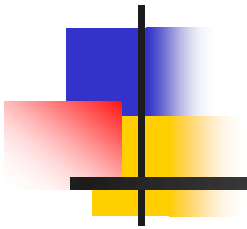


spring 1강-스프링 프레임워크 소개

양 명 숙

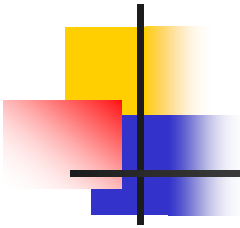
[now4ever7@gmail.com]

목차

- 프레임워크
- 스프링 프레임워크
- 종속객체 주입(DI : Dependency Injection)
- 애스펙트 지향 프로그래밍 (AOP: Aspect-Oriented Programming)

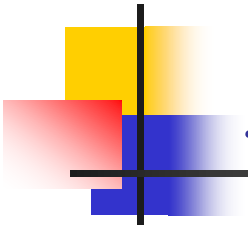
1. 자바 프로젝트 - 라이브러리쪽 build path - external lib 해서 각종 스프링 라이브러리 호출

2. 완료 후 application Context.xml을 src에 붙이기



프레임워크

- 프레임워크
  - 사전적 의미 – 뼈대, 틀, 골자
  - 기존에 개발자들이 개발을 하면서 나왔던 많은 표준안과 구조를 정리한 라이브러리
  - **어떤 문제영역을 해결하기 위한 잘 설계된 일반적인, 재사용 가능한 모듈** + 사용자가 Framework를 확장하여 비즈니스 요구사항을 만족시키는 완전한 애플리케이션 소프트웨어를 완성시키는 작업 => 완전한 애플리케이션 소프트웨어
  - **재사용되는 공통된 부분**은 Framework로 구현되어 다른 사람이 그 내부를 가공 없이 이용하도록 제공됨



프레임워크

- 일반적으로 라이브러리나 프레임워크는 특정 업무분야나 한 가지 기술에 특화된 목표를 가지고 만들어짐
  - 그래서 **프레임워크는 애플리케이션의 특정 계층에서 주로 동작하는 한 가지 기술 분야에 집중**됨
- 하지만 스프링은 애플리케이션 프레임워크라는 특징을 가짐
  - 애플리케이션 프레임워크는 특정 계층이나 , 기술, 업무 분야에 국한되지 않고 **애플리케이션의 전 영역을 포괄하는 범용적인 프레임워크**를 말한다
  - 애플리케이션 프레임워크는 애플리케이션 개발의 전 과정을 빠르고 편리하며 효율적으로 진행하는데 일차적인 목표를 두는 프레임워크다

스프링 프레임워크

- 스프링
  - 기업용 애플리케이션을 만들기 위한 다양한 기능을 제공하는 프레임워크
  - JEE(Java Enterprise Edition) 가 제공하는 기능들을 대신 제공하며, 다양한 기능과 특유의 편의성으로 인해 현재는 자바에서 가장 표준적인 프레임워크로 자리 잡았다
  - JEE의 다양한 기능을 제공하면서도 EJB가 제시하던 어려운 개발 방식이 아닌 일반적인 자바 객체(POJO:Plain Old Java Object) 를 사용함

- 엔터프라이즈 어플리케이션은 표준화되고 모듈화된 컴포넌트로 구성된다.
- J2EE는 개발자에게 복잡한 개발 과정 없이 해당 컴포넌트를 자동적으로 처리해주는 환경을 제공한다.
- 트랜잭션 지원이나 보안, 분산 컴퓨팅 등 지원
- 컴포넌트 – 작은 의미의 프로세스로서 전체 프로젝트에서 독립적으로 존재할 수 있는 구성 요소를 말함
  - 기존의 시스템을 유지하면서 추가에 편리

- **EJB :** 표준화된 분산 컴퓨팅 프로토콜인 **OMA**를 기반으로하고, **RMI** 의 장점을 살려서 **SUN**에서 제공하는 분산 프로그램 방식

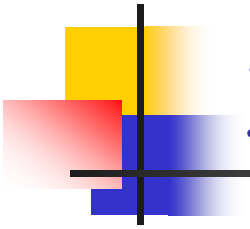
스프링 프레임워크

- 자바
  - 서블릿 출현으로 웹 기반 어플리케이션 구축에 자바가 사용되기 시작
  - 트랜잭션, 보안 등을 제공하는 EJB가 제공되면서 자바는 **엔터프라이즈 어플리케이션을 구축**하는데 필요한 기본 기술로 자리잡아 감
  - 하지만, EJB 2 버전까지는 개발 과정이 불편-개발 속도를 향상시키는 데 한계
    - 또한 EJB 는 반드시 EJB 스펙에 정의된 인터페이스에 따라 코드를 작성하도록 제약하고 있기 때문에 개발자가 기존에 작성한 POJO(Plain Old Java Object)를 변경해야 한다는 단점
- Rod Johnson – EJB를 사용하지 않고 엔터프라이즈 어플리케이션을 개발하는 방법 소개 => 스프링 프레임워크의 모태가 됨
  - 로드 존슨이 "Expert One-on-One:J2EE Design and Development" 책을 통해 소개
- 스프링 프레임워크 – 현재는 단순한 웹 어플리케이션 구축에서부터 금융 시스템과 같은 복잡한 엔터프라이즈 어플리케이션까지 사용범위가 확대되었음

스프링 프레임워크

- 스프링 프레임워크
  - 스프링은 가벼운 DI 및 AOP기반 컨테이너이자 프레임워크
  - 엔터프라이즈 어플리케이션에서 필요로 하는 기능을 제공하는 경량 프레임워크
  - J2EE(Java Enterprise Edition)가 제공하는 다수의 기능을 지원하고, DI(Dependency Injection)나 AOP(Aspect Oriented Programing)와 같은 기능도 지원
  - 애플리케이션의 전 영역을 포괄하는 범용적인 프레임워크

https://spring.io/tools3/sts/all 에서 다운받기



자바 개발 간소화

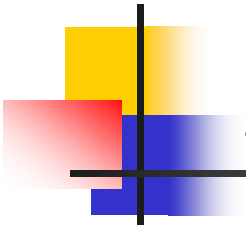
- 스프링(Spring)
  - 로드 존슨이 "Expert One-on-One:J2EE Design and Development" 책을 통해 소개한 오픈 소스 프레임워크
    - EJB를 사용하지 않고 엔터프라이즈 어플리케이션을 개발하는 방법 소개
  - 엔터프라이즈 애플리케이션 개발의 복잡함을 겨냥해 만들어졌다
  - 스프링은 EJB로만 할 수 있었던 작업을 평범한 자바빈을 사용해서 할 수 있게 해줌
  - 스프링의 기본 임무 : 자바 개발 간소화
    - 자바 개발을 폭넓게 간소화

스프링 프레임워크

- 스프링이 제공하는 주요 기능과 특징
  - 스프링은 경량 컨테이너
    - 스프링은 자바 객체를 담고 있는 컨테이너
    - 스프링 컨테이너는 이들 **자바 객체의 생성, 소멸과 같은 라이프 사이클을 관리**하며, 스프링 컨테이너로부터 **필요한 객체를 가져와 사용**할 수 있음
  - 스프링은 DI(Dependency Injection) 지원
    - 설정파일이나 어노테이션을 통해서 객체 간의 의존 관계를 설정할 수 있도록 함
    - 객체는 의존하고 있는 객체를 직접 생성하거나 검색할 필요가 없음
  - 스프링은 AOP(Aspect Oriented Programming)를 지원함
    - 스프링은 자체적으로 AOP를 지원하고 있기 때문에 **트랜잭션이나 로깅, 보안**과 같이 여러 모듈에서 공통으로 필요로 하지만 실제 모듈의 핵심은 아닌 기능들을 분리해서 각 모듈에 적용할 수 있음

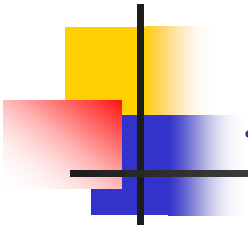
스프링 프레임워크

- 스프링은 POJO(Plain Old Java Object)를 지원함
  - 스프링 컨테이너에 저장되는 자바 객체는 특정한 인터페이스를 구현하거나 클래스를 상속받지 않아도 됨
  - 기존에 작성한 코드를 수정할 필요 없이 스프링에 사용할 수 있음
- 스프링은 트랜잭션 처리를 위한 일관된 방법을 제공함
  - JDBC를 사용하든, JPA를 사용하든, 컨테이너가 제공하는 트랜잭션을 사용하든, 설정 파일을 통해 트랜잭션 관련 정보를 입력하기 때문에, 트랜잭션 구현에 상관없이 동일한 코드를 여러 환경에서 사용할 수 있음
- 스프링은 영속성(Persistence)과 관련된 다양한 API를 지원
  - JDBC를 비롯하여 iBatis/mybatis, 하이버네이트, JPA 등 데이터베이스 처리를 위해 널리 사용되는 라이브러리와의 연동을 지원



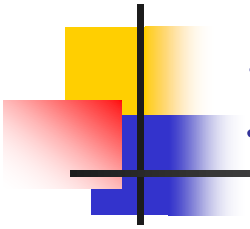
스프링 프레임워크

- 스프링은 다양한 API에 대한 연동을 지원
  - JMS, 메일, 스케줄링 등 엔터프라이즈 어플리케이션을 개발하는 데 필요한 다양한 API를 설정 파일과 어노테이션을 통해서 쉽게 사용할 수 있도록 지원
- 스프링은 자체적으로 MVC 프레임워크를 제공
  - 스프링만 사용해도 MVC 기반의 웹 어플리케이션을 쉽게 개발할 수 있음
- 스트럿츠2, JSF 와 같은 프레임워크와의 연동을 지원



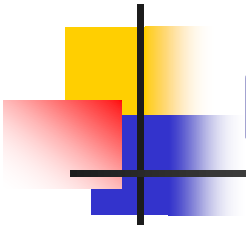
스프링 개요

- 스프링의 주요 기능
  - JEE가 제공하는 다양한 기능 제공
  - 종속객체 주입(DI : Dependency Injection)
  - 애스펙트 지향 프로그래밍 (AOP: Aspect-Oriented Programming)
    - => 애플리케이션 객체 간의 결합도를 줄여줌



자바 개발 간소화

- 스프링의 4가지 주요 전략
  - POJO를 이용한 가볍고 비 침투적인 개발
  - DI 와 인터페이스 지향을 통한 느슨한 결합도
  - 애스펙트와 공통 규약을 통한 선언적 프로그래밍
  - 애스펙트와 템플릿을 통한 상투적인 코드 축소



POJO의 힘

- EJB2, struts, 웹워크, 태피스트리의 초기 버전 등은 간단한 자바 클래스가 아니라 자신만의 프레임워크를 강요했다
- 이런 무거운 프레임워크는 개발자에게 불필요한 코드가 흩어져 있는 클래스 작성을 강요하고, 프레임워크에 묶어두고, 테스트 수행도 어려웠다
- 스프링 – API를 이용하여 애플리케이션 코드의 분산을 가능한 막는다
  - 스프링은 스프링에 특화된 인터페이스 구현이나 스프링에 특화된 클래스 확장을 거의 요구하지 않음
  - 클래스에 스프링의 annotation 이 붙는 경우 외에는 POJO 임

EJB2.1 – 침략적인 API

일반적으로 필요하지도 않은 메소드를 강제로 구현하게 함

```
package com.ejb.session;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class  HelloWorldBean implements SessionBean
{
	public void ejbActivate(){}
	public void ejbPassivate(){}
	public void ejbRemove(){}
	public void setSessionContext(SessionContext ctx){}
	public void ejbCreate(){}

	//EJB의 핵심 비즈니스 로직
	public String sayHello()
	{
		return "Hello World!";
	}
}
```

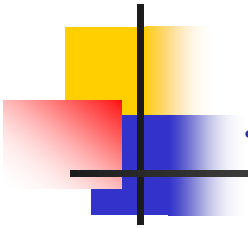
스프링이 관리하는 빈으로 다시 작성

```
package com.spring;

public class  HelloWorldBean
{
    public String sayHello()
    {
        return "Hello World!";
    }
}
```

- HelloWorldBean은 구현하지도, 확장하지도 않았고, spring API에서 어떤것도 import하지 않았다

- HelloWorldBean은 군살이 없고, 평범하며, 모든 구문은 POJO다

- 스프링이 POJO에 힘을 불어 넣는 방법 중 하나는 DI를 활용한 조립
- DI 를 통해 애플리케이션 객체 상호간의 결합도를 낮춘다.



