 방법에 따라, Dependency Injection과, Dependency Lookup 방법이 있음

기존 객체 제어 방식



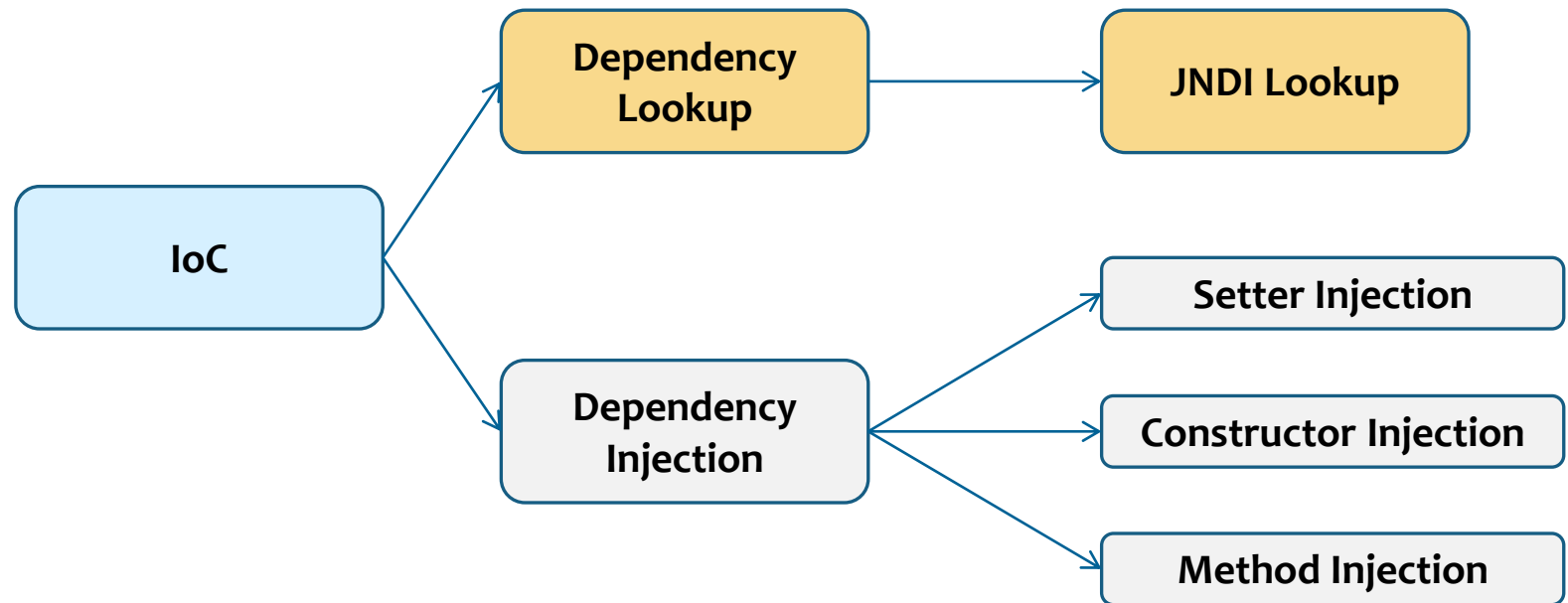
IoC 객체 제어 방식





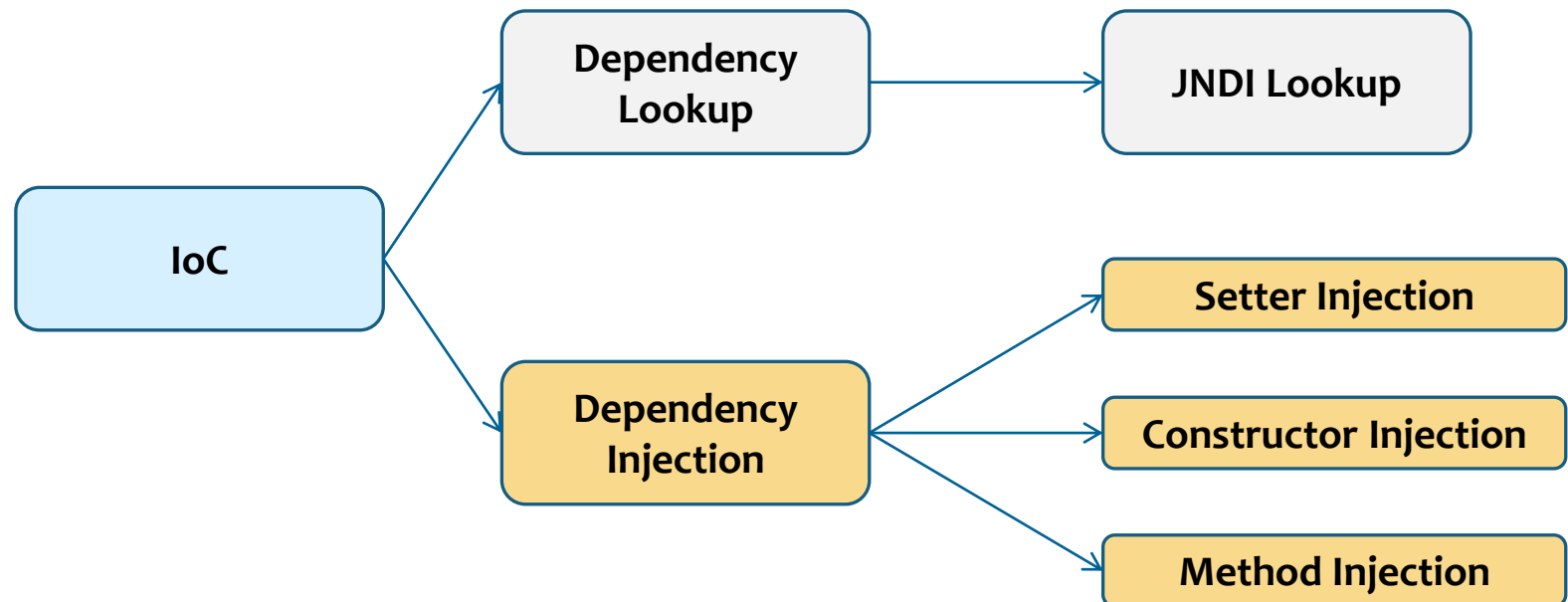
## 2-1. IoC – Dependency Lookup

- Dependency Lookup 방식은 EJB나 Spring에서 JNDI 리소를 얻는 방식
- 컨테이너가 제공하는 Lookup Context를 통해서 필요한 자원이나 객체를 얻을 수 있음
  - 결과적으로 컨테이너 API에 대한 의존성을 높하게 됨
- 컨테이너와의 의존성을 줄이기 위해서는 DI 방식을 사용



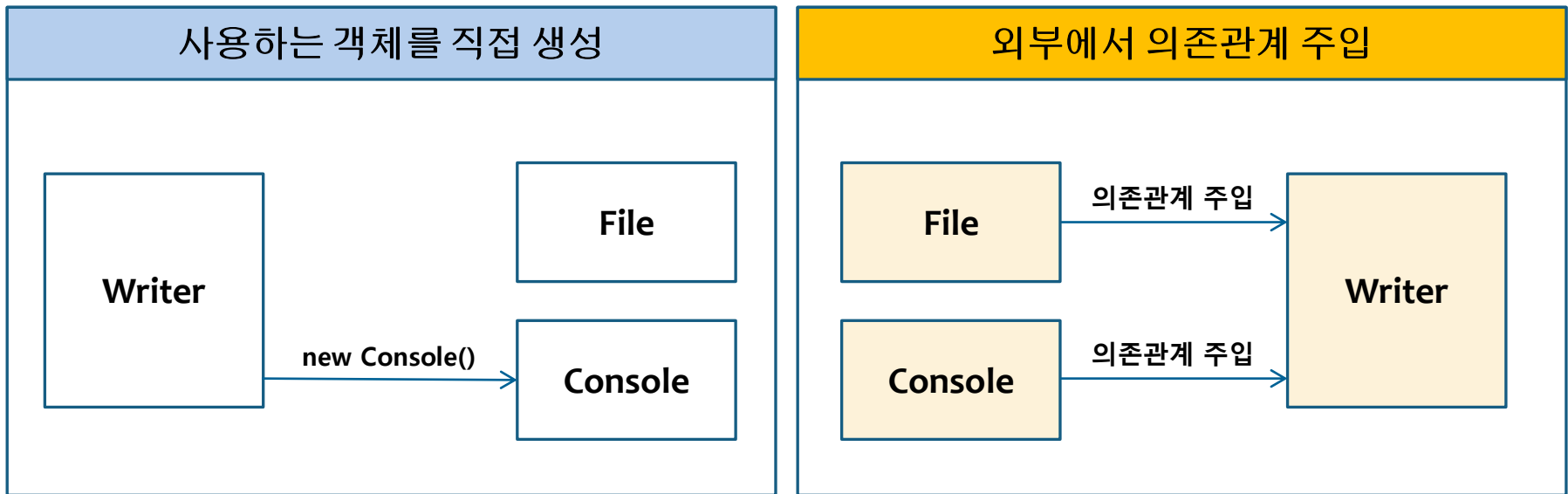
## 2-1. IoC – Dependency Injection

- Dependency Injection은 컨테이너가 직접 의존구조를 객체에 설정할 수 있도록 지정하는 방식
- 객체는 컨테이너의 존재 여부를 알 필요가 없음  
→ lookup 관련된 코드도 객체에서 사라짐
- Injection은 Setter, Constructor, Method Injection 로 구분
- DI 프레임워크는 PicoContainer, Spring DI, Google Guice 가 존재



## 2-1. IoC – Dependency Injection

- 의존 관계 주입 Dependency Injection를 줄여서 DI
- DI를 통해, 외부에서 객체를 생성하는 시점에 참조하는 객체에게 의존관계를 제공
- 협업 객체의 참조를 어떻게 얻어낼 것인가라는 관점에서 책임성의 역행(Inversion of responsibility)라고도 함
- 객체가 인터페이스만 알고 있으므로, 느슨한 결합(Loose coupling)이 가능