스프링 프레임워크 설치

- 스프링 프레임워크 3.1 버전
  - http://www.springsource.org
  - https://spring.io/

  - spring-framework-3.1.1.RELEASE.zip
  - spring-framework-4.0.1.RELEASE.zip

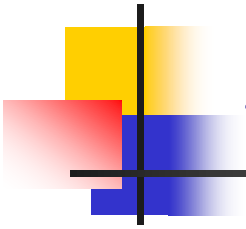
- 스프링 3 버전부터는 자바 5 이상의 버전 필요

종속 객체 주입(DI)

- 종속 객체 주입(DI : Dependency Injection)
  - 실제 어플리케이션에서는 두 개 이상의 클래스가 서로 협력하여 비즈니스 로직을 수행함
  - 이때 각 객체는 협력하는 객체에 대한 레퍼런스(종속객체)를 얻어야 하고, 그 결과 결합도가 높아지고, 테스트하기 힘든 코드가 만들어지기 쉽다.

ProductDAO 에 강하게 결합됨

```
public class ProductService {
        private ProductDAO dao;
        public ProductService(){
                dao = new ProductDAO();
        }//생성자

        public ProductBean detailProduct(int pdNo){
                return dao.detailProduct(pdNo);
        }
```



종속 객체 주입

- DI를 이용 – 객체는 시스템에서 각 객체를 조율하는 제 3자에 의해 생성 시점에 종속객체가 부여됨
  - 객체는 종속객체를 생성하거나 얻지 않음
  - 종속객체는 종속객체가 필요한 객체에 주입됨

종속 객체 주입(DI : Dependency Injection)

- 종속 객체 주입(DI : Dependency Injection)
  - 객체 간의 의존 관계를 객체 자신이 아닌 외부의 조립기가 수행해 준다는 개념
  - 어플리케이션은 규모에 따라 수십에서 수백, 수천 개의 객체로 구성
  - 웹 어플리케이션의 경우 – 클라이언트의 요청을 받아주는 컨트롤러 객체, 비즈니스 로직을 수행하는 서비스 객체, 데이터 접근을 수행하는 DAO 객체 등으로 구성됨
  - 이런 객체들은 독립적으로 기능을 수행하기보다는, 서로 의존하여 어플리케이션의 기능을 구현

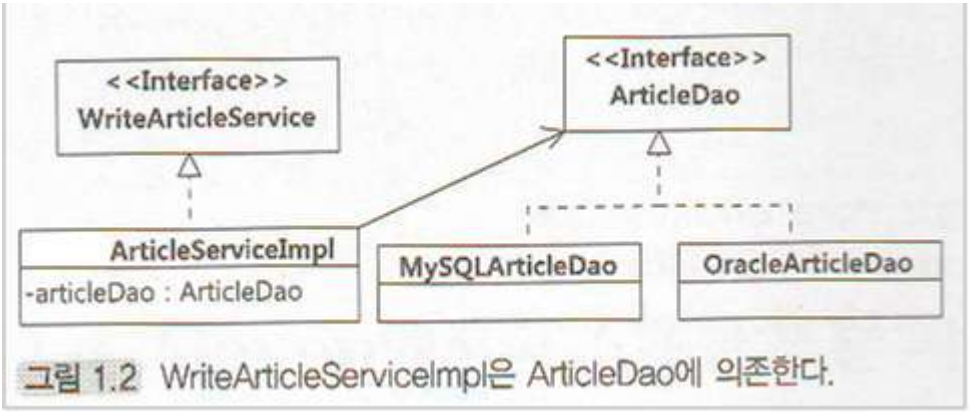


그림 1.2 WriteArticleServiceImpl은 ArticleDao에 의존한다.

ProductDAO 에 강하게 결합됨

```
public class ProductService {
        private ProductDAO dao;
        public ProductService(){
                dao = new ProductDAO();
        }//생성자

        public ProductBean detailProduct(int
pdNo){
                return dao.detailProduct(pdNo);
        }
```

종속 객체 주입(DI : Dependency Injection)

- ArticleServiceImpl 클래스는 ArticleDao 인터페이스에 의존하고 있음
- ArticleServiceImpl 클래스의 객체는 실제로 ArticleDao 인터페이스를 구현한 MySQLArticleDao 클래스의 객체나 OracleArticleDao 클래스의 객체에 의존하게 됨
- 예) ArticleDao 인터페이스의 구현 클래스로서 MySQLArticleDao클래스를 사용한다면, 메모리에 생성된 객체는 다음과 같은 의존 관계를 형성

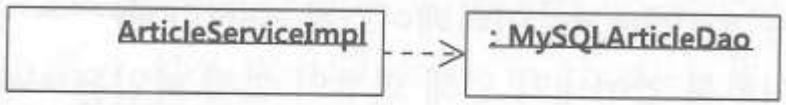


그림 1.3 실제 생성된 객체 간의 의존 관계

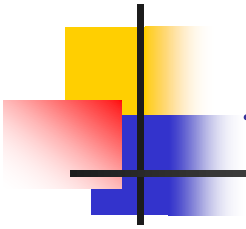
- ArticleServiceImpl 객체가 의존하는 객체는 ArticleDao 인터페이스를 구현한 클래스의 객체가 됨
  - ArticleServiceImpl 클래스는 실제로 의존할 객체를 지정할 수 있는 방법을 필요로 함
- [1] 실제로 의존하는 객체를 지정하는 간단한 방법 – 코드에 직접 명시

```
public class ArticleServiceImpl{
        private ArticleDao  articleDao = new MySQLArticleDao();
}
```

종속 객체 주입(DI : Dependency Injection)

- 코드에 직접 의존 클래스를 명시하는 방법의 단점
  - 단위 테스트를 어렵게 함
  - 의존하는 클래스가 변경되는 경우 코드를 변경해야 하는 문제
    - 예) 의존하는 클래스를 MySQLArticleDao에서 OracleArticleDao 로 변경해야 하는 경우 코드를 변경한 뒤 재컴파일 해야 함
- [2] 의존 관계를 처리하는 또 다른 방법
  - Factory 패턴이나 JNDI 등을 사용해서 의존 클래스를 검색하는 방법

```
public class ArticleServiceImpl{
        private ArticleDao articleDao = ArticleDaoFactory.create();
        …
}
```



종속 객체 주입(DI : Dependency Injection)

- ArticleDaoFactory 클래스는 ArticleDao 인터페이스를 구현한 클래스 중에서 어떤 클래스를 사용해야 할지의 여부를 알아내기 위해 외부의 설정 파일을 사용할 수도 있고, 시스템 프로퍼티를 사용할 수도 있을 것이다
  - Factory나 JNDI를 사용하면 의존 클래스가 변경되면 코드를 변경해야 하는 문제를 없앨 수는 있지만, ArticleServiceImpl 클래스를 테스트하려면 올바르게 동작하는 Factory 또는 JNDI 에 등록된 객체를 필요로 한다는 점에서, 코드에서 직접 의존하는 객체를 생성해 줄 때 발생하는 문제점을 완전히 극복하지 못함

종속 객체 주입(DI : Dependency Injection)

- [3] 외부의 조립기를 사용하는 방법
  - 의존 관계에 있는 객체가 아닌 외부의 조립기가 각 객체 사이의 의존 관계를 설정해 줌

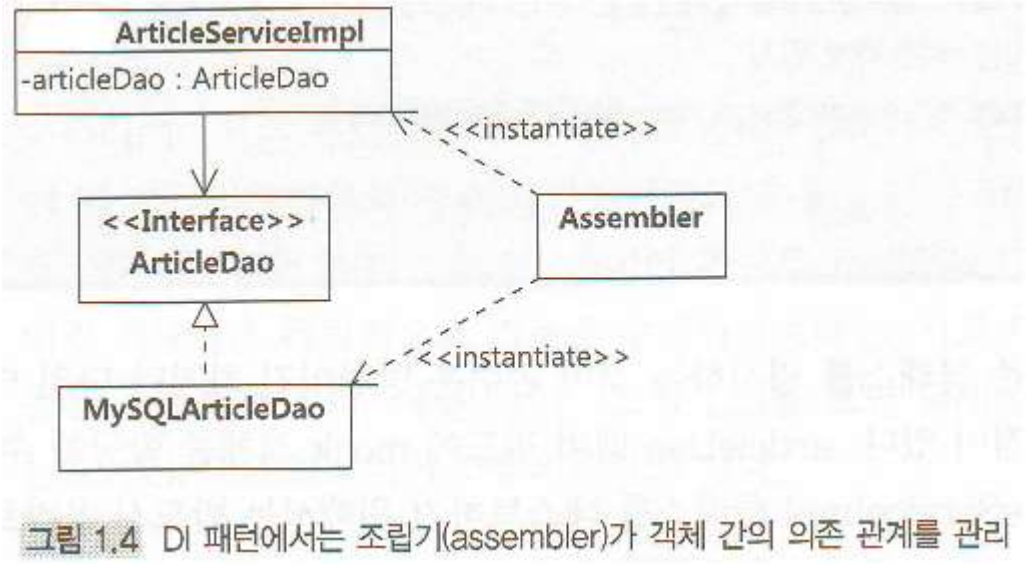
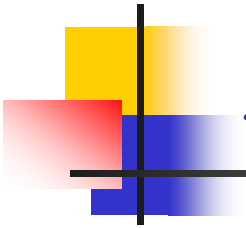


그림 1.4 DI 패턴에서는 조립기(assembler)가 객체 간의 의존 관계를 관리

  - ArticleServiceImpl 클래스의 코드는 MySQLArticleDao 객체를 생성하거나 검색하기 위한 코드가 포함되어 있지 않음
  - 대신, 조립기의 역할을 하는 Assembler 가 MySQLArticleDao 객체를 생성한 뒤 ArticleServiceImpl 객체에 전달해 줌



종속 객체 주입(DI : Dependency Injection)

- 조립기가 의존 관계를 관리해주는 방식 => DI 패턴이라고 함

- DI 패턴을 적용할 경우 ArticleServiceImpl 클래스는 의존하는 객체를 전달받기 위한 설정 메서드(setter method)나 생성자를 제공할 뿐 ArticleServiceImpl 에서 직접 의존하는 클래스를 찾지 않음

종속 객체 주입(DI : Dependency Injection)

- 예) ArticleServiceImpl 클래스는 다음과 같이 의존하는 객체를 전달받기 위한 생성자를 제공하게 됨

```
public class ArticleServiceImpl{
        private ArticleDao articleDao;

        //생성자에서 의존하는 객체를 전달받음
        public ArticleServiceImpl(ArticleDao articleDao){
                this.articleDao = articleDao;
        }
        …
}
```

```
public class ProductService {
                private ProductDAO pdDao;
                public ProductService(){
                                pdDao = new ProductDAO();
                }//생성자
```

```
public class ProductService {
                private ProductDAO pdDao;
                public ProductService(ProductDAO pdDao){
                                this.pdDao = pdDao;
                }//생성자
```

- ArticleServiceImpl 클래스는 의존 객체를 직접 생성하지도 않고, Factory나 JNDI를 이용하지도 않음
- 단지 **의존 객체를 전달 받을 수 있는** 메서드나 생성자만을 제공할 뿐.
- ArticleServiceImpl 클래스가 의존하고 있는 객체를 ArticleServiceImpl 객체에 전달해주는 역할은 조립기가 맡게 됨

종속 객체 주입(DI : Dependency Injection)

- 조립기는 내부적으로 다음과 같은 형태의 코드를 사용하여 ArticleServiceImpl 객체에 의존 객체를 전달하게 될 것임

```
public class Assembler{
        public ArticleService getArticleService() {
                ArticleDao articleDao = new MySQLArticleDao();
                ArticleService  service = ArticleServiceImpl(articleDao);
                return service;
        }
}
public class UsingService{
        public void useService(){
                …
                //조립기로부터 사용할 객체를 구함
                ArticleService  service = assembler.getArticleService();
                service.write();
        }
}
```

종속 객체 주입(DI : Dependency Injection)

- 조립기가 ArticleServiceImpl 에 MySQLArticleDao 객체를 넣어 주기 때문에, ArticleServiceImpl 클래스에는 의존 객체를 찾기 위한 코드가 필요하지 않게 됨
- 의존(Dependency)하는 객체를 조립기가 삽입(inject)해 주기 때문에 이 방식을 DI(Dependency Injection) 패턴이라고 부름
- 비슷한 의미로 Ioc(Inversion of Control)이라고도 표현함
- DI 패턴을 적용하면, 불필요한 의존 관계를 없애거나 줄일 수 있게 됨
  - 예) 의존하는 객체를 코드에서 직접 생성하는 경우 [그림1.5]과 같이 인터페이스와 구현 클래스에 대해서 의존 관계를 갖게 되지만, **DI 패턴을 적용하면 인터페이스에만 의존**하게 됨
  - 따라서 ArticleServiceImpl 을 수정할 필요없이 ArticleDao 구현 클래스를 MySQLArticleDao 에서 OracleArticleDao 로 교체할 수 있게 됨

종속 객체 주입(DI : Dependency Injection)

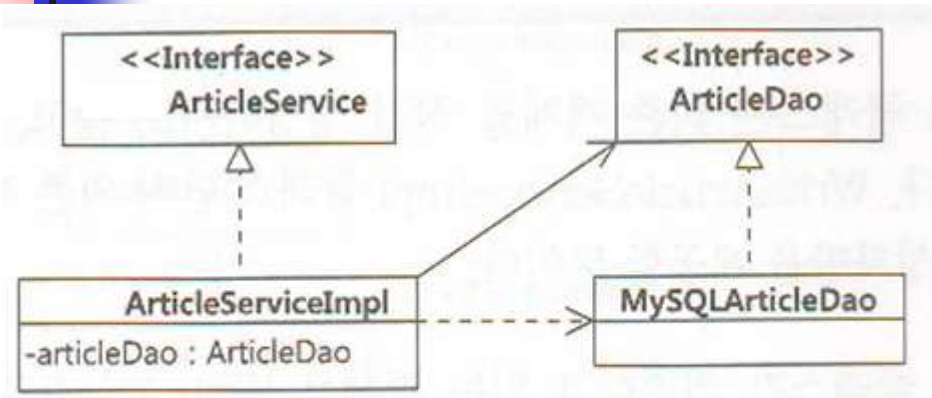
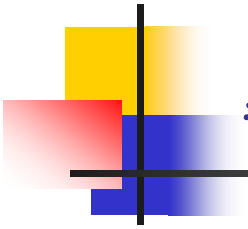


그림 1.5 의존 객체를 직접 생성할 경우 불필요한 의존 관계가 증가한다.

```
public class ArticleServiceImpl{
        private ArticleDao articleDao;

        public ArticleServiceImpl(){
                articleDao = new MySqlArticleDao();
        }
}
```

DI 패턴을 이용하면 ArticleServiceImpl 클래스는 오직 ArticleDao 인터페이스에만 의존하게 되며 그외 다른 클래스에는 의존하지 않게 됨

```
public class ArticleServiceImpl{
        private ArticleDao articleDao;

        //생성자에서 의존하는 객체를 전달받음
        public ArticleServiceImpl(ArticleDao articleDao){
                this.articleDao = articleDao;
        }
        …
}
```

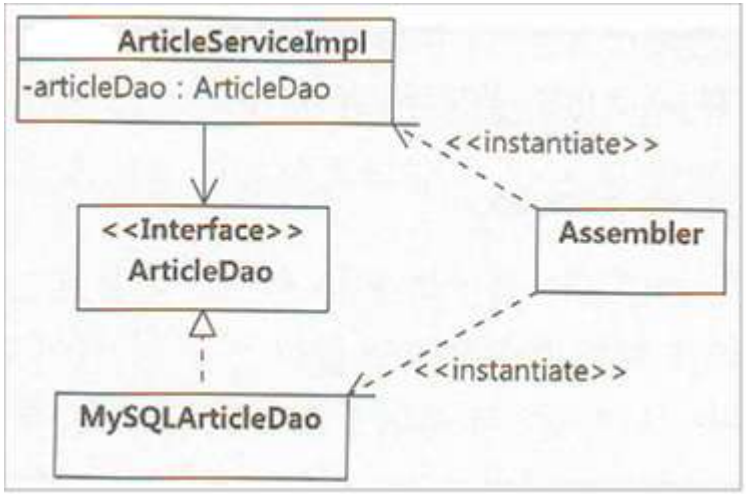


종속 객체 주입(DI : Dependency Injection)

- DI 패턴을 사용할 때의 또 다른 장점
  - 단위 테스트를 수행하는 게 수월해짐
- DI 패턴을 적용하려면 각 객체들을 조립해 주는 조립기가 필요함
  - 조립기를 직접 만들 수도 있지만, 주로 DI 패턴을 지원해주는 프레임워크를 사용하는 것이 좋다

스프링에서의 DI

- 스프링은 설정 파일과 어노테이션을 이용하여 손쉽게 객체 간의 의존 관계를 설정하는 기능을 제공하므로, 스프링을 객체 조립기로 사용할 수 있음




```
public class ArticleServiceImpl{
        private ArticleDao articleDao;

        //생성자에서 의존하는 객체를 전달받음
        public ArticleServiceImpl(ArticleDao articleDao){
                this.articleDao = articleDao;
        }
        …
}
```

- => 조립기가 ArticleServiceImpl 객체와 MySQLArticleDao 객체를 조립해줌

스프링에서의 DI

- [1] DI 패턴을 적용한 자바 코드
  - DI 패턴을 적용한 경우 ArticleServiceImpl 클래스는 생성자나 설정 메서드(setter)를 이용하여 의존 객체를 전달받을 수 있음
  - 예) 생성자 방식을 이용하여 의존 객체를 전달받는 ArticleServiceImpl 클래스

```
package mysite.spring.di;

public class ArticleServiceImpl{

    private ArticleDao articleDao;

    //생성자
    public ArticleServiceImpl(ArticleDao articleDao) {
        this.articleDao = articleDao;
    }

    @Override
    public void write(Article article) {
        System.out.println("ArticleServiceImpl.write() 메서드 실행");
        articleDao.insert(article);
    }
}
```

스프링에서의 DI

- 예) 설정 메서드 방식을 이용 – setter 메서드 이용

```
package mysite.spring.di;
public class ArticleServiceImpl{
    private ArticleDao articleDao;

    //setter
    public setArticleDao(ArticleDao articleDao) {
        this.articleDao = articleDao;
    }
.....
}
```

- ArticleServiceImpl 클래스가 전달받게 될 의존 객체 중 하나인 MySQLArticleDao 클래스는 스프링이 제공하는 DI 테스트를 위해 다음과 같이 작성

```
package mysite.spring.di;
public class MySQLArticleDao implements ArticleDao {
            @Override
            public void insert(Article article) {
                        System.out.println("MySql DB에 insert, MySQLArticleDao.insert() 실행");
            }
}
```

<bean> 엘리먼트는 스프링에서 가장 기본적인 설정단위로, 스프링에게 객체를 만들어 달라는 의미다

스프링에서의 DI

- [2] 스프링 설정 파일을 이용한 의존관계 설정
  - 스프링을 이용하여 ArticleServiceImpl 객체와 MySQLArticleDao 객체 사이의 의존 관계를 어떻게 처리할까?
  - 스프링은 코드나 설정 파일을 이용하여 객체 간의 의존관계를 설정할 수 있음
  - 아래 스프링 설정 파일을 클래스패스에 위치시킨다

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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-3.0.xsd">
        <bean name="articleService"
                class="mysite.spring.di.ArticleServiceImpl">
                <constructor-arg>
                        <ref bean="articleDao" />
                </constructor-arg>
        </bean>

        <bean name="articleDao" class="mysite.spring.di.MySQLArticleDao">
        </bean>
</beans>
```

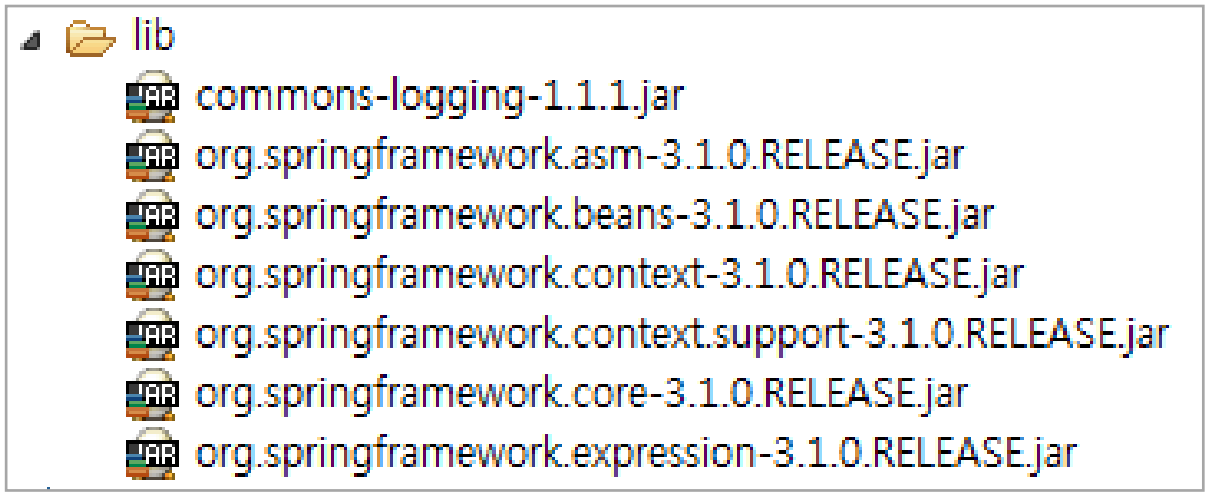
스프링에서의 DI

- 스프링은 각 객체를 bean 으로 관리함
  - <beans> 태그 – 스프링 설정 파일의 루트 태그
  - <bean> 태그 – 스프링이 관리할 하나의 객체를 설정하는 데 사용됨
    - <bean> 태그의 name 속성 – 빈의 이름을 의미,
    - class 속성 – 생성될 객체의 클래스 타입
  - <constructor-arg> 태그 – articleService 빈 객체를 생성할 때 생성자에 전달할 파라미터를 명시하기 위해 사용됨
    - 생성자에 articleDao 빈 객체를 전달한다고 명시
    - 다음과 같은 의미