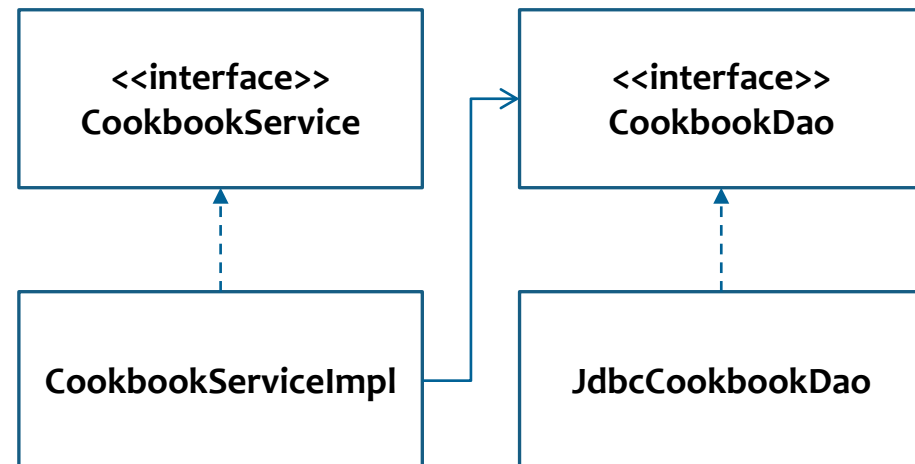


※ 객체는 인터페이스에 의한 의존관계만을 알고 있으며, 구현 클래스에 대한 차이를 모르기 때문에, 서로 다른 구현체로 대체가 가능

## 2-1. IoC – Dependency Injection

- 사용하는 객체 생성을 IoC 컨테이너에게 위임하며, 객체 생성 로직이 없어짐
- 객체 생성에 대한 주도권을 객체를 필요한 하는 곳으로 넘겨주어, 필요할 때 필요한 곳에서 객체를 생성하는 방법
- Spring은 Setter, Constructor, Method Injection 등 3가지 DI 패턴을 모두 지원
- Spring은 의존관계를 설정하는 방법으로 XML, Annotation, 자바 소스에서 지정할 수 있음

```
public class CookbookServiceImpl implements CookbookService {  
    private CookbookDao cookbookDao;  
  
    public void setCookbookDao(CookbookDao cookbookDao) {  
        this.cookbookDao = cookbookDao;  
    }  
  
    @Override  
    public List<Cookbook> findAllCookbooks() {  
        //  
        return cookbookDao.retrieveAll();  
    }  
  
    ...  
}
```



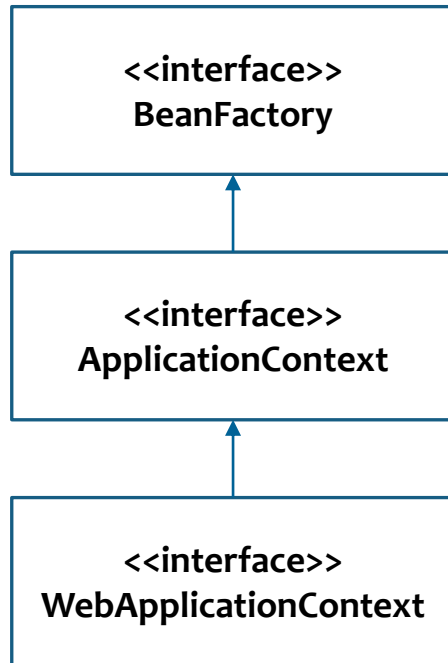
## 2-2. Spring IoC – Spring IoC 용어

- 빈(Bean)이란, Spring이 IoC 방식으로 관리하는 객체  
→ 관리되는 객체(Managed Object)
- IoC 컨테이너는 빈의 생성과 제어의 관점에서 빈 팩토리(Bean Factory)
- 애플리케이션 컨텍스트는 Spring이 제공하는 애플리케이션 지원 기능을 모두 포함하는 의미
- Spring 프레임워크는 IoC 컨테이너와 애플리케이션 컨텍스트를 포함한 Spring의 모든 기능을 포괄

IoC 용어	설명
bean	Spring이 IoC 방식으로 관리하는 객체 (관리되는 객체) Spring 직접 생성과 제어를 담당하는 오브젝트만을 빈이라고 함
bean factory	Spring이 IoC를 담당하는 핵심 컨테이너로 빈을 등록, 생성, 조회하고 반환, 그 외에 부가적으로 빈을 관리하는 기능을 담당. 보통은 이 빈 팩토리를 바로 사용하지 않고, 이를 확장한 애플리케이션 컨텍스트를 이용
application context	빈 팩토리를 확장한 IoC 컨테이너로 빈을 등록, 관리하는 기본적인 기능은 빈 팩토리와 동일. Spring이 제공하는 각종 부가 서비스를 추가로 제공
configuration metadata	애플리케이션 컨텍스트 또는 빈 팩토리가 IoC를 적용하기 위해 사용하는 메타정보
Spring framework	IoC 컨테이너, 애플리케이션 컨텍스트를 포함해서 Spring이 제공하는 모든 기능

## 2-2. Spring IoC – IoC 컨테이너

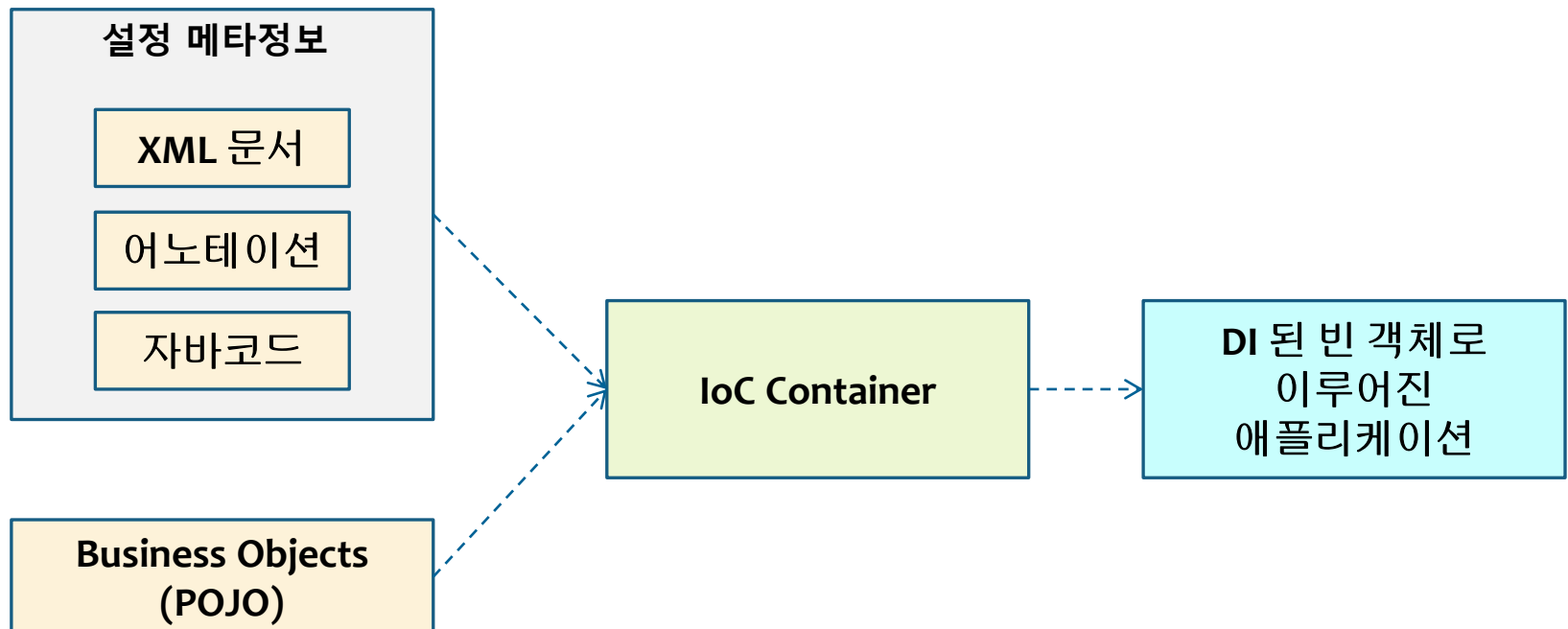
- 오브젝트의 생성과 관계설정, 사용, 제거 등의 작업을 애플리케이션 코드 대신 독립된 컨테이너가 담당
- 코드가 아닌 컨테이너가 객체에 대한 제어권을 가지고 있기 때문에 IoC  
→ Spring 컨테이너 = IoC 컨테이너
- Spring에서 IoC를 담당하는 컨테이너에는 BeanFactory, ApplicationContext
- Spring은 기본적으로 따로 설정하지 않으면 내부에서 생성하는 빈 객체를 모두 싱글톤으로 생성



- bean 객체에 대한 생성과 제공을 담당
  - 단일 유형의 객체를 생성하는 것이 아니라, 여러 유형의 bean을 생성, 제공
  - 객체 간의 연관 관계를 설정, 클라이언트의 요청 시 bean을 생성
  - bean의 라이프 사이클을 관리
- 
- BeanFactory가 제공하는 모든 기능 제공
  - 엔터프라이즈 애플리케이션을 개발하는데 필요한 여러 기능을 추가
  - I18N, 리소스 로딩, 이벤트 발생 및 통지
  - 컨테이너 생성 시 모든 빈 정보를 메모리에 로딩
  - 싱글톤 레지스트리로서의 애플리케이션 컨텍스트임
  - 기존의 오브젝트 팩토리과 비슷한 방식으로 동작하는 IoC 컨테이너
- 
- 웹 환경에서 사용할 때 필요한 기능이 추가된 애플리케이션 컨텍스트
  - 가장 많이 사용, 특히 **XmlWebApplicationContext**를 가장 많이 사용

## 2-2. Spring IoC – 설정메타정보

- 하나의 애플리케이션은 IoC 컨테이너에 의해 POJO 클래스에 설정 메타정보가 결합되어 만들어 짐
- 설정 메타정보는 애플리케이션을 구성하는 객체와 객체사이의 상호 의존성을 포함함
- Spring은 XML 설정 파일과 어노테이션 설정, 자바소스로 메타정보 설정 가능



## 2-3. Spring XML 빈(bean) 관리 – 빈 선언

- XML 방식으로 Spring 빈 설정 가능
  - 단순하며 사용하기 쉬움
  - 가장 많이 사용하는 방식 – 빈의 메타 정보를 XML 문서 형태로 기술
- XML 설정파일에 <bean> 태그를 작성하여 빈을 선언
- Spring은 빈을 선언함과 동시에 <property> 태그의 ref 요소를 이용하여 의존관계를 설정

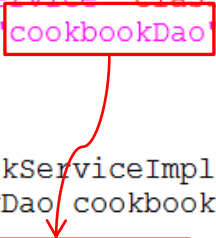
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans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

```
  <bean id="cookbook" class="com.lectopia.spring.dao.jdbc.JdbcCookbookDao" />
  <bean id="cookbookService" class="com.lectopia.spring.service.impl.CookbookServiceImpl">
    <property name="cookbookDao" ref="cookbook" />
  </bean>
</beans>
```

```
public class CookbookServiceImpl implements CookbookService {
    private CookbookDao cookbookDao;

    public void setCookbookDao(CookbookDao cookbookDao) {
        this.cookbookDao = cookbookDao;
    }

    ...
}
```