```
MySQLArticleDao  articleDao = new MySQLArticleDao();
ArticleServiceImpl  articleService = new ArticleServiceImpl(articleDao);
```

스프링에서의 DI

- [3] 클래스패스 설정 및 ApplicationContext 를 이용한 빈 객체 사용
  - 스프링을 이용하여 설정 파일을 로딩한 뒤, 설정 파일에 명시한 빈 객체를 사용하는 작업이 남았다
  - 스프링 컨테이너를 사용하려면 아래의 jar 파일을 클래스패스에 추가해 주어야 함



```
ApplicationContext context =
    new ClassPathXmlApplicationContext("applicationContext.xml "); //스프링 컨텍스트 로드
```

스프링에서의 DI

ClassPathXmlApplicationContext – 애플리케이션의 클래스 패스에 있는 하나 이상의 XML 파일에서 스프링 컨텍스트를 로드함

- 작성한 설정 파일로부터 BeanFactory 를 생성하고, BeanFactory 로부터 필요한 빈 객체를 가져와서 사용하는 예제

```
package mysite.spring.di;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

public class Main {

        public static void main(String[] args) {
                Resource resource = new ClassPathResource("applicationContext.xml");
                BeanFactory beanFactory = new XmlBeanFactory(resource);
                ArticleServiceImpl articleService = (ArticleServiceImpl) beanFactory
                                .getBean("articleService");
                articleService.write(new Article());
        }
}
```

//스프링 컨텍스트 로드

// ArticleService 빈 얻기

// ArticleService 사용

스프링에서의 DI

또는

```
//스프링 컨텍스트 로드
ApplicationContext context =
        new ClassPathXmlApplicationContext("applicationContext.xml ");

// ArticleService 빈 얻기
 ArticleServiceImpl articleService
     = (ArticleServiceImpl)context.getBean(“articleService");

// ArticleService 사용
 articleService.write(new Article());
```

ClassPathXmlApplicationContext 를 사용해 applicationContext.xml 을 로드하고 ArticleService 에 대한 레퍼런스를 얻는다

스프링에서의 DI

- XmlBeanFactory 클래스는 resource가 나타내는 xml 파일로부터 스프링 설정 내용을 로딩하여 빈 객체를 생성하는 BeanFactory 구현 클래스임
  - BeanFactory – 빈 객체를 관리하는 컨테이너
- BeanFactory 객체를 생성하면, BeanFactory로부터 빈 객체를 가져와 사용할 수 있게 됨
  - 설정파일에서 <bean>태그의 name 속성을 이용하여 빈 객체에 이름을 부여했는데, 이때 부여한 이름을 사용하여 빈 객체를 구할 수 있음
- 스프링 컨테이너로부터 빈 객체를 가져오려면 BeanFactory.getBean(String name) 메서드 사용
- 실행 결과

```
ArticleServiceImpl.write() 메서드 실행
MySQLArticleDao.insert() 실행
```

전체 코드
Main.java

```
public static void main(String[] args) {
        /*ArticleServiceImpl1service = new ArticleServiceImpl1();

        Article article = new Article();
        article.setNo(1);
        article.setTitle("안녕");

        service.write(article);                    */

        //applicationContext.xml 스프링 설정파일에서 해당 빈 객체를 얻어온다
        //1. 스프링 컨텍스트 로드
        ApplicationContext context
                    =new ClassPathXmlApplicationContext("applicationContext.xml");

        //Resource resource = new ClassPathResource("applicationContext.xml");
        //BeanFactory context = new XmlBeanFactory(resource);

        //2. 빈(ArticleServiceImpl2) 얻어오기
        ArticleServiceImpl2 articleService
                    = (ArticleServiceImpl2)context.getBean("articleServiceImpl2");

        //3. 빈 사용하기
        Article article = new Article();
        article.setNo(2);
        article.setTitle("생성자에서 종속객체 주입 받는 경우");
        articleService.write(article);
```

```
    //3. setter에서 종속객체를 주입한 경우
            ApplicationContext context
            = new ClassPathXmlApplicationContext("applicationContext.xml");

            ArticleServiceImpl3  articleService3
                        =  (ArticleServiceImpl3) context.getBean("articleServiceImpl3");
            Article bean = new Article();
            bean.setNo(3);
            bean.setTitle("setter 에서 종속객체 주입");

            articleService3.write(bean);
}
```

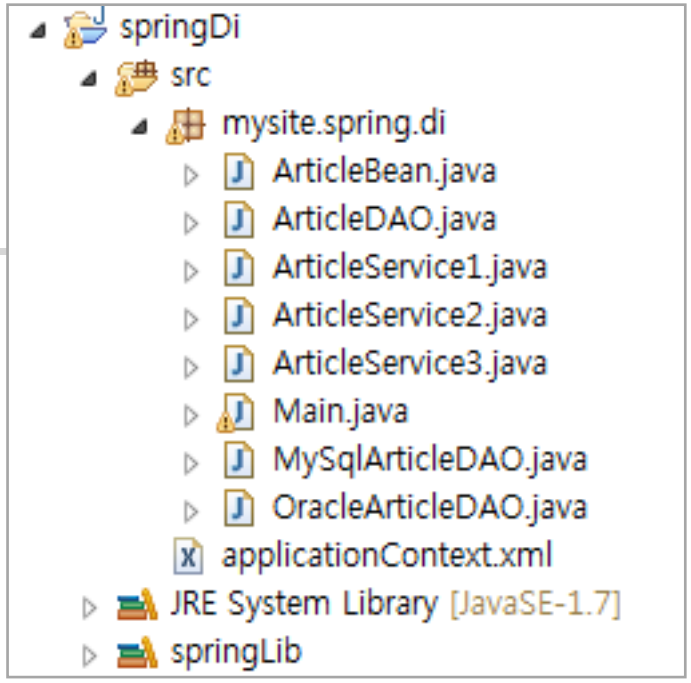
전체 코드
Main.java

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean name="articleServiceImpl2"
          class="mysite.spring.di.ArticleServiceImpl2">
          <constructor-arg>
                <ref bean="mySqlArticleDAO"/>
          </constructor-arg>
    </bean>

    <!-- <bean name="articleServiceImpl3"
          class="mysite.spring.di.ArticleServiceImpl3">
          <property name="articleDao" ref ="mySqlArticleDAO"/>
    </bean> -->
    <bean name="articleServiceImpl3"
          class="mysite.spring.di.ArticleServiceImpl3"
          p:articleDao-ref ="mySqlArticleDAO"/>

    <bean name="oracleArticleDAO" class="mysite.spring.di.OracleArticleDAO"></bean>
    <bean name="mySqlArticleDAO" class="mysite.spring.di.MySqlArticleDAO"></bean>
```

```
<bean name="articleServiceImpl2"
      class="mysite.spring.di.ArticleServiceImpl2">
        <constructor-arg  ref ="mySqlArticleDAO" />
</bean>
```

```
package mysite.spring.di;

public class Article {
		public int no;
		public String title;

		public int getNo() {
			return no;
		}
		public void setNo(int no) {
			this.no = no;
		}
....
}
```



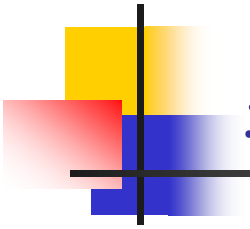

```
package mysite.spring.di;

public interface ArticleDao {

		void insert(Article article);
}
```

```
ArticleServiceImpl3.write() 메서드 호출
db에 접속해서 테이블에 insert한다 : oracleDao.insert()
---------메인 끝----------
```

```
ArticleServiceImpl3.write() 메서드 호출
MySql db에서 테이블에 insert한다 : mySqlDao.insert()
---------메인 끝----------
```

```
package mysite.spring.di;
public class MySqlArticleDAO implements ArticleDAO{
	@Override
	public void insert(Article article) {
		System.out.println("MySql db에서 테이블에 insert한다 : mySqlDao.insert()");
	}
}
public class OracleArticleDAO implements ArticleDAO{
	@Override
	public void insert(Article article) {
		System.out.println("oracle db에 접속해서 테이블에 insert한다 : oracleDao.insert()");
	}
}
public class ArticleServiceImpl{
	private ArticleDAO articleDao;
	//생성자
	public ArticleServiceImpl(){
		//dao 객체가 필요하므로 직접 객체 생성해서 사용함
		articleDao = new OracleArticleDAO();
	}
	@Override
	public void write(Article article) {
		System.out.println("articleServiceImpl.write() 메서드 호출");
		articleDao.insert(article);
	}
}
```

```
public class ArticleServiceImpl2{
	private ArticleDAO articleDao;

	//DI를 이용하여 dao객체를 직접 생성하지 않고, 외부 조립기가 주입하도록 한다
	//생성자 방식 - 생성자를 이용한 종속객체 주입
	public ArticleServiceImpl2(ArticleDAO articleDao){
		//생성자의 매개변수에 필요한 객체를 지정함
		this.articleDao = articleDao;
	}
	@Override
	public void write(Article article) {
		System.out.println("articleServiceImpl2.write() 메서드 호출");
		articleDao.insert(article);
	}
}
public class ArticleServiceImpl3 {
	//멤버변수-프로퍼티
	private ArticleDAO articleDao;

	//DI를 이용하여 dao객체를 직접 생성하지 않고, 외부 조립기가 주입하도록 한다
	//프로퍼티 설정 방식 - setter를 이용한 종속객체 주입
	public void setArticleDao(ArticleDAO articleDao){
		//setter의 매개변수에 필요한 객체를 지정함
		this.articleDao = articleDao;
	}
	@Override
	public void write(Article article) {
		System.out.println("articleServiceImpl3.write() 메서드 호출");
		articleDao.insert(article);
	}
}
```



스프링에서의 DI

- main() 메서드는 applicationContext.xml 파일을 기반으로 스프링 애플리케이션 컨텍스트를 생성함
- ID 가 articleService 인 빈을 조회하기 위해 팩토리로 애플리케이션 컨텍스트를 사용
- ArticleService 객체에 대한 레퍼런스를 얻은 후에 간단히 write() 메소드를 호출해 작업을 처리함
- 이 클래스는 ArticleDao가 어떤 유형의 dao인지에 대해서는 아무것도 알지 못함
- MySQLArticleDao를 다룬다는 사실도 알지 못함
- applicationContext.xml 파일만이 어떤 구현체인지 알고 있음

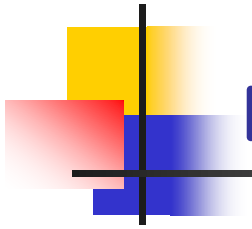
model 1 방식의 코드

ArticleServiceImpl2.java

```
package mysite.spring.di;
public class ArticleServiceImpl2{

    private ArticleDao articleDao;

    public ArticleServiceImpl2() {
        articleDao = new MySQLArticleDao2();
    }

    @Override
    public void write(Article article) {
        System.out.println("ArticleServiceImpl2.write() 메서드 실행");
        articleDao.insert(article);
    }
}
```

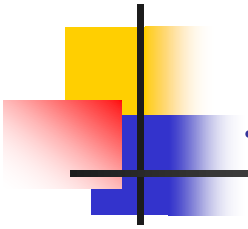
MySQLArticleDao2.java

```
package mysite.spring.di;
public class MySQLArticleDao2 implements ArticleDao {
            @Override
            public void insert(Article article) {
                        System.out.println("MySQLArticleDao2.insert() 실행");
            }
}
```



model 1 방식의 코드

```
package mysite.spring.di;

public class Main {
		public static void main(String[] args) {
			ArticleServiceImpl2  articleService = new ArticleServiceImpl2();
			Article article  = new Article()
			articleService.write(article);
		}
}
```