## 2-3. Spring XML 빈(bean) 관리 – Autowiring

- XML 방식으로 빈을 선언하면, 자동 엮음(Autowiring)을 이용하여 Spring 빈 사이의 의존관계 설정 가능
- Autowiring은 명시적으로 프로퍼티나 생성자 파라미터 지정 없이 컨테이너가 DI 설정을 추가
- 대표적인 Autowiring은 **byName, byType**

Autowiring 유형	설명
byName	빈(bean)의 이름(ID)이 엮여지는 속성의 이름과 같은 것을 컨테이너에서 검색 만일 일치하는 빈(bean)이 없으면 속성은 엮여지지 않은 상태로 남게 됨
byType	빈(bean)의 타입이 엮여지는 속성의 타입과 같은 것을 컨테이너에서 단일한 빈(bean) 을 검색 만일 일치하는 빈(bean)이 없으면 속성은 엮여지지 않은 상태로 남게 됨 일치하는 빈(bean)이 2개 이상이면 org.springframework.beans.factory.UnsatisfiedDependencyException 예외 발생
constructor	엮여질 빈(bean) 생성자들 중에 하나의 파라미터들로 컨테이너의 하나 이상의 빈 (bean)들을 일치 모호한 빈(bean)이나 모호한 생성자의 경우, org.springframework.beans.factory.UnsatisfiedDependencyException 예외 발생
autodetect	처음에 constructor를 사용해서 Autowiring을 시도하고 그 다음에 byType으로 Autowiring을 시동 모호한 부분은 constructor와 byType과 같이 동일한 방식으로 처리

## 2-3. Spring XML 빈(bean) 관리 – Autowiring

- Autowiring을 이용할 경우, Autowiring 방식에 대한 선언 필요
- 각 빈마다 Autowiring 방식 지정 가능
- default-autowire를 사용하면 선언된 모든 빈에 적용 가능

Autowiring을 적용하지 않은 빈 선언

```
<bean id="cookbook" class="com.lectopia.spring.dao.jdbc.JdbcCookbookDao" />
<bean id="cookbookService"
      class="com.lectopia.spring.service.impl.CookbookServiceImpl">
  <property name="cookbookDao" ref="cookbook" />
</bean>
</beans>
```

DI가 필요한 클래스

```
public class CookbookServiceImpl implements CookbookService {
    private CookbookDao cookbookDao;

    public void setCookbookDao(CookbookDao cookbookDao) {
        this.cookbookDao = cookbookDao;
    }

    ...
}
```

autowire를 이용하여 빈의 Autowiring 방식 설정

```
<beans xmlns="http://www.springframework.org/schema/beans" ***>

<bean id="cookbook" class="com.lectopia.spring.dao.jdbc.JdbcCookbookDao" />
<bean id="cookbookService"
      class="com.lectopia.spring.service.impl.CookbookServiceImpl"
      autowire="byType" />
</beans>
```

default-autowire를 이용한 모든 빈의 Autowiring 방식 설정

```
<beans xmlns="http://www.springframework.org/schema/beans" ***
      default-autowire="byType">

<bean id="cookbook" class="com.lectopia.spring.dao.jdbc.JdbcCookbookDao" />
<bean id="cookbookService"
      class="com.lectopia.spring.service.impl.CookbookServiceImpl" />
</beans>
```

## 2-4. Spring Annotation 빈(bean) 관리 – 빈 선언

- 클래스에 어노테이션을 추가하는 방식으로 Spring 빈 설정 가능
- 빈으로 사용할 클래스에 특별한 어노테이션을 부여해 주면 자동으로 빈으로 등록
- “오브젝트 빈 스캐너”로 빈 스캐닝을 통해 자동으로 빈 등록 가능
- 빈 스캐너는 기본적으로 빈의 아이디로 클래스 이름(클래스 이름의 첫 글자만 소문자로 바꾼 것)을 사용

```
@Component
public class CookbookServiceImpl implements CookbookService {
    @Autowired
    private CookbookDao cookbookDao;

    @Autowired
    private RecipeDao recipeDao;

    @Override
    public void setCookbookDao(CookbookDao cookbookDao) {
        this.cookbookDao = cookbookDao;
    }

    ...
}
```

## 2-4. Spring Annotation 빈(bean) 관리 – 빈 선언

- 어노테이션으로 빈 선언 시 <context:component-scan> 설정 필요
- context 네임스페이스의 태그를 처리할 수 있는 핸들러가 컨테이너 인프라 빈을 등록
- 컨테이너 인프라 빈의 빈 스캐너 기능을 통해 @Component와 같은 스테레오 타입의 클래스를 빈으로 등록
- componet-scan은 빈 스캐너 기능과 <context:annotation-config>의 기능을 포함

```
<?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">

  <!-- Service Component Scan -->
  <context:component-scan base-package="com.lectopia.spring.service" />

  <!-- DAO Component Scan -->
  <context:component-scan base-package="com.lectopia.spring.dao" />

</beans>
```

## 2-4. Spring Annotation 빈(bean) 관리 – Stereotype Annotation

- Spring은 ApplicationContext의 BeanDefinition를 자동적으로 등록  
→ @Repository, @Service, @Controller, @Component 스테레오타입 클래스들을 자동으로 인식
- Spring이 관리하는 모든 컴포넌트를 계층 별로 빈의 특성이나 종류에 따라 스테레오타입 어노테이션을 사용  
→ AOP 포인트 컷 표현식을 사용하여 특정 어노테이션을 선언한 클래스만을 선정할 수 있음

스테레오타입 어노테이션	설명
@Repository	데이터 액세스 계층의 DAO 또는 Repository 클래스에 사용 DataAccessException 자동변환과 같은 AOP의 적용 대상을 선정하기 위해 사용하기도 함
@Service	서비스 계층의 클래스에 사용
@Controller	프리젠테이션 계층의 MVC 컨트롤러에 사용 Spring 웹 서블릿에 의해 웹 요청을 처리하는 컨트롤러 빈으로 선정
@Component	위의 계층 구분을 적용하기 어려운 일반적인 경우에 사용

## 2-4. Spring Annotation 빈(bean) 관리 – Injection Annotation

- 어노테이션을 이용하여 Spring 빈 사이의 의존관계 설정 가능
- 지원하는 표준 스펙에 따라 **@Resource**, **@Autowired**, **@Inject** 3가지로 분류
- Spring은 기본적으로 **byType** 방식으로 객체 타입에 맞는 Bean을 검색하여 의존성을 주입함

Injection 어노테이션	설명
@Resource	JSR-250 표준 Annotation, Spring 2.5 부터 지원 JSR-250은 JNDI를 이용한 datasource 등의 인젝션을 위한 사용 멤버변수, Setter 메소드에 사용 가능
@Autowired	Spring 2.5부터 지원 Spring에서만 사용가능, required 속성을 통해 DI 여부 조정 멤버변수, Setter, 생성자, 일반 메소드에 사용 가능
@Inject	JSR-330 표준, Spring 3.0부터 사용 가능 프레임워크에 종속적이지 않음, javax.inject-x.x.x.jar 필요 멤버변수, Setter, 생성자, 일반 메소드에 사용 가능

## 2-4. Spring Annotation 빈(bean) 관리 – DI

- 클래스의 참조할 객체에 @Autowired와 같은 특정 어노테이션을 추가
- 어노테이션으로 객체 사이의 의존관계를 설정하기 위해서는  
<context:annotation-config> 설정 필요
- annotation-config는 관계설정, 후처리 등의 기능이 있는 컨테이너 인프라 빈을 등록
- annotation-config는 빈을 등록하는 기능이 없기 때문에, 빈 선언을 추가

```
<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 id="cookbook" class="com.lectopia.spring.dao.jdbc.JdbcCookbookDao" />
  <bean id="cookbookService"
    class="com.lectopia.spring.service.impl.CookbookServiceImpl" />
</beans>
```

→ @Autowired를 처리하기 위한 빈을 등록

설정 파일에 stereotype annotation을 처리할 수 있는 빈을 등록하지 않으면 무시 됨

```
@Component
public class CookbookServiceImpl implements CookbookService {
```

```
    @Autowired
    private CookbookDao cookbookDao;
```

```
    @Override
    public List<Cookbook> findAllCookbooks() {
        //
        return cookbookDao.retrieveAll();
    }
    ...
}
```

프로퍼티에 의존성 주입을 위해 Injection Annotation을 선언

## 2-4. Spring Annotation 빈(bean) 관리 – DI

- **<context:component-scan>** 은 **<context:annotation-config>**의 기능 포함  
→ **component-scan** 선언 시 **annotation-config** 설정 생략 가능

```
<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">
```

```
<!-- Service Component Scan -->
```

```
<context:component-scan base-package="com.lectopia.spring.service" />
```

```
<!-- DAO Component Scan -->
```

```
<context:component-scan base-package="com.lectopia.spring.dao" />
```

```
</beans>
```

**@Autowired**와  
**@Component**를 처리하기  
위한 빈 등록  
**base-package**로 빈스캔 및  
DI하려고하는 빈들을 선택

**@Component**

```
public class CookbookServiceImpl implements CookbookService {
```

설정 파일에 **stereotype annotation**을 처리할 수  
있는 빈을 등록하였으므로, 빈으로 등록 됨

**@Autowired**

```
private CookbookDao cookbookDao;
```

**@Override**

```
public List<Cookbook> findAllCookbooks() {
    //
    return cookbookDao.retrieveAll();
}
```

프로퍼티에 의존성 주입을 위해 **Injection Annotation**을 선언

```
}
```

```
...
```



## 2-4. Spring Annotation 빈(bean) 관리 – @Resource

- @Resource를 이용하여 필드에 주입하거나, setter에 주입
- @Autowired와 동일한 기능을 처리
  - JNDI를 이용한 datasource 등의 인젝션을 위한 JSR-250 표준
- 동일한 타입의 빈이 여러 개일 경우 name을 통해서 빈을 구분
  - byName 방식으로 선언된 Bean의 이름(DI)를 검색하여 의존성 주입

### 필드 주입

```
@Repository
public class JdbcCookbookDao implements CookbookDao {
    @Resource(name = "dataSource2")
    private DataSource dataSource;

    @Override
    public List<Cookbook> retrieveAll() {
        ...
    }
}
```

### Setter 주입

```
@Repository
public class JdbcCookbookDao implements CookbookDao {

    private DataSource dataSource;

    @Resource(name = "dataSource")
    public void setDataSource(DataSource dataSource){
        this.dataSource = dataSource;
    }

    ...
}
```

```
<!-- DAO Component Scan -->
<context:component-scan base-package="com.lectopia.string.dao" />

<!-- Embedded Database -->
<jdbc:embedded-database id="dataSource1" type="H2">
    <jdbc:script location="classpath:schema.ddl" />
    <jdbc:script location="classpath:init-data.sql" />
</jdbc:embedded-database>

<!-- Embedded Database -->
<jdbc:embedded-database id="dataSource2" type="HSQL">
    <jdbc:script location="classpath:schema.ddl" />
    <jdbc:script location="classpath:init-data.sql" />
</jdbc:embedded-database>
```

## 2-4. Spring Annotation 빈(bean) 관리 – @Autowired

- @Autowired를 이용하여 생성자 또는 필드, 일반 메소드에 의존성 주입 가능
- 동일한 타입의 bean이 여러 개일 경우에는 @Qualifier("이름")으로 빈을 식별
- 생성자가 여러 개일 경우에는 하나의 생성자에만 적용 가능

```
@Component
public class CookbookServiceImpl implements CookbookService {
    private CookbookDao cookbookDao;

    @Autowired
    public CookbookServiceImpl(
        @Qualifier("cookbook") CookbookDao cookbookDao,
        RecipeDao recipeDao) {
        this.cookbookDao = cookbookDao;
        this.recipeDao = recipeDao;
    }

    ...
}
```

생성자 주입

필드 주입

```
@Component
public class CookbookServiceImpl implements CookbookService {
    @Autowired
    @Qualifier("cookbook")
    private CookbookDao cookbookDao;

    @Override
    public List<Cookbook> findAllCookbooks() {
        //
        return cookbookDao.retrieveAll();
    }

    ...
}
```

## 2-5. Spring 빈 – Scope

- Spring 빈의 생성 범위는 Singleton, Prototype, Request, Session
- Spring 빈은 기본적으로 모든 빈을 싱글톤 빈으로 생성  
→ 컨테이너가 제공하는 빈의 인스턴스는 항상 동일
- 컨테이너가 항상 새로운 인스턴스를 반환하게 만들려면 Scope를 Prototype으로 설정

Scope	설명
singleton	Spring 컨테이너 당 하나의 인스턴스 만 생성 (기본)
prototype	컨테이너에 빈을 요청할 때 마다 새로운 인스턴스 생성
request	HTTP 요청 별로 새로운 인스턴스를 생성
session	HTTP 세션 별로 새로운 인스턴스를 생성

### XML 형식으로 선언한 빈의 Scope 설정

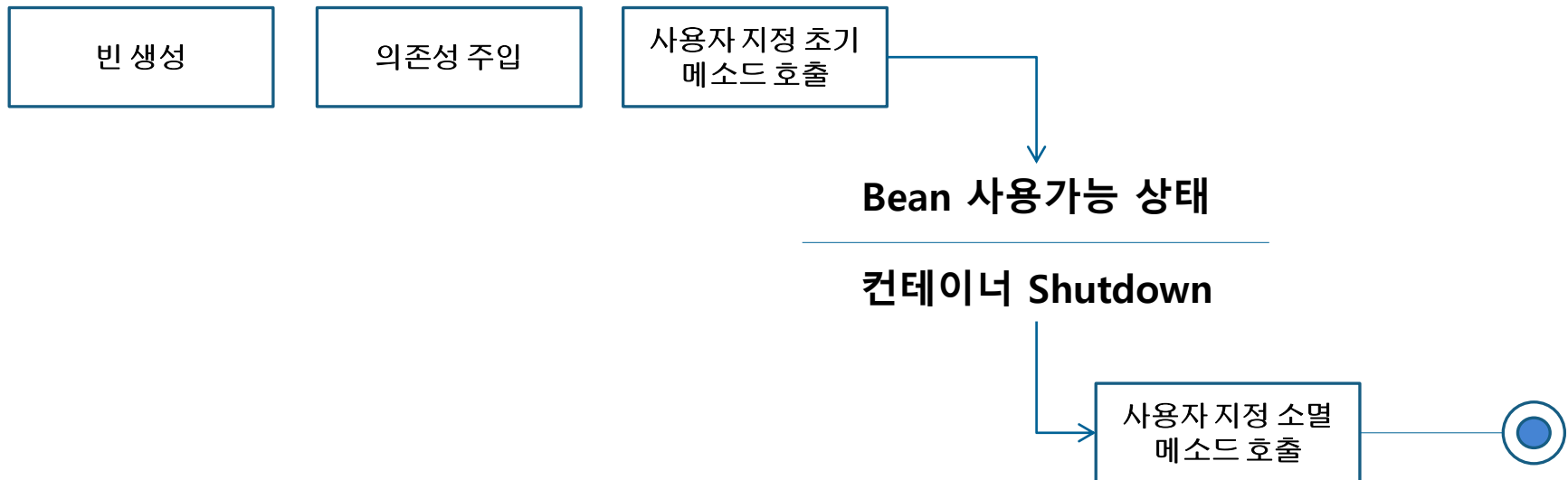
```
<bean id="recipeDao"  
      class="com.lectopia.spring.dao.JdbcRecipeDao"  
      scope="prototype" />
```

### Annotation 형식으로 선언한 빈의 Scope 설정

```
@Repository  
@Scope("prototype")  
public class JdbcRecipeDao implements RecipeDao {  
  
    @Resource  
    private DataSource dataSource;  
  
}
```

## 2-5. Spring 빈 – Scope 빈 생명주기

- Spring IoC 컨테이너는 인스턴스의 생성과 삭제 시점에 호출되는 메소드의 설정이 가능
- 빈의 초기화 메소드는 DI를 통해 모든 프로퍼티가 세팅된 후에만 가능한 초기화를 지원 함
- 빈 소멸 메소드는 컨테이너가 종료될 때 호출되어 종료 전에 처리해야 하는 작업 수행



## 2-5. Spring 빈 – 빈의 초기화 메소드

- XML 방식으로 빈을 설정했다면, 빈의 init-method 속성에 초기화 메소드를 선언
- 어노테이션 방식으로 선언한 경우, 빈 클래스의 초기화 메소드에 @PostConstruct를 추가
- @Bean(init-method)와 Spring의 초기화 메소드 콜백 이용 가능

### @PostConstruct로 사용자 초기화 메소드 설정

```
@Component
public class CookbookServiceImpl implements CookbookService {
    @Autowired
    private CookbookDao cookbookDao;

    @PostConstruct
    public void initCookbookService() {
        System.out.println("@PostConstruct 빈 초기화시 메소드 - start -");
        List<Cookbook> books = cookbookDao.retrieveAll();
        for (Cookbook book : books) {
            System.out.println("@PostConstruct 요리책 목록 : " + cookbook.getName());
        }
        System.out.println("@PostConstruct 빈 초기화시 메소드 - end -");
    }

    ...
}
```

```
@Test
public void testInitCookbookServices() {
    ApplicationContext ac =
        new ClassPathXmlApplicationContext("applicationContext.xml");
}
```

### init-method로 사용자 초기화 메소드 설정

```
<bean id="cookbookService"
      class="com.lectopia.spring.service.impl.CookbookServiceImpl"
      init-method="initCookbookService" />
```

## 2-5. Spring 빈 – 빈의 제거 메소드

- XML 방식으로 빈을 설정했다면, 빈에 destroy-method 속성에 제거 메소드를 선언
- 어노테이션 방식으로 선언한 경우, 빈 클래스의 초기화 메소드에 @PreDestroy 추가
- @Bean(destroy-method)와 Spring의 제거 메소드 콜백 이용 가능

### @PreDestroy로 사용자 제거 메소드 설정

```
@Component
public class CookbookServiceImpl implements CookbookService {
    @Autowired
    private CookbookDao cookbookDao;

    @PreDestroy
    public void destroyCookbookService() {
        System.out.println("@PreDestroy 빈 초기화시 메소드 - start -");
        List<Cookbook> books = cookbookDao.retrieveAll();
        for (Cookbook book : books) {
            System.out.println("@PostConstruct 요리책 목록 : " + cookbook.getName());
        }
        System.out.println("@PreDestroy 빈 초기화시 메소드 - end -");
    }

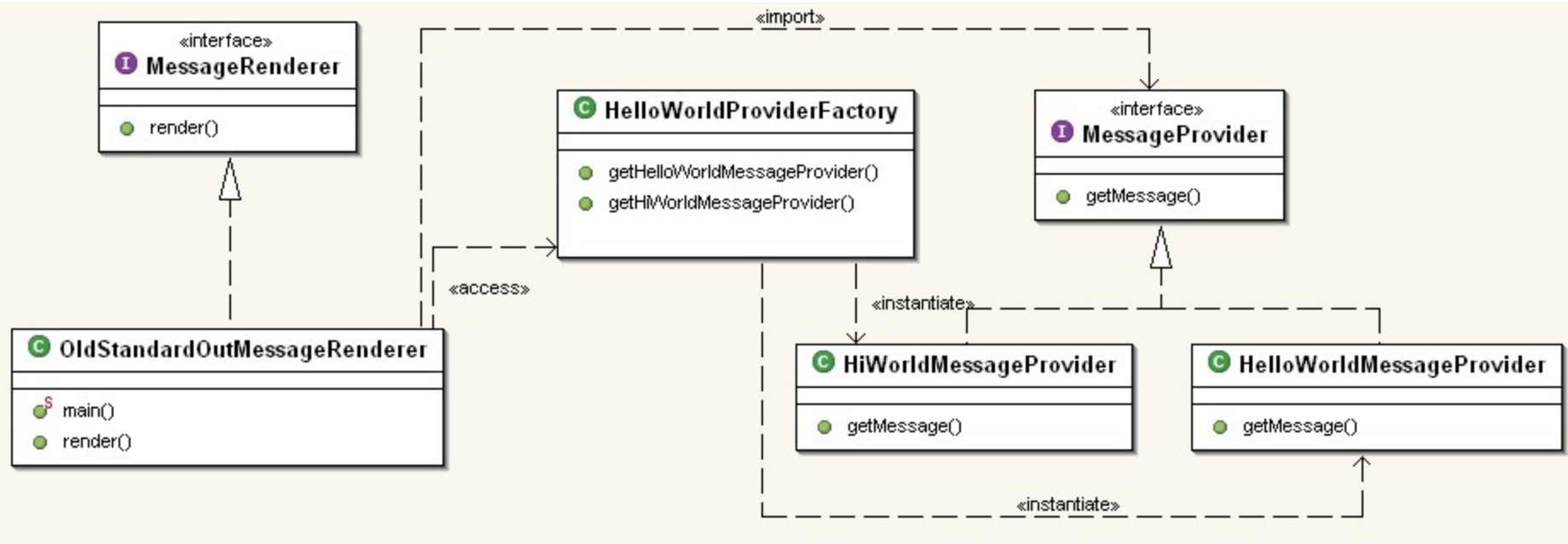
    ...
}
```