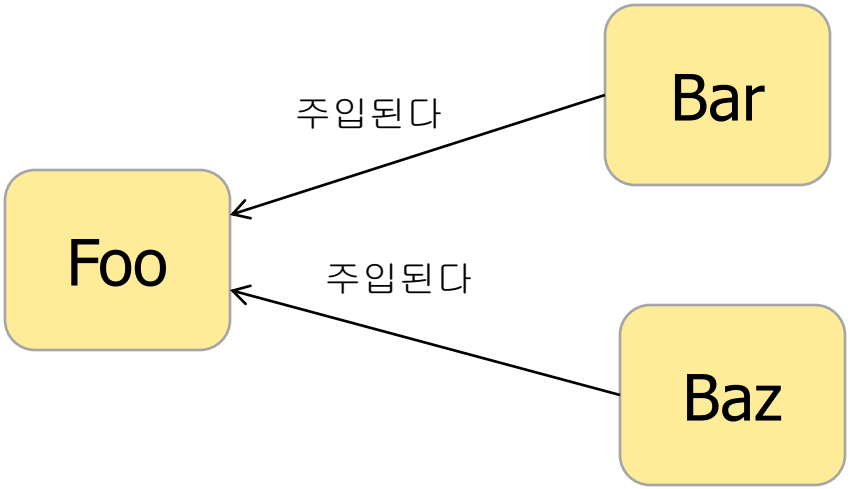


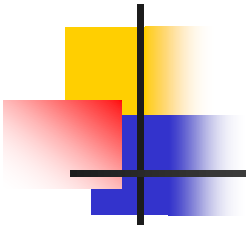
스프링에서의 DI

- 요점 – ArticleServiceImpl 가 ArticleDao 의 특정 구현체에 결합되지 않는다
  - ArticleDao 인터페이스를 구현하기만 하면 ArticleServiceImpl 에게 어떤 종류의 dao을 요청하든 문제가 되지 않음
  - => DI의 주요 이점인 **'느슨한 결합도'**
  - 어떤 객체가 자신이 필요로 하는 종속객체를 인터페이스를 통해서만 알고 있다면 사용하는 객체 쪽에는 아무런 변경 없이 종속 객체를 다른 구현체로 바꿀 수 있다.

종속객체 주입

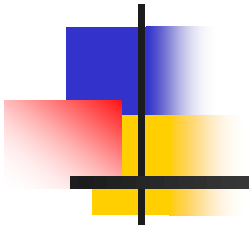
- 와이어링(wiring)
  - 애플리케이션 **컴포넌트 간의 관계를 정하는 것**
  - 스프링에서 컴포넌트를 와이어링하는 일반적인 방법은 XML을 이용하는 방법
- 종속객체 주입
  - 객체가 스스로 종속객체를 획득하는 것과는 반대로 객체에 종속객체가 부여되는 것을 의미함





애플리케이션 컨텍스트

- applicationContext.xml 스프링 설정 파일을 로드하여 애플리케이션 구동하기
- 애플리케이션 컨텍스트(Application Context)
  - 스프링 애플리케이션에서 애플리케이션 컨텍스트는 빈에 관한 정의들을 바탕으로 **빈들을 엮어 줌**
  - 애플리케이션 컨텍스트는 애플리케이션을 구성하는 **객체의 생성과 와이어링을 책임짐**
  - applicationContext.xml 에서는 빈들이 XML 파일에 선언되어 있으므로 애플리케이션 컨텍스트로 ClassPathXmlApplicationContext를 사용
  - ClassPathXmlApplicationContext – 애플리케이션의 클래스 패스에 있는 하나 이상의 XML 파일에서 스프링 컨텍스트를 로드함



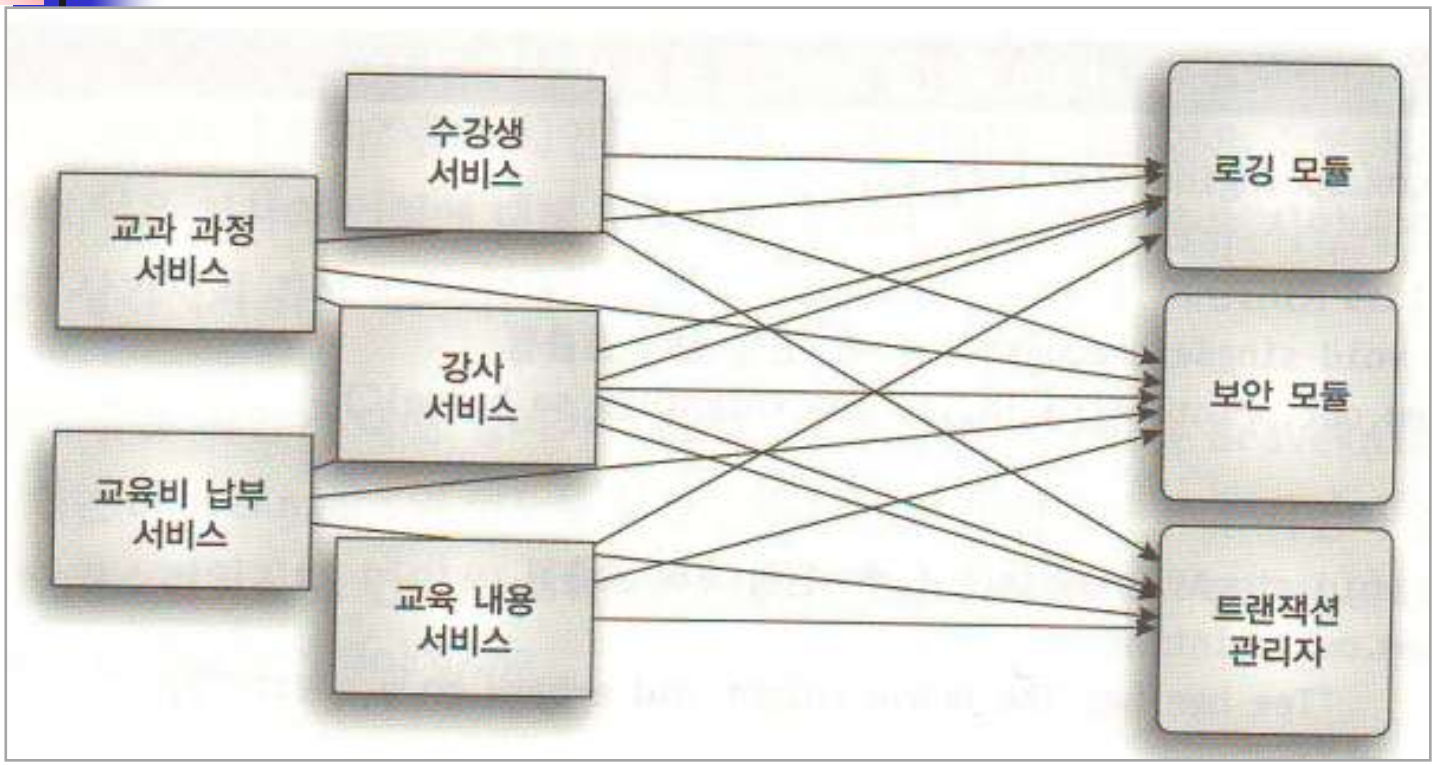
AOP

애스펙트 적용(AOP)

AOP - 기존 비즈니스 로직에 영향을 주지 않고 필요한 추가 처리를 곳곳에 넣을 수 있는 개발 기법
주로 인증이나 로깅 등을 적용할 때 많이 사용

- DI – 소프트웨어 컴포넌트의 결합도를 낮춰줌
- 애스펙트 지향 프로그래밍 (AOP: Aspect-Oriented Programming) – 애플리케이션 전체에 걸쳐 사용되는 기능을 재사용할 수 있는 컴포넌트에 담을 수 있게 해줌
  - 소프트웨어 시스템 내부의 관심사들을 서로 분리하는 기술
  - 시스템은 보통 특정한 기능을 책임지는 여러 개의 컴포넌트로 구성됨
  - 각 컴포넌트는 본연의 특정한 기능 외에 로깅이나 트랜잭션 관리, 보안 등의 시스템 서비스도 수행해야 하는 경우가 많다
  - 이러한 시스템 서비스는 시스템의 여러 컴포넌트에 관련되는 경향이 있기 때문에 횡단 관심사 라고 함
  - 이러한 관심사가 여러 컴포넌트에 퍼지게 되면,
    - 시스템 전반에 걸친 관심사를 구현하는 코드가 여러 컴포넌트에서 중복되어 나타나고
    - 컴포넌트의 코드가 본연의 기능과 관련 없는 코드로 인해 지저분해짐

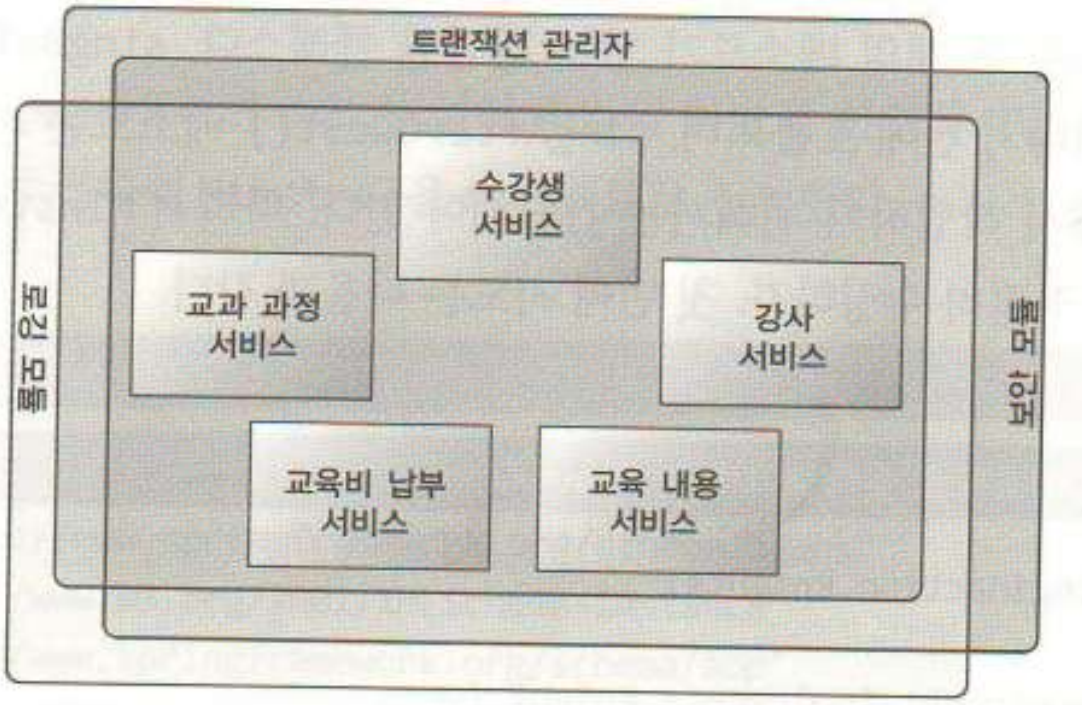
애스펙트 적용하지 않은 경우



시스템 관련 관심사가 주 관심사가 아닌 모듈들에 로깅이나 보안 등 시스템 전반에 걸친 관심사에 대한 호출들이 여기저기 흩어져 있다

각 객체가 본연의 책임을 수행할 뿐만 아니라 로깅과 보안, 트랜잭션 컨텍스트에 대해서도 알아야 함

AOP 적용

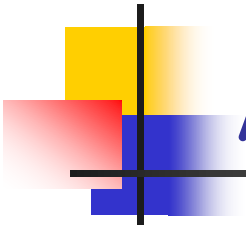


AOP 를 이용하면 시스템 전반에 걸친 관심사가 관련 컴포넌트를 감싼다
이렇게 함으로써 애플리케이션의 컴포넌트들이 본연의 비즈니스 기능에 집중할 수 있다.

AOP

AOP 에서는 핵심 로직을 구현한 클래스를 실행하기 전, 후에 **Aspect** 를 적용하고,
그 결과로 핵심 로직을 수행하면 그에 앞서 공통 모듈을 실행하거나 또는 로직 수행 이후에 공통 모듈을 수행하는 방식으로 공통 기능을 실행하게 됨

- AOP
  - 애플리케이션 코드에 산재해서 나타나는 **부가적인 기능을 독립적으로 모듈화**하는 프로그래밍 모델
    - 스프링은 AOP를 이용해서, 다양한 엔터프라이즈 서비스를 적용하고도 깔끔한 코드를 유지할 수 있게 해줌
  - 시스템 서비스를 모듈화해서 컴포넌트에 **선언적**으로 적용할 수 있게 해줌
  - AOP를 이용하면 시스템 서비스에 대해서는 전혀 알지 못하면서 응집도가 높고 **본연의 관심사에 집중하는 컴포넌트를 만들 수 있다**
    - 애스펙트 – 애플리케이션의 여러 컴포넌트를 덮는 담요처럼 생각하자
  - 애플리케이션의 핵심은 비즈니스 기능을 구현하는 모듈들로 구성되어 있고, AOP를 이용해서 이 핵심 애플리케이션 위에 추가적인 기능을 여러 겹으로 덮고 있다
  - **핵심 기능을 구현하는 모듈에는 아무런 변화도 가하지 않고 추가적인 기능을 선언적으로 적용할 수 있음**



AOP와 스프링

- 여러 부분에 걸쳐서 공통으로 사용되는 기능
  - 로깅이나, 트랜잭션 처리, 보안과 같은 기능은 대부분의 어플리케이션에서 필요로 함
- 트랜잭션이나 보안과 같은 기능은 어떤 특정 모듈에서 필요로 하기 보다는, 어플리케이션 전반에 걸쳐서 필요한 기능이며, 실제 핵심 비즈니스 로직과는 구분되는 기능임
  - 예) 게시글 쓰기의 경우 현재 사용자가 권한이 있는 지 검사하는 보안 처리 로직과 데이터 무결성 처리를 위한 트랜잭션 처리 로직은 게시글 쓰기를 수행하기 위해 필요한 로직과는 구분되는 기능임
  - 보안 처리와 트랜잭션 처리는 글쓰기뿐 아니라 글삭제, 글목록 등 다양한 기능을 구현하는 데 필요한 공통 관심 사항(cross-cutting concern)임
  - 이런 공통 관심 사항들을 객체 지향 기법(상속이나 패턴)을 사용해서 여러 모듈에 효과적으로 적용하는 데는 한계가 있으며, 이런 한계를 극복하기 위해 AOP 라는 기법이 소개됨

AOP 소개

- 보안이나 트랜잭션과 같은 공통 기능을 별도의 모듈로 구현한 뒤, 각 기능을 필요로 하는 곳에서 사용하게 될 경우, 각 모듈과 공통 모듈 사이의 의존관계는 [그림]과 같이 형성됨

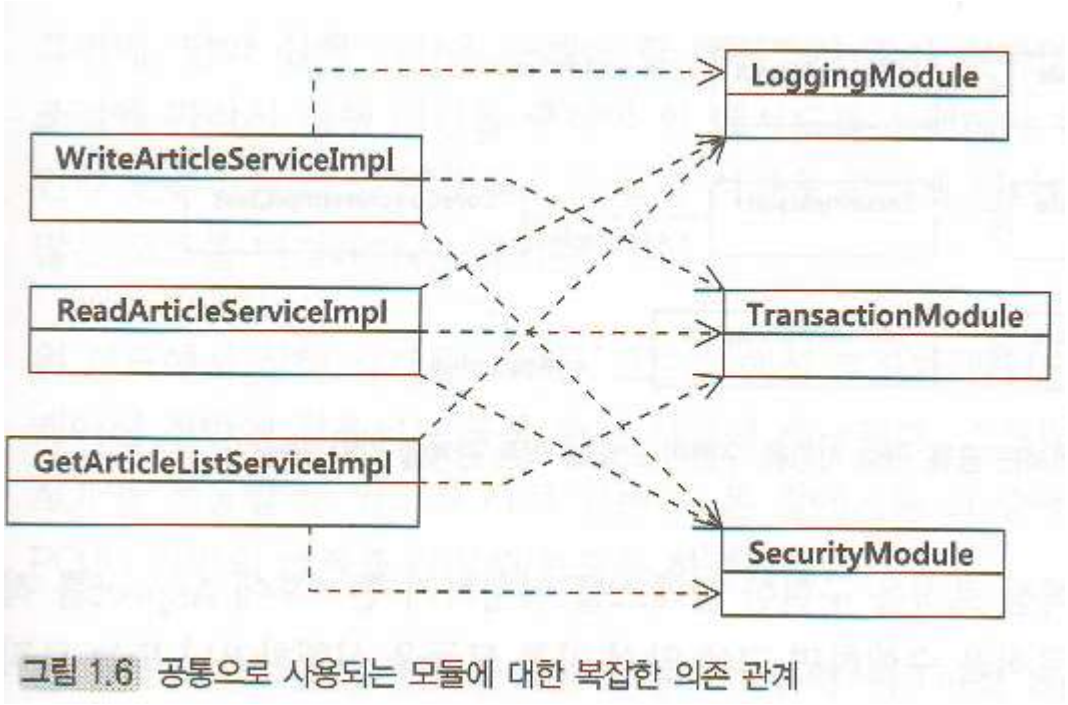


그림 1.6 공통으로 사용되는 모듈에 대한 복잡한 의존 관계

•어플리케이션 전반에 걸쳐서 공통으로 사용되는 기능이 많아 질수록, 그리고 공통 모듈을 사용하는 클래스가 많아 질수록 의존 관계는 점점 복잡해짐
 - 공통 모듈을 사용하는 코드가 여러 곳에서 중복되는 문제도 발생

AOP 소개

- AOP(Aspect Oriented Programming) – 공통의 관심 사항을 적용해서 발생하는 의존 관계의 복잡성과 코드 중복을 해소해 주는 프로그래밍 기법
  - AOP 에서는 각 클래스에서 공통 관심 사항을 구현한 모듈에 대한 의존 관계를 갖기 보다는, Aspect 를 이용하여 핵심 로직을 구현한 각 클래스에 공통 기능을 적용하게 됨

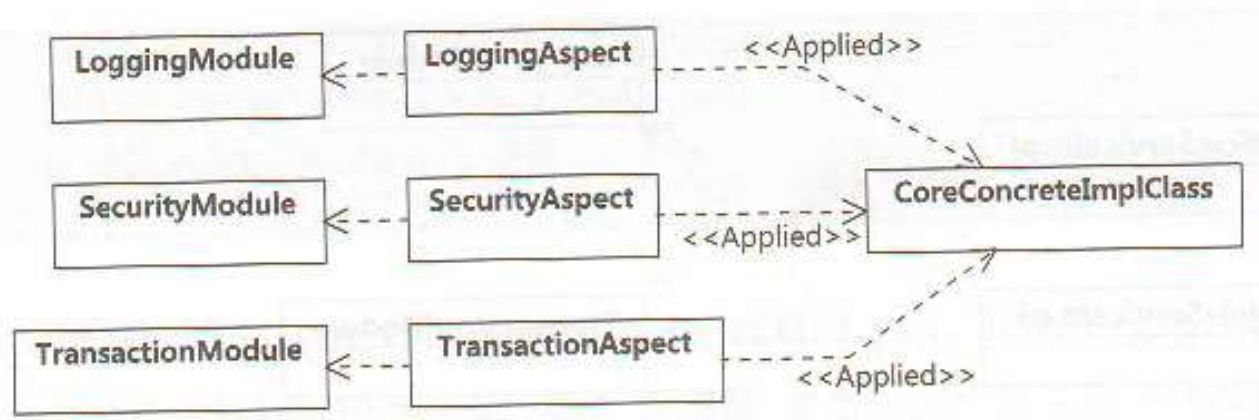


그림 1.7 AOP에서는 공통 관심 사항을 구현한 모듈에 의존 관계를 갖지 않는다.

AOP 소개

- [그림]은 Aspect 를 이용할 경우 로직을 구현한 클래스가 어떻게 변경되는지를 보여줌
- 핵심 로직을 구현한 클래스는 더 이상 여러 공통 모듈에 의존하지 않음
- 즉, 핵심 로직 구현 클래스에서 공통 모듈을 사용하는 코드를 포함하지 않음
- AOP 에서는 핵심 로직을 구현한 클래스를 실행하기 전, 후에 Aspect 를 적용하고, 그 결과로 핵심 로직을 수행하면 그에 앞서 공통 모듈을 실행하거나 또는 로직 수행 이후에 공통 모듈을 수행하는 방식으로 공통 기능을 실행하게 됨

~~~한 aspect가 실행 될 때엘때
~~~

AOP 소개

- AOP – 문제를 바라보는 관점을 기준으로 프로그래밍하는 기법
  - 문제를 해결하기 위한 핵심 관심사항과 전체에 적용되는 공통 관심사항을 기준으로 프로그래밍함으로써 공통 모듈을 여러 코드에 쉽게 적용할 수 있도록 도와줌
  - AOP의 기본적인 개념 – 공통 관심 사항을 구현한 코드를 핵심 로직을 구현한 코드 안에 삽입한다는 것

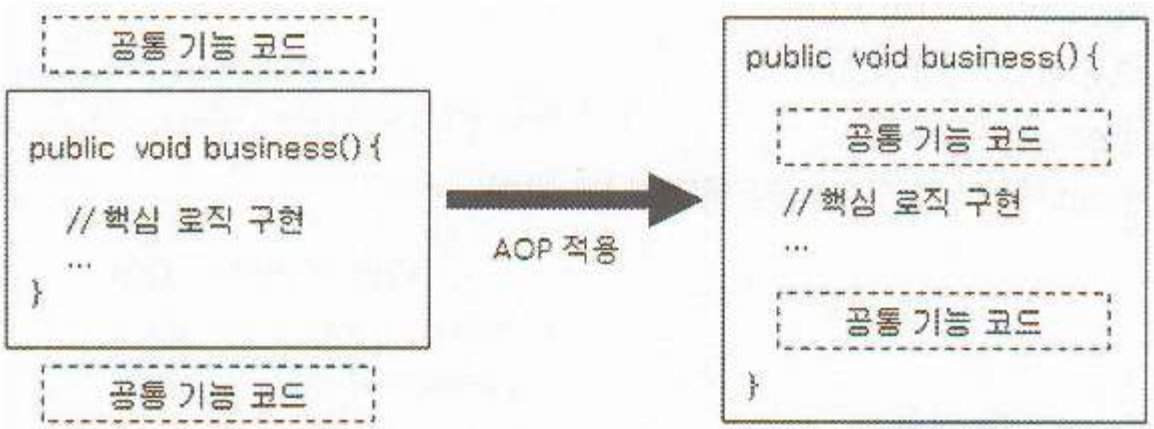
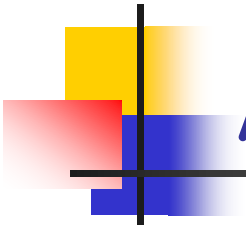


그림 5.1 AOP에서 공통 기능을 구현한 코드가 핵심 로직에 삽입되어 실행된다.