### destroy-method로 사용자 제거 메소드 설정

```
<bean id="cookbookService"
      class="com.lectopia.spring.service.impl.CookbookServiceImpl"
      destroy-method="destroyCookbookService" />
```

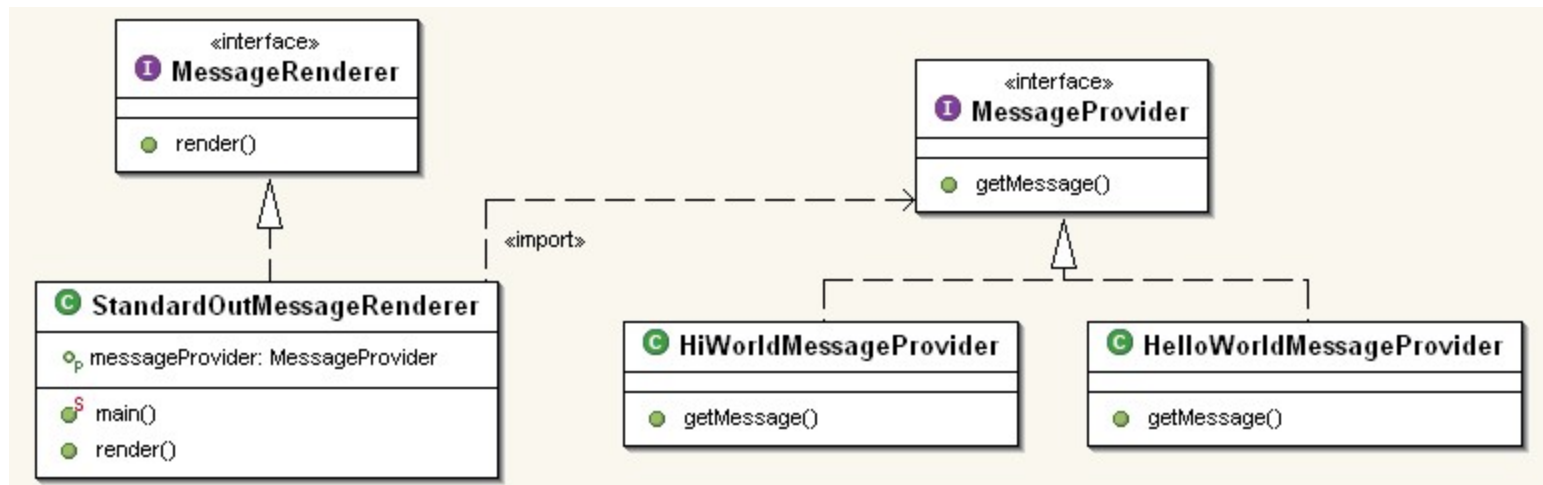
## 2-6. 실습

### 일반적인 개발 방법



## 2-6. 실습

Spring의 BeanFactory를 적용했을 때의 개발 방법





## 3-1. AOP 개요

ex. 메소드 실행 중 예외가 발생하는 이력을 저장하기 위한 코드

- 메소드 수행 중 예외가 발생하면 예외 발생 이력을 저장하기 위해 업무로직과 무관한 코드가 존재

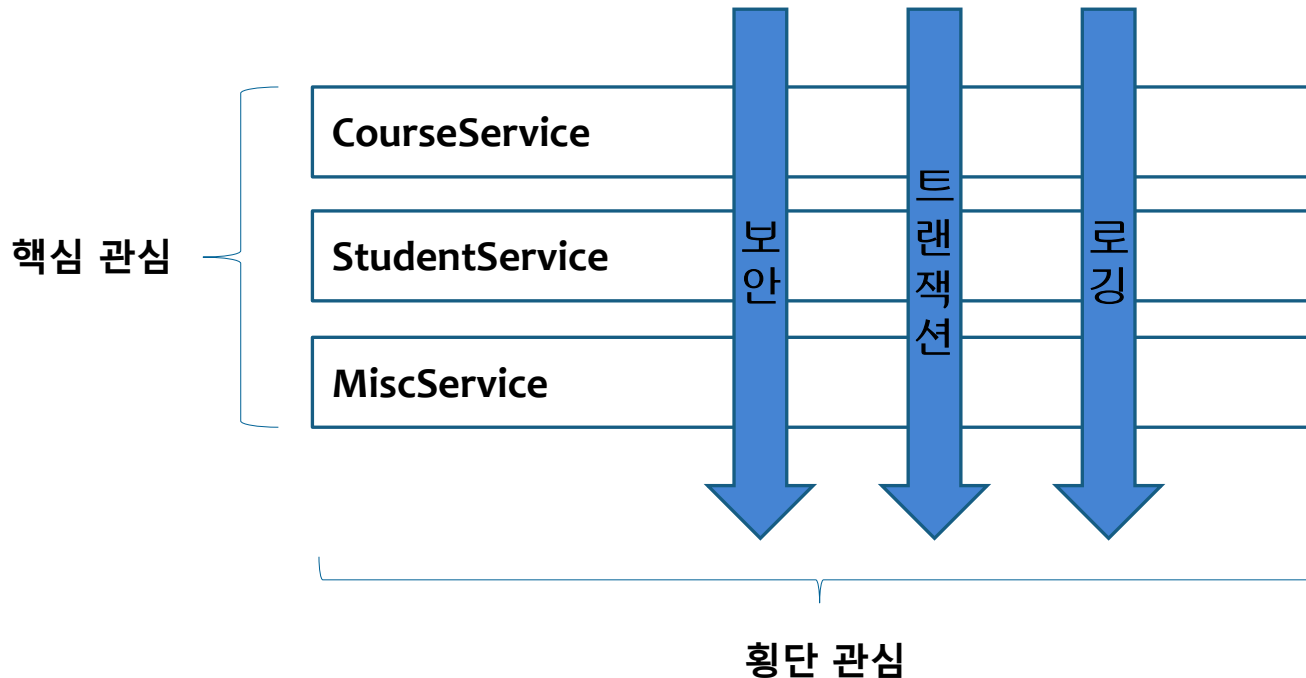
```
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    private UserDao userDao;

    @Autowired
    private ReportDao reportDao;

    @Overried
    public void registerUser(User user) {
        try {
            userDao.create(user);
        } catch (Exception ex) {
            ex.printStackTrace();
            reportDao.create("UserServiceImpl.registerUser()", e.getMessage());
        }
    }
}
```

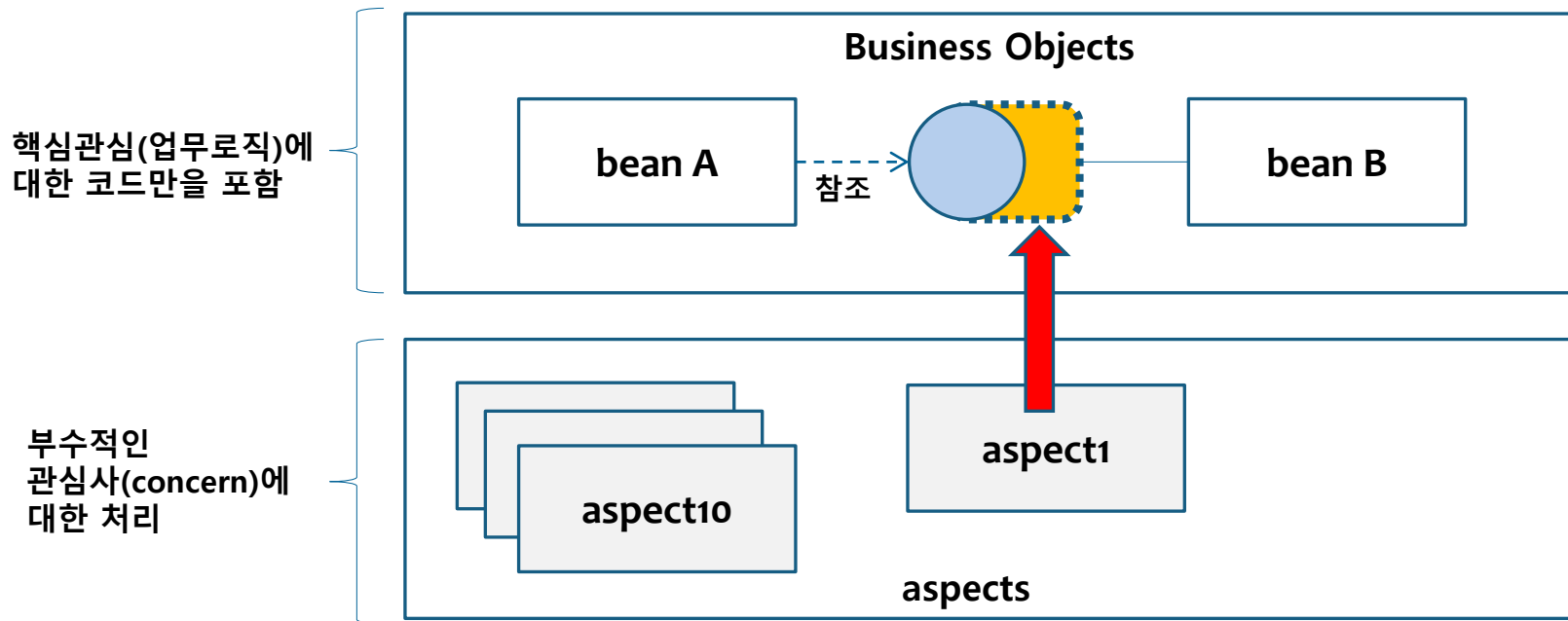
### 3-1. AOP 개요 – 횡단관심 (Cross-cutting concern)

- 복잡한 애플리케이션은 보안, 로깅, 트랜잭션과 같은 다양한 Concern 이 존재
- 횡단관심은 이런 Concern이 시스템을 구성하는 모듈의 여기저기에 흩어져 있는 현상을 말함
- 각각의 모듈의 주요 관심사는 특정 도메인에 대해서 서비스를 제공하는 것  
→ 이들 각각의 모듈들은 보안이나 트랜잭션 관리와 같은 유사한 보조적인 (Ancillary)기능을 필요로 함



### 3-1. AOP 개요 – AOP

- Aspect Oriented Programming을 줄여서 AOP
- 새로운 특성을 적용하려는 클래스의 변경 없이 해당 기능을 어디에 어떻게 적용할지를 선언적으로 정의
- Cross-cutting concern은 aspect라고 하는 특별한 객체로 모듈화
- 각각의 관심사(Concern)에 대한 로직을 여러 코드에 걸쳐 작성하지 않고, 하나의 모듈로 만들어 사용