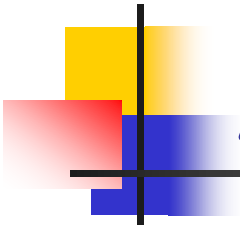
AOP 소개

- AOP 기법에서는 [그림]과 같이 핵심 로직을 구현한 코드에서 공통 기능을 직접적으로 호출하지 않음
- 핵심 로직을 구현한 코드를 컴파일하거나, 컴파일 된 클래스를 로딩하거나, 또는 로딩한 클래스의 객체를 생성할 때 AOP가 적용되어 핵심 로직 구현 코드 안에 공통 기능이 삽입됨
- AOP 프로그래밍에서는 AOP 라이브러리가 공통 기능을 알맞게 삽입해 주기 때문에, 개발자는 게시글 쓰기나 목록 읽기와 같은 핵심 로직을 구현할 때 트랜잭션 적용이나 보안 검사와 같은 공통 기능을 처리하기 위한 코드를 핵심 로직 코드에 삽입할 필요가 없음



AOP 소개

- 핵심 로직을 구현한 코드에 공통 기능 관련 코드가 포함되어 있지 않기 때문에 적용해야 할 공통 기능이 변경되더라도 핵심 로직을 구현한 코드를 변경할 필요가 없음
- 단지, 공통 기능 코드를 변경한 뒤 핵심 로직 구현 코드에 적용하기만 하면 됨



AOP 소개

- AOP에서 중요한 점
  - Aspect 가 핵심 로직 구현 클래스에 의존하지 않는다는 점
  - AOP에서는 **설정 파일이나 설정 클래스** 등을 이용하여 Aspect 를 여러 클래스에 적용할 수 있도록 하고 있음
  - => 하나의 Aspect 를 개발하게 되면, Aspect 를 수정할 필요 없이 여러 클래스에 적용할 수 있음

스프링에서의 AOP

- 예) 메서드의 실행 시간을 출력해 주는 코드
- [1] AOP를 적용하지 않는다면.

```
public void someMethod(){
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();

        executeLogic();

        stopWatch.stop();
        long executionTime = stopWatch.getTotalTimeMillis();
}
```

실행 시간을 구해야 할 메서드가 많아 진다면?
또는 조건에 따라서 실행시간을 구해야 할 메서드를 선택해야 한다면?
⇒ 요구가 변경될 때마다 많은 코드를 변경해야 함
실행 시간을 구하는 코드는 핵심 로직과 관련된 것이라기 보다는 어플리케이션 전반에 적용되는 공통 관심사항에 해당됨

스프링에서의 AOP

- [2] AOP를 적용
- 아래 jar 파일을 클래스패스에 추가해 주어야 함

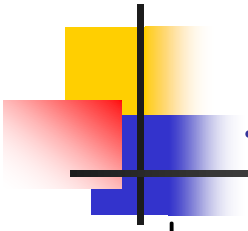
org.springframework.asm-3.1.0.RELEASE.jar
org.springframework.core-3.1.0.RELEASE.jar
org.springframework.beans-3. 1.0.RELEASE.jar
org.springframework.aop-3.1.0.RELEASE.jar
org.springframework.context-3.1.0.RELEASE.jar
org.springframework.expression-3.1.0.RELEASE.jar
aopalliance-1.0.jar
aspectjweaver-1.6.5.jar
commons-logging-1.1.1.jar

공통 관심 사항을 구현한 POJO 클래스 작성하기
스프링 프레임워크가 제공하는 StopWatch 클래스를 이용하여 메서드 실행시간을 구할 것이며, 공통 기능을 구현한 클래스는 다음과 같다

스프링에서의 AOP

- JoinPoint : 공통 관심 사항이 적용될 수 있는 지점 (ex)메서드 호출 시, 객체 생성 시 등)
- ProceedingJoinPoint : JoinPoint 의 하위클래스로서 proceed() 메서드를 가지고 있다.
- proceed() 메서드를 사용하여 실제 타켓 메서드를 호출하게 된다

```
package mysite.spring.di;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;
public class LoggingAspect {
        private Log log = LogFactory.getLog(getClass());

        public Object logging(ProceedingJoinPoint joinPoint) throws Throwable {
                log.info("기록 시작");
                StopWatch stopWatch = new StopWatch();
                try {
                        stopWatch.start();
                        Object retValue = joinPoint.proceed();
                        return retValue;
                } catch (Throwable e) {
                        throw e;
                } finally {
                        stopWatch.stop();
                        log.info("기록 종료");
                        log.info(joinPoint.getSignature().getName() + "메서드 실행 시간 : "
                                        + stopWatch.getTotalTimeMillis());
                }
        }
}
```



스프링에서의 AOP

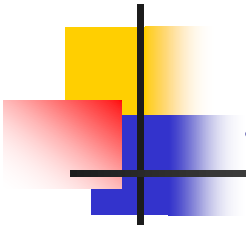
- logging() 메서드는 Aspect가 적용되는 메서드의 실행시간을 구한 뒤 Log를 통해 출력하도록 구현됨
- logging() 메서드가 파라미터로 전달받는 ProceedingJoinPoint 객체는 Aspect 가 적용되는 객체 및 메서드에 대한 정보를 담고 있으며, 이 객체를 통해서 핵심 로직을(Aspect 가 적용되는 메서드)실행할 수 있게 됨
- LoggingAspect는 핵심 기능을 구현한 클래스나 인터페이스에 전혀 의존하지 않고, 오직 ProceedingJoinPoint 에만 의존함
  - Aspect를 적용할 객체에 의존하지 않도록 함으로써 코드 수정 없이 LoggingAspect를 여러 클래스에 적용할 수 있게 됨

스프링에서의 AOP

- 공통 기능을 구현한 Aspect 클래스를 작성했으면, Aspect를 어떤 클래스의 어떤 메서드에 적용할지를 설정해야 함
- 스프링의 xml 설정 파일을 다음과 같이 작성함으로써 LoggingAspect가 ArticleServiceImpl 클래스에 적용되도록 설정하자

```
<?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:aop="http://www.springframework.org/schema/aop"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
   <bean id="loggingAsp" class="mysite.spring.di.LoggingAspect" />

   <aop:config>
      <aop:pointcut id="servicePointcut"
         expression="execution(* *..*Service.*(..))" />

      <aop:aspect id="loggingAspect" ref="loggingAsp">
         <aop:around pointcut-ref="servicePointcut" method="logging" />
      </aop:aspect>
   </aop:config>
</beans>
```

loggingAspect의 logging()메서드를 이름이 Service로 끝나는 인터페이스를 구현한 모든 클래스의 모든 메서드 앞, 뒤로 실행하라



스프링에서의 AOP

- xml 스키마 확장(aop 네임스페이스)를 사용하여 AOP를 설정하였음
- 이름이 Service로 끝나는 인터페이스를 구현한 모든 클래스의 모든 메서드에 LoggingAspect 가 적용됨
- 공통 관심사항을 구현한 클래스를 작성했고, 공통 관심사항을 어떻게 적용할지에 대한 내용을 설정 파일로 작성했으면, 실제로 Aspect 가 적용되는지 테스트

스프링에서의 AOP

```
package com.spring.di;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class MainForAop {
        public static void main(String[] args) {
                String configLocations = "applicationContext.xml";
                ApplicationContext context = new ClassPathXmlApplicationContext(
                                configLocations);
                ArtService articleService = (ArtService) context.getBean("articleService3");
                articleService.insert(new ArticleBean());
        }
}
```

- BeanFactory 대신 ApplicationContext 를 사용
- 스프링 설정 파일로 commonConcern.xml 이 추가됨

```
package com.spring.di;

public interface ArtService {
	public void insert(ArticleBean articleBean) ;
}



public class ArticleService3 implements ArtService{
	//3번째 방법 - DI 이용
	//setter에서 종속객체 주입받는 경우
	private ArticleDAO articleDao;

	//setter - 종속객체 주입
	public void setArticleDao(ArticleDAO articleDao){
		this.articleDao = articleDao;
	}

	//메서드
	public void insert(ArticleBean articleBean) {
		articleDao.insert(articleBean);
		System.out.println("ArticleService3의 insert()메서드!!"
				+ " setter에서 종속객체를 주입받는 경우");
	}
}
```

스프링에서의 AOP

- 실행 결과

```
2013. 7. 22 오후 11:08:17 madvirus.spring.chap01.LoggingAspect logging
정보: 기록 시작
WriteArticleServiceImpl.write() 메서드 실행
MySQLArticleDao.insert() 실행
2013. 7. 22 오후 11:08:17 madvirus.spring.chap01.LoggingAspect logging
정보: 기록 종료
2013. 7. 22 오후 11:08:17 madvirus.spring.chap01.LoggingAspect logging
정보: write메서드 실행 시간 : 0
```

- LoggingAspect 의 코드가 ArticleServiceImpl 클래스의write() 메서드 앞, 뒤로 실행되었음

스프링에서의 AOP

```
public Object logging(ProceedingJoinPoint joinPoint) throws Throwable {
                        log.info("기록 시작");
                        ....
                        Object retValue = joinPoint.proceed();
                        ....
                        log.info("기록 종료");
                        log.info(joinPoint.getSignature().getName() + "메서드 실행 시간 : "
                                + stopWatch.getTotalTimeMillis());
}
```

- articleService.write(new Article()) 코드가 실행되면 LoggingAspect.logging() 메서드가 실행됨
- 그런 뒤, joinPoint.proceed() 코드가 실행될 때 실제 ArticleServiceImpl 클래스의write() 메서드가 실행됨