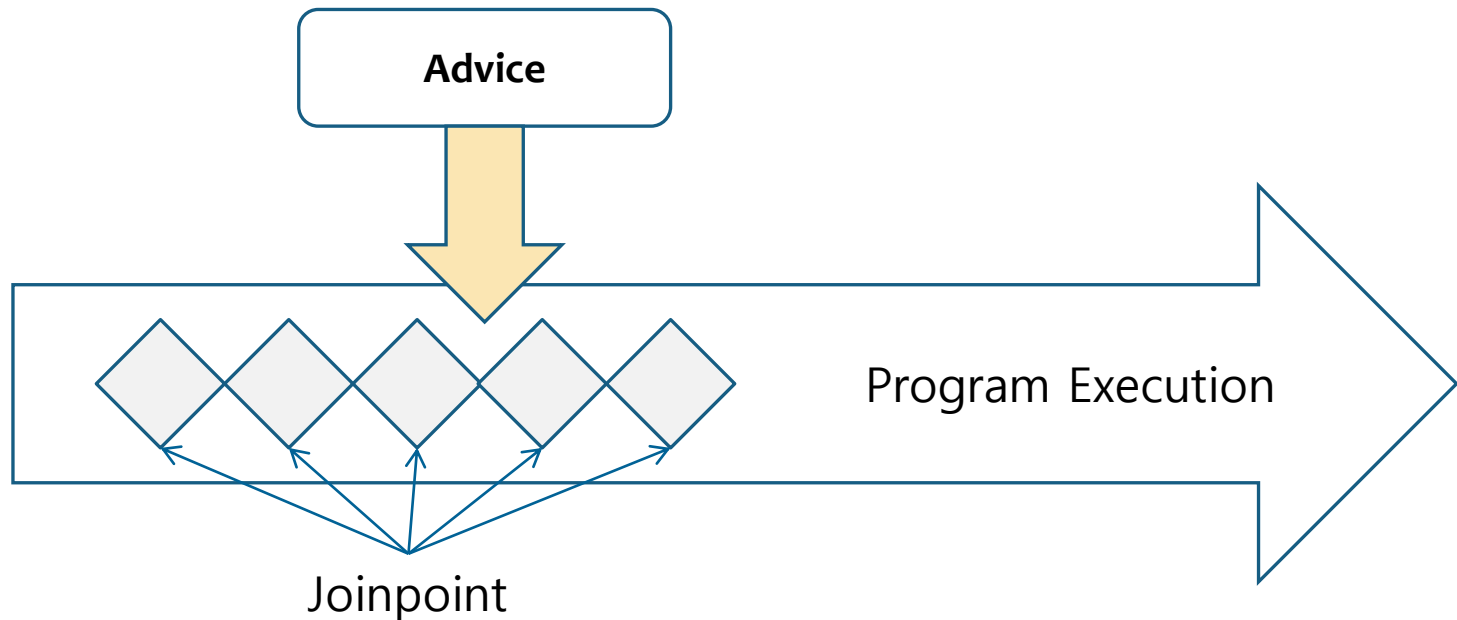


### 3-1. AOP 개요 – AOP의 용어

AOP 용어	설명
Aspect	Aspect는 횡단관심사의 동작과 그 횡단관심사를 적용하는 소스 코드에서의 위치를 모아 둔 것 하나 이상의 Advice(동작)과 Pointcut(동작을 적용하는 조건)의 조합
Joinpoint	Advice가 실행하는 동작을 끼워 넣을 수 있는 시점 Spring의 조인포인트는 메소드가 호출될 때와 메소드가 원래 호출한 곳으로 돌아갈 때
Advice	Joinpoint에서 실행되는 코드로 보안, 로깅, 트랜잭션 관리 등의 코드를 기술
Pointcut	Pointcut은 Joinpoint와 Advice의 중간에 있으면서 Advice를 적용할 조인포인트를 선별
Target	Advice가 수행될 객체, 특정 행위를 추가하기 원하는 객체
Proxy	대상(target) 객체에 Advice를 적용한 후에 생성되는 객체 클라이언트 객체에 한정해서 대상 객체 (pre-AOP)와 proxy 객체(post-AOP)는 동일
Weaving	proxy되는 객체를 생성하는 대상(target) 객체에게 aspect를 적용하는 과정 (process) 대상 객체의 생명주기 내에서 여러 지점에 발생할 수 있음

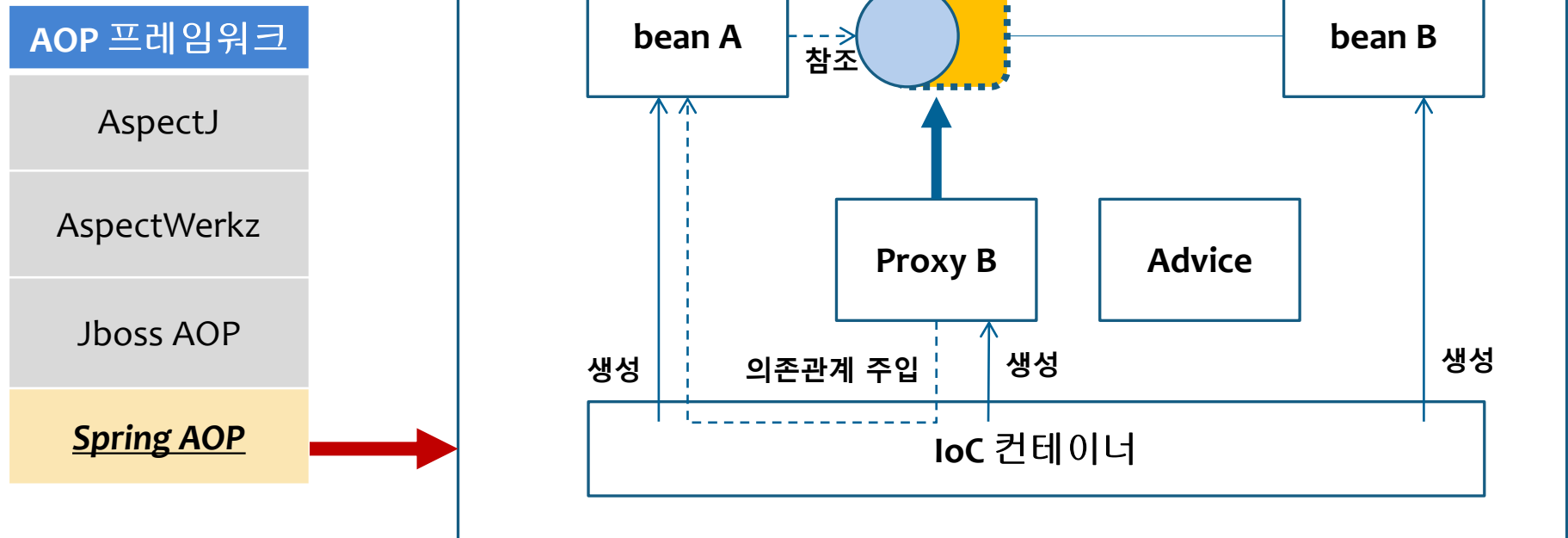
### 3-1. AOP 개요 – AOP의 용어

- 프로그램이 처리되는 과정은 일련의 메소드 호출로 이루어짐
- AOP는 인터셉트만 제공하는 것이 아니라 Pointcut(조건)으로 매치되는 Joinpoint를 제공
- Pointcut으로 Advice가 핵심기능과 분리된 독립적인 모듈이 될 수 있게 함
- Aspect의 기능(Advice)은 하나 이상의 Joinpoint 시점에 프로그램의 실행으로 구성 (Weave)



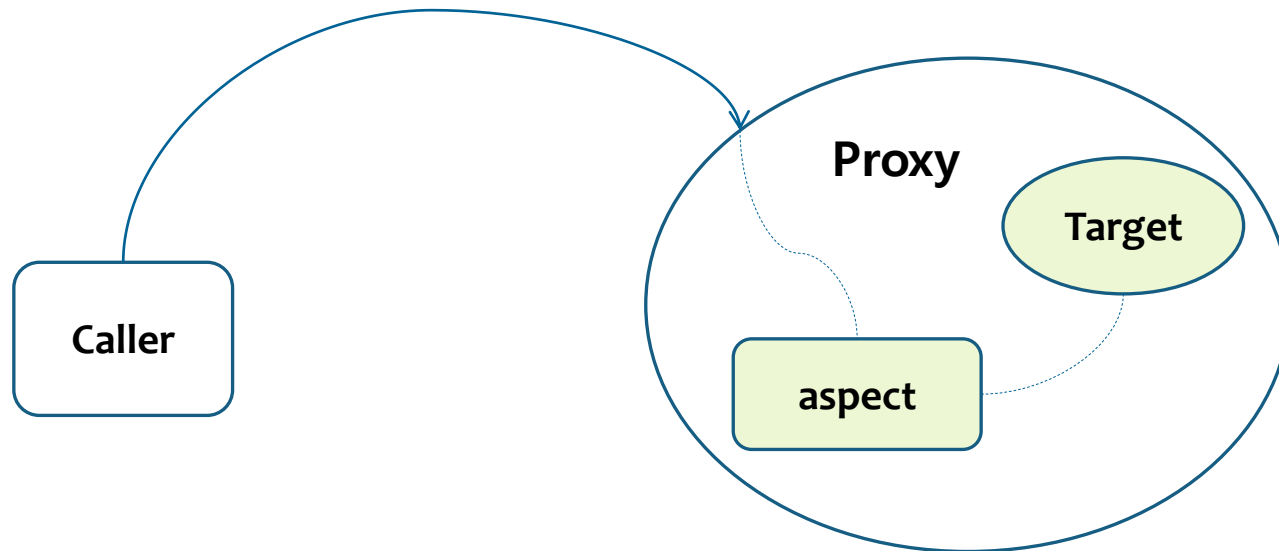
## 3-2. Spring AOP

- Spring은 동적 Proxy를 기반으로 하므로 메소드 Joinpoint만을 지원하며, 세밀한 advice 구성을 할 수 없음
- AspectJ나 Jboss의 경우 생성자 Joinpoint나 필드 Joinpoint를 추가로 지원
- Spring은 JEE 구현에 대한 프레임워크를 제공하는데 초점을 맞추기 때문에 메소드 Interception 만으로도 충분 함  
→ 그 이상의 기능의 AOP가 필요하면 AspectJ와 Spring AOP를 통합해서 사용해야 함