스프링에서의 AOP

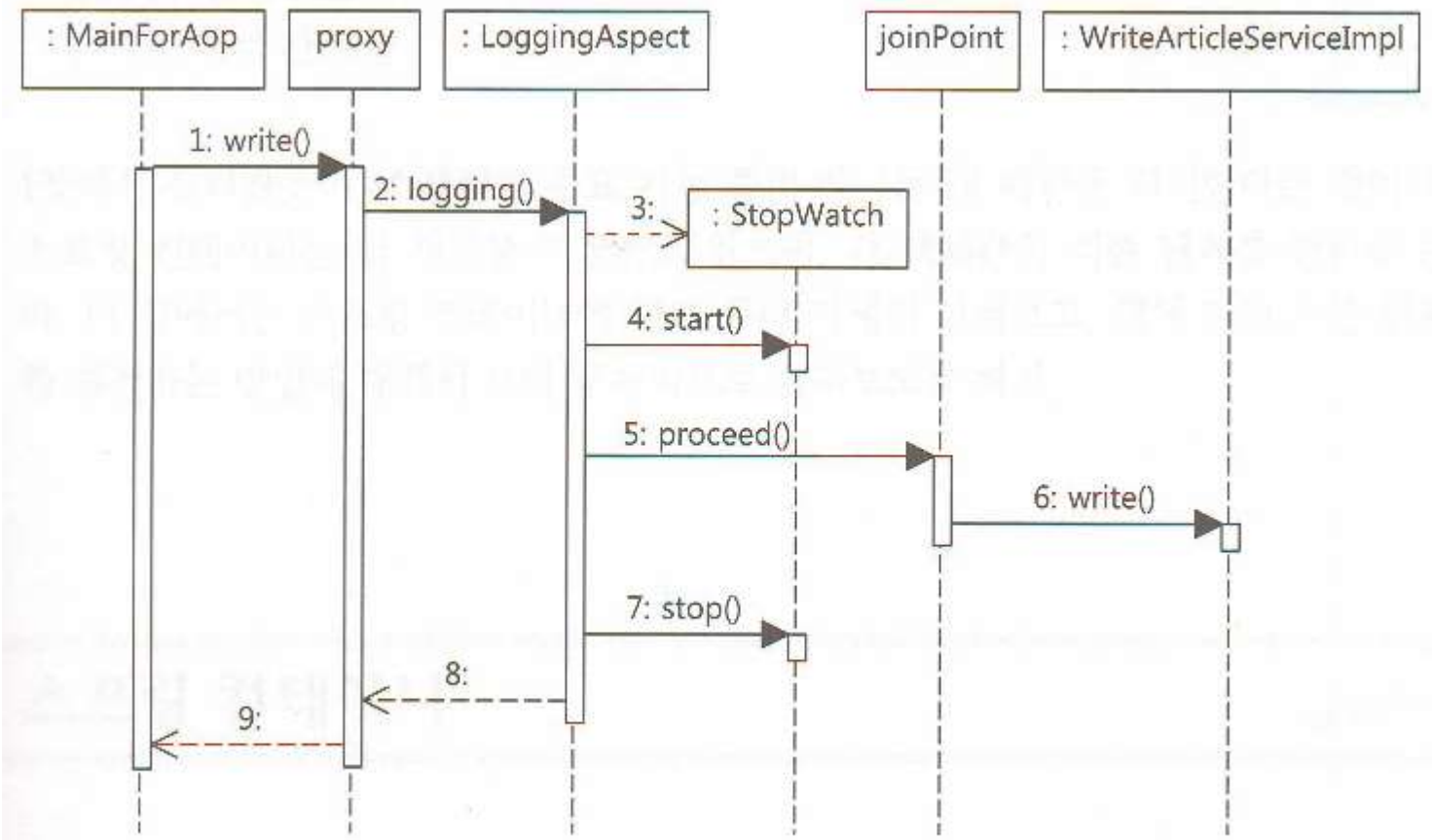


그림 1.8 LoggingAspect의 실행 순서

스프링에서의 AOP

- 스프링은 프록시를 이용하여 AOP를 구현
- MainForAop 클래스가 사용하는 객체는 ArticleServiceImpl 객체가 아니라 스프링이 런타임에 생성한 프록시 객체가 되며, 프록시가 내부적으로 다시 LoggingAspect 의 logging() 메서드를 호출하여 공통 관심사항이 실제 ArticleServiceImpl 객체에 적용됨
- 공통 관심사항(LoggingAspect)를 적용하는 과정에서 핵심 로직을 구현한 클래스(ArticleServiceImpl)의 코드를 변경하지 않았음
  - 단지 공통 관심사항을 구현한 Aspect 클래스를 작성하였고, 설정 파일을 이용하여 Aspect 를 핵심 로직을 구현한 클래스에 적용했을 뿐.
- 스프링은 JEE 어플리케이션을 구현하는 데 필요한 수준으로 AOP를 지원
  - 스프링 AOP 를 이용하면 핵심 로직 코드의 수정 없이 웹 어플리케이션에 보안, 로깅, 트랜잭션과 같은 공통 관심 사항을 AOP를 이용하여 간단하게 적용할 수 있음

예제2 - AOP

```
package com.di.aop;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.JoinPoint;
public class LoggingDAOAspect {
	private Log log = LogFactory.getLog(getClass());

	public void before(JoinPoint joinPoint) {
		log.info("[ 기록 시작, target="+ joinPoint.getTarget()+"]");
		log.info("메서드 - " +  joinPoint.getSignature().getName());

		Object[] argsArr=joinPoint.getArgs();
		for(int i=0;i<argsArr.length;i++) {
			log.info(i+"번째 매개변수 - " + argsArr[i]);
		}
	}

	public void afterReturing(JoinPoint joinPoint) {
		log.info("[ 기록 끝 target="+ joinPoint.getTarget()+"]");
	}
```

예제2 – AOP

```
    public void afterError(JoinPoint joinPoint) {
            log.info("DAO error : joinPoint="+ joinPoint);
            for(Object obj : joinPoint.getArgs()) {
                    log.info("DAO error : 매개변수 - " +  obj);
            }
    }

}
```

applicationContext.xml

```
<bean name="loggingAsp" class="com.di.aop.LogingAspect"></bean>
<aop:config>
        <aop:pointcut expression="execution(* *..*Service.*(..))"
                id="servicePointCut"/>
        <aop:aspect id="logAspect" ref="loggingAsp">
                <aop:around method="logging" pointcut-ref="servicePointCut"/>

        </aop:aspect>
</aop:config>

<bean name="loggingDaoAsp" class="com.di.aop.LoggingDAOAspect"></bean>
<aop:config>
        <aop:pointcut expression="execution(* *..*DAO*.*(..))"
                id="daoPointCut"/>
        <aop:aspect id="logDaoAspect" ref="loggingDaoAsp">
                <aop:before method="before" pointcut-ref="daoPointCut"/>
                <aop:after-returning method="afterReturing"
                        pointcut-ref="daoPointCut"/>
                <aop:after-throwing method="afterError"  pointcut-ref="daoPointCut"/>
        </aop:aspect>
</aop:config>
```

```
정보: [ 기록 시작, target=com.di.MySqlArticleDAO@400cff1a]
6월 12, 2018 5:32:23 오후 com.di.aop.LoggingDAOAspect before
정보: 메서드 - insert
6월 12, 2018 5:32:23 오후 com.di.aop.LoggingDAOAspect before
정보: 0번째 매개변수 - ArticleVO [no=2, title=생성자를 이용한 종속객체 주입]
Mysql DB에 ArticleVO [no=2, title=생성자를 이용한 종속객체 주입] 정보 insert됨!
6월 12, 2018 5:32:23 오후 com.di.aop.LoggingDAOAspect afterReturing
정보: [ 기록 끝 target=com.di.MySqlArticleDAO@400cff1a]
ArticleServiceImpl2 - 생성자를 이용한 의존객체주입!
6월 12, 2018 5:32:23 오후 com.di.aop.LoggingDAOAspect before
정보: [ 기록 시작, target=com.di.MySqlArticleDAO@400cff1a]
6월 12, 2018 5:32:23 오후 com.di.aop.LoggingDAOAspect before
정보: 메서드 - insert
6월 12, 2018 5:32:23 오후 com.di.aop.LoggingDAOAspect before
정보: 0번째 매개변수 - ArticleVO [no=3, title=setter를 이용한 종속객체 주입]
Mysql DB에 ArticleVO [no=3, title=setter를 이용한 종속객체 주입] 정보 insert됨!
6월 12, 2018 5:32:23 오후 com.di.aop.LoggingDAOAspect afterReturing
정보: [ 기록 끝 target=com.di.MySqlArticleDAO@400cff1a]
```

AOP 용어

- Advice – **언제** 공통 관심 기능을 핵심 로직에 적용할 지를 정의하고 있다.
  - 예) '메서드를 호출하기 전'에 공통기능(트랜잭션을 시작한다)을 적용한다는 것을 정의
- Joinpoint – Advice를 **적용 가능한 시점**을 의미함
  - 메서드 호출, 필드값 변경 등이 Joinpoint에 해당함
- Pointcut – Joinpoint 의 부분집합으로서 **실제로 Advice가 적용되는 Joinpoint** 를 나타냄
  - 스프링에서는 정규표현식이나 AspectJ의 문법을 이용하여 Pointcut을 정의할 수 있다
- Weaving – Advice를 핵심 로직 코드에 적용하는 것
  - 공통 코드를 핵심 로직코드에 삽입하는 것이 weaving
- Aspect – 여러 객체에 공통으로 적용되는 **공통 관심 사항**
  - 트랜잭션이나 보안 등이 Aspect의 예

스프링에서 구현 가능한 Advice 종류

종 류	설 명
Before Advice	대상 객체의 메서드 호출 전에 공통 기능을 실행한다.
After Returning Advice	대상 객체의 메서드가 예외 없이 실행한 이후에 공통 기능을 실행한다.
After Throwing Advice	대상 객체의 메서드를 실행하는 도중 예외가 발생한 경우에 공통 기능을 실행한다.
After Advice	대상 객체의 메서드를 실행하는 도중에 예외가 발생했는지의 여부와 상관없이 메서드 실행 후 공통 기능을 실행한다. (try-catch-finally의 finally 블록과 비슷하다.)
Around Advice	대상 객체의 메서드 실행 전, 후 또는 예외 발생 시점에 공통 기능을 실행하는데 사용된다.

Advice 정의 관련 태그

태 그	설 명
〈aop:before〉	메서드 실행 전에 적용되는 Advice를 정의한다.
〈aop:after-returning〉	메서드가 정상적으로 실행된 후에 적용되는 Advice를 정의한다.
〈aop:after-throwing〉	메서드가 예외를 발생시킬 때 적용되는 Advice를 정의한다. try-catch 블록에서 catch 블록과 비슷하다.
〈aop:after〉	메서드가 정상적으로 실행되는지 또는 예외를 발생시키는지 여부에 상관없이 적용되는 Advice를 정의한다. try-catch-finally 에서 finally 블록과 비슷하다.
〈aop:around〉	메서드 호출 이전, 이후, 예외 발생 등 모든 시점에 적용 가능한 Advice를 정의한다.

AspectJ의 Pointcut 표현식

```
<aop:config>
    <aop:pointcut id="servicePointcut"
      expression="execution(* *..*Service.*(..))" />

    <aop:aspect id="loggingAspect" ref="loggingAsp">
      <aop:around pointcut-ref="servicePointcut" method="logging" />
    </aop:aspect>
</aop:config>
```

- execution 명시자 – Advice를 적용할 메서드를 명시할 때 사용
- 기본 형식

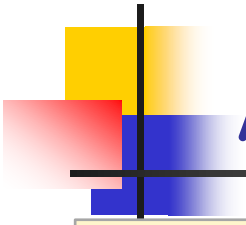
execution(수식어패턴? 리턴타입패턴 클래스이름패턴?이름패턴(파라미터패턴)

  - 수식어패턴 – 생략 가능한 부분, public, protected 등
  - 리턴타입패턴 – 리턴 타입을 명시
  - 클래스이름패턴, 이름패턴 – 클래스 이름 및 메서드 이름을 패턴으로 명시
  - 파라미터패턴 – 매칭될 파라미터에 대해서 명시함
    - 각 패턴은 '*' 을 이용하여 모든 값을 표현할 수 있다.
    - '..' 을 이용하여 0개 이상이라는 의미를 표현할 수 있다

예

execution(public void set*(..))
리턴타입이 **void**이고 메서드 이름이 **set** 으로 시작하고, 파라미터가 **0**개 이상인 메서드 호출

- execution(* madvirus.spring.chap05.*.*())
  madvirus.spring.chap05 패키지의 파라미터가 없는 모든 메서드 호출
- execution(* madvirus.spring.chap05..*.*(..))
  madvirus.spring.chap05 패키지 및 하위 패키지에 있는, 파라미터가 0개 이상인 메서드 호출
- execution(Integer madvirus.spring.chap05..WriteArticleService.write(..))
  리턴 타입이 Integer인 WriteArticleService 인터페이스의 write() 메서드 호출
- execution(* get*(*))
  이름이 get으로 시작하고 1개의 파라미터를 갖는 메서드 호출
- execution(* get*(*, *))
  이름이 get으로 시작하고 2개의 파라미터를 갖는 메서드 호출
- execution(* read*(Integer, ..))
  메서드 이름이 read로 시작하고, 첫 번째 파라미터 타입이 Integer이며, 1개 이상의 파라미터를 갖는 메서드 호출



AspectJ의 Pointcut 표현식

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)

- 반환하는 타입 패턴(ret-type-pattern), 이름 패턴, 파라미터 패턴을 제외한 모든 부분은 선택사항이다