## 3-2. Spring AOP

- Aspect 들은 실행 시 Spring이 관리하는 빈(bean)에 Proxy 클래스를 감싸서(Wrapping) 구성
- Spring은 해당 Proxy로 감싸지는 빈이 어플리케이션에서 필요할 때까지 Proxy로 감싸지는 객체를 생성하지 않음
- Spring은 실행시 Proxy를 생성하기 때문에 Spring의 AOP의 aspect를 구성하는 특별한 컴파일러가 필요 없음
- ApplicationContext를 사용하는 경우 Proxy로 감싸지는 객체는 BeanFactory로부터 모든 빈이 로딩된 후 생성

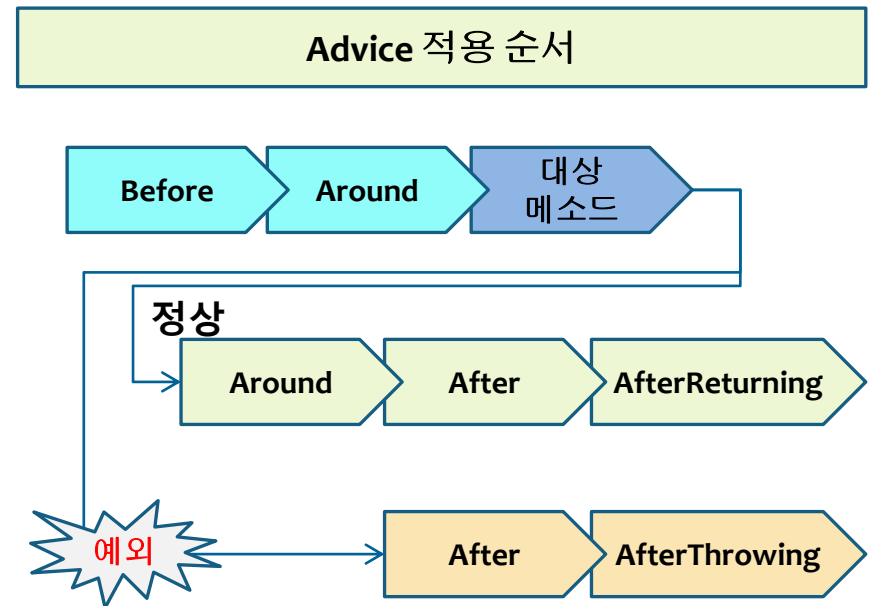


- Spring의 aspect는 대상(target) 객체를 감싸는 (wrap) proxy로 구현됨
- 이 Proxy는 메소드 호출을 처리, 추가적인 aspect 로직을 수행. 대상 메소드 호출
- aspect 로직이 수행되는 시점은 Proxy가 메소드 호출을 가로채는 시간과 대상 빈의 메소드를 호출하는 시간 사이

### 3-2. Spring AOP – Spring에서 제공하는 Advice

- Spring은 AOP 프록시 안에서 Advice가 적용되는 시점에 따라 5가지 타입의 Advice를 제공
- 트랜잭션과 같은 대표적인 Advice는 이미 제공되므로 구현할 필요 없음
- 대상 메소드가
  - 정상 실행: Before → Around → 대상 메소드 → Around → After → AfterReturning 순으로 적용
  - 예외 발생: Before → Around → 대상 메소드 → After → AfterThrowing 순으로 적용

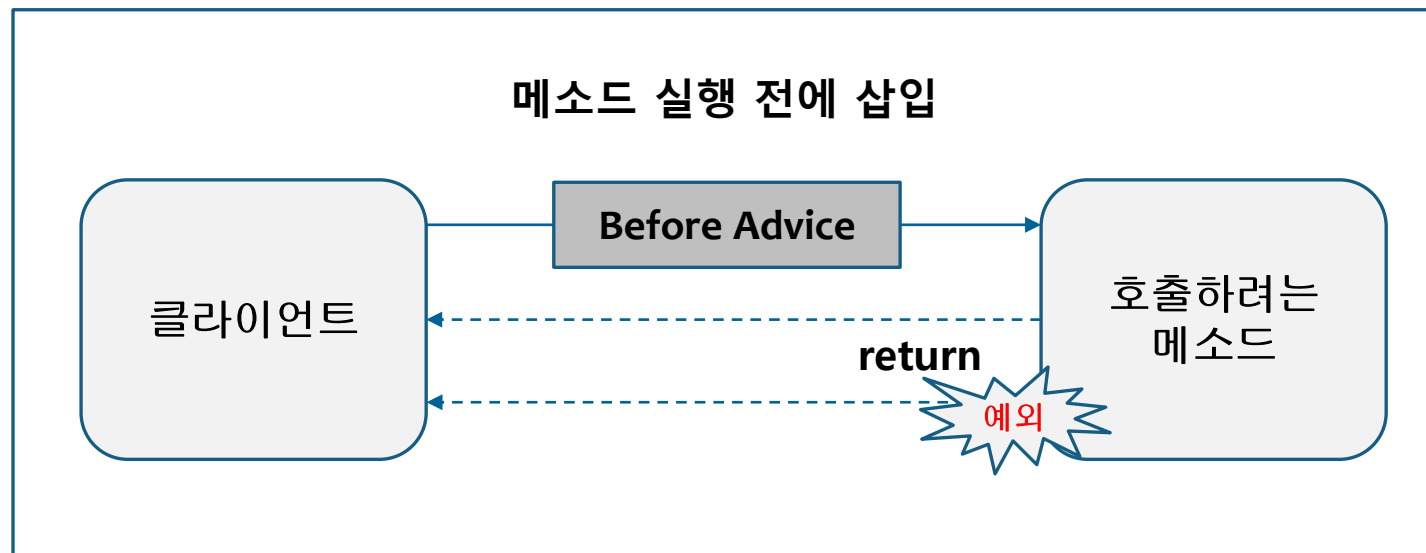
Advice 타입	설명
Before	Joinpoint 앞에서 실행할 Advice
After	Joinpoint 뒤에서 실행할 Advice
AfterReturning	Joinpoint 가 정상 종료한 다음 실행되는 Advice
Around	Joinpoint 앞뒤에 실행되는 Advice
AfterThrowing	Joinpoint 에서 예외가 발생했을 때 실행되는 Advice





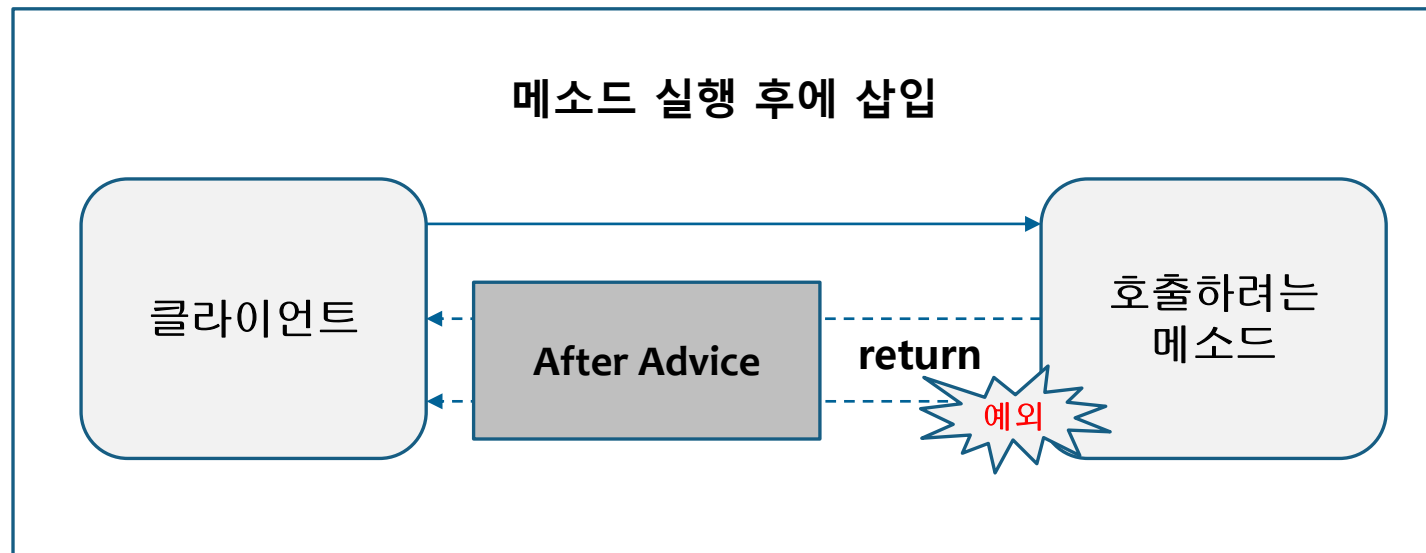
## 3-2. Spring AOP – Before Advice

- Before Advice는 조인포인트 이전에만 실행되는 Advice
- 호출하려는 메소드에 대해 제어할 수 없음
  - 예외가 발생하지 않는 한 타킷 오브젝트 메소드는 항상 실행
- 메소드에 @Before 어노테이션을 추가하여 Advice를 구현



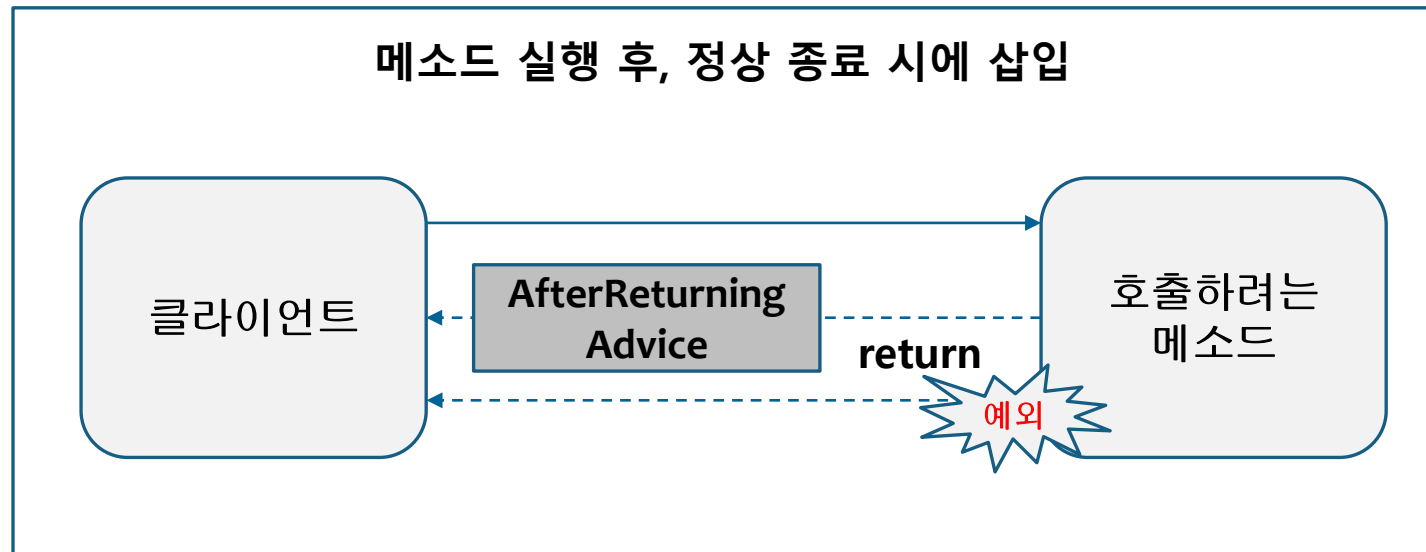
## 3-2. Spring AOP – After Advice

- After Advice는 타깃 오브젝트의 메소드가 종료되는 시점에 실행되는 Advice 타입
- 타깃 오브젝트의 메소드가 정상종료와 예외발생 모두 실행  
→ 타깃 오브젝트의 메소드의 return 값 또는 예외에 대해 직접 전달 받을 수 없음
- 메소드에 @After 어노테이션을 추가하여 Advice를 구현



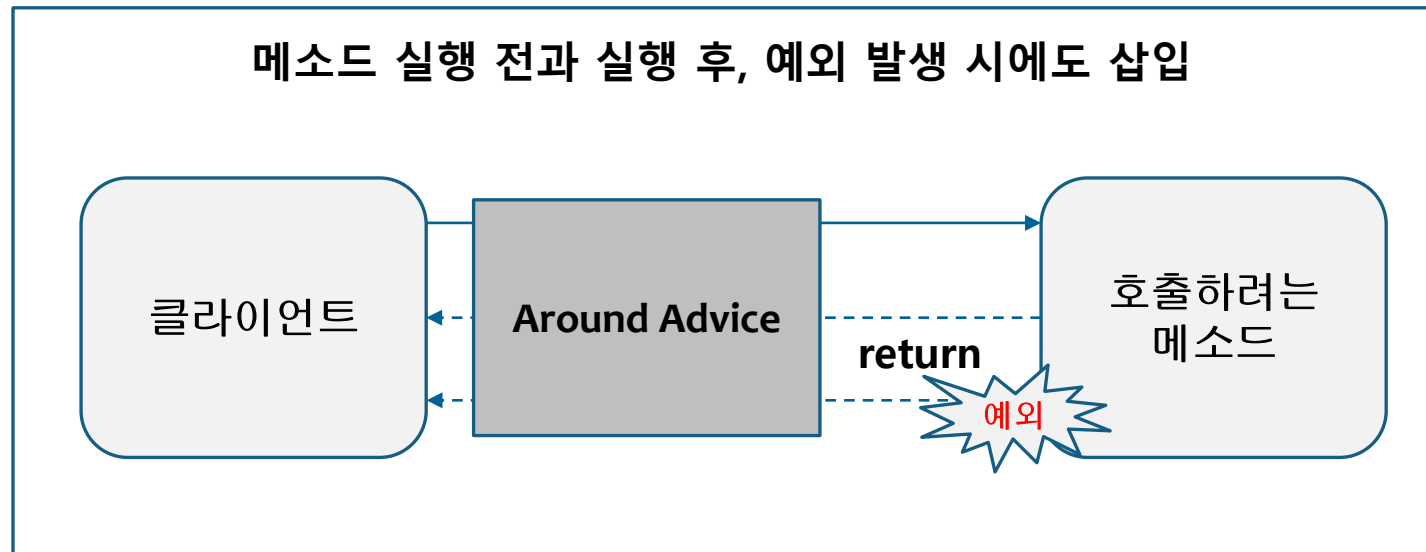
## 3-2. Spring AOP – AfterReturning Advice

- AfterReturning Advice는 타겟 오브젝트 메소드가 예외 없이 정상 종료 시 실행되는 Advice
- 메소드에 @AfterReturning 어노테이션을 추가하여 Advice를 구현
- 정상 종료 후 실행되는 Advice이므로 메소드의 리턴 값을 참조 할 수 있음
- 리턴 값을 변경할 수 없기 때문에, 변경을 원할 경우에는 Around Advice를 사용



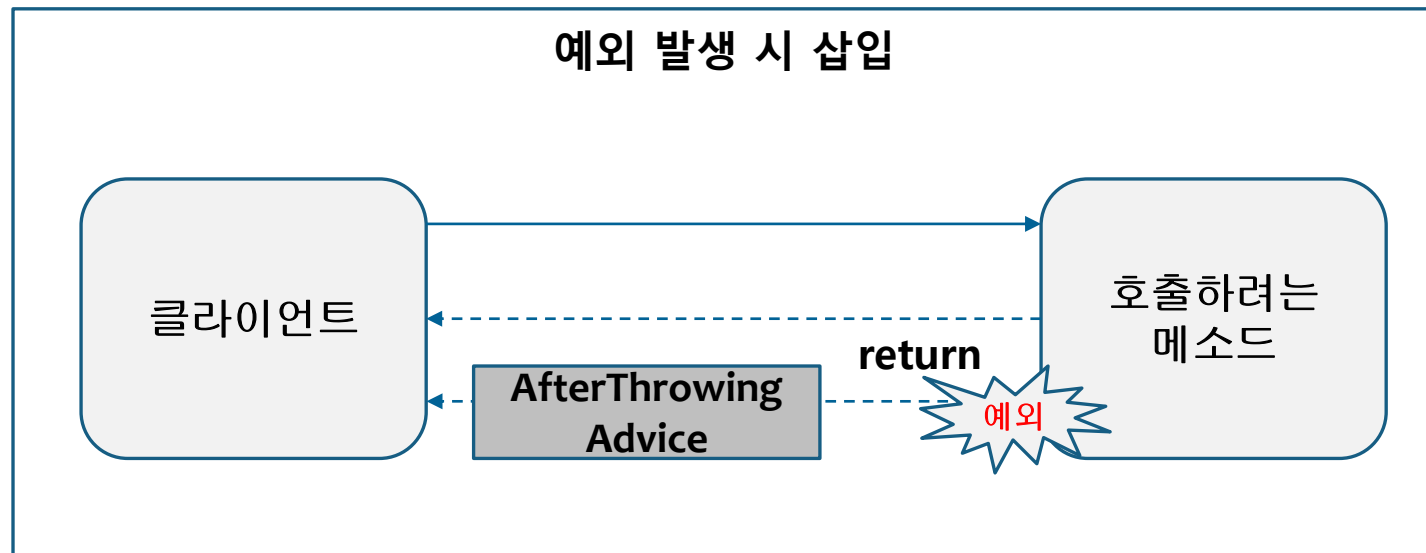
## 3-2. Spring AOP – Around Advice

- Around Advice는 프록시를 통해서 타깃 오브젝트의 메소드가 호출되는 전 과정을 모두 제어할 수 있는 Advice
- 타깃 오브젝트의 메소드의 실행까지 제어가 가능한 가장 강력한 Advice
- 다른 Advice와 달리 대상 메소드의 호출을 Advice 안에서 처리해야 함
- 반드시 `ProceedingJoinPoint`의 `proceed()`를 호출하여 타깃 메소드를 실행시켜야 함



## 3-2. Spring AOP – AfterThrowing Advice

- AfterThrowing Advice는 타겟 오브젝트 메소드 수행 중 예외가 발생하는 경우 실행되는 Advice 타입
- 메소드에 @AfterThrowing 어노테이션을 추가하여 Advice를 구현  
→ 포인트컷 설정인 "execution(throwing="예외변수명")으로 특정 예외만 처리하도록 설정 가능
- \* AfterThrowing Advice로 처리하려는 예외를 상속한 예외는 모두 동작



### 3-3. Spring AOP 적용 – Dependency 추가

- Spring AOP를 적용하기 위해서 Maven dependency에 aspect관련 라이브러리를 추가
- Spring AOP를 적용할 대상 메소드를 작성하거나 식별
- Aspect를 작성하고 빈으로 등록

```
<dependencies>
  <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>${org.aspect-version}</version>
  </dependency>
  <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>${org.aspect-version}</version>
  </dependency>
  ...
</dependencies>
```

### 3-3. Spring AOP 적용 – AOP 적용 대상 메소드 작성

- AOP를 적용할 대상 메소드를 작성
- 예외가 발생시 예외 발생 이력을 저장하기 위해 업무로직과는 무관한 로직이 존재하는 코드를 수정
- 업무로직과 무관한 예외가 발생 시 예외 발생 이력을 저장하는 로직을 제거
- 제거한 로직은 Aspect로 작성

#### 업무로직 + Aspect로직

```
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    private UserDao userDao;

    @Autowired
    private ReportDao reportDao;

    @Overried
    public void registerUser(User user) {
        try {
            userDao.create(user);
        } catch (Exception ex) {
            ex.printStackTrace();
            reportDao.create("UserServiceImpl.registerUser()",
                             e.getMessage());
        }
    }
}
```

#### 업무로직 이외의 로직은 제거

```
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    private UserDao userDao;

    @Overried
    public void registerUser(User user) {
        User registdUser = userDao.retrieve(user.getUserId());
        if (registdUser == null) {
            throw new MyExcpetion("이미 등록된 사용자 ID.");
        }

        userDao.create(user);
    }
}
```

### 3-3. Spring AOP 적용 – Aspect 작성

- 예외가 발생 시 예외 발생 이력을 저장하는 로직에 대한 Aspect를 작성  
→ AfterThrowing Advice로 작성
- <aop:aspect-autoproxy>으로 어노테이션으로 선언한 aspectj를 사용할 수 있도록 함
- XML 방식으로 선언할 경우, AspectJ를 빈으로 등록하고 <aop:config>를 이용하여 선언

```
@Aspect
@Component
public class TransactionAspect {
    @Autowired
    private ReportDao reportDao;

    @AfterThrowing(value="execution(* *.register*(..))", throwing="ex")
    public void registerTransactionException(JoinPoint joinpoint, Throwable ex) {
        Signature signature = joinpoint.getSignature();
        ...
        reportDao.create(signature.getDeclaringTypeName() + "." +
            joinpoint.getSignature().getName(), ex.getMessage());
    }
}
```

```
<content:component-scan base-package="com.lectopia.spring.common.aop" />
<aop:aspectj-autoproxy />
...
```

어노테이션으로 선언한 aspectj

XML 방식으로 선언한 aspectj

```
<bean id="transactionAspect"
      class="com.lectopia.spring.common.aop.TransactionAspect" />
<aop:config>
    <aop:aspect ref="transactionAspect">
        <aop:after-throwing pointcut="execute(* *.register*(..))"
            method="registerTransactionException" />
    </aop:aspect>
</aop:config>
```