

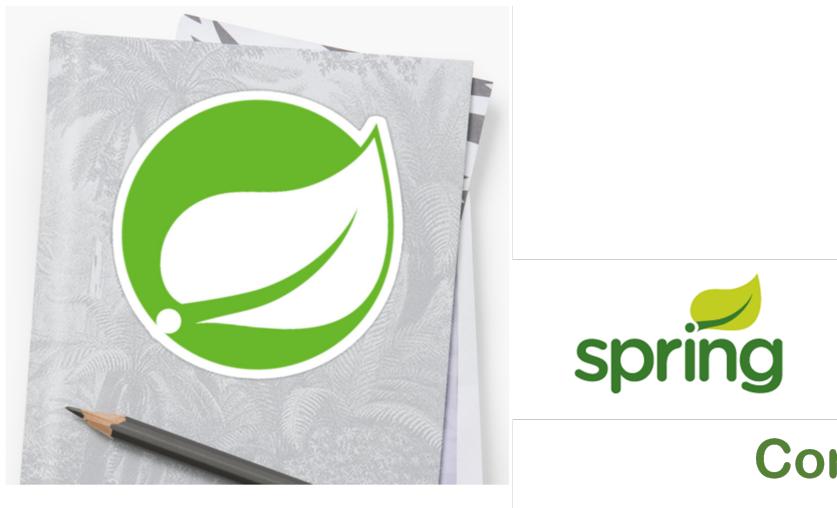
# Spring Framework Seminar

2017년 10월 22일 일요일 오전 12:13

작성자 : 구동완

This document describes the **concept** of Spring Framework.

- **Concept of Spring Framework**

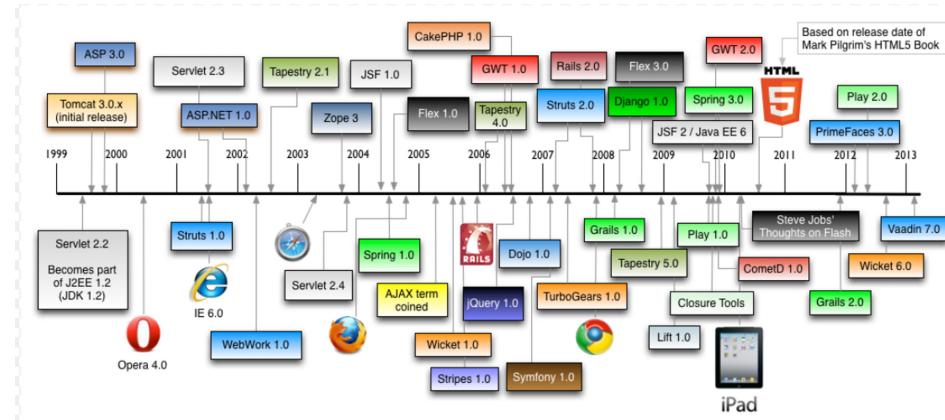


- **Table of Contents**

1. Spring (Spring Framework) Overview
2. Major Elements & DI
3. IoC
4. AOP
5. AOP Proxy
6. Dynamic Proxy
7. Patterns
8. AOP + Advice (Handler)
9. PointCut, JoinPoint
10. Bean Post Processor
11. Spring MVC (Model2)
12. Spring MVC (AOP with XML)
13. AOP with Annotation
14. Spring MVC Process
15. References

## 1 Spring (Spring Framework) Concept

### 1) Spring History



- Rod Johnson 2002년 집필작 Expert One-on-One J2EE Design and Development에 선보인 Code가 시초.
- 2003년 6월 최초 Apache 2.0 license로 공개.
- 2004년 3월 1.0버전 릴리즈, 2012년 3.1 버전 공개.

## 2) Spring Definition

- Definition Structure



- 스프링(Spring) = 스프링 프레임워크 (Spring Framework)

- 자바 엔터프라이즈 개발을 편하게 해주는 경량급 Open Source Application Framework for Java Platform  
- 자바 개발을 위한 프레임워크로 종속 객체를 생성해주고, 조립해주는 도구  
- 자바로 된 프레임워크로 자바SE로 된 자바 객체(Pojo)를 자바EE에 의존적이지 않게 연결해주는 역할.

- 연관 개념 (Spring Study 중 개인적으로 모르는 개념을 정리한 것)

• Java Platform은 Java API + JVM(Java Virtual Machine)으로 구성

- 개발 목적으로 크게 3가지로 구분 가능
  - i. Java SE(Java Standard Edition)
    - 1) 기본적인 자바개발환경 Platform
    - 2) JVM + API 개발환경
  - ii. Java EE(Java Enterprise Edition) - J2EE
    - 1) Java SE + Web Server 개발 환경
    - 2) JSP, Servlet, EJB, JMS 등의 기능
      - Servlet ? Client 요구 처리를 한 후 결과를 되돌려주는 Java 서버 모듈
    - 3) WAS Server라 부르며, JBoss Glass fish, Jboss, Apache, Jetty 등의 제품 존재.
  - iii. Java ME(Java Micro Edition) - J2ME
    - 1) Embedded 장비를 위한 개발 환경
    - 2) Java SE의 기능을 축소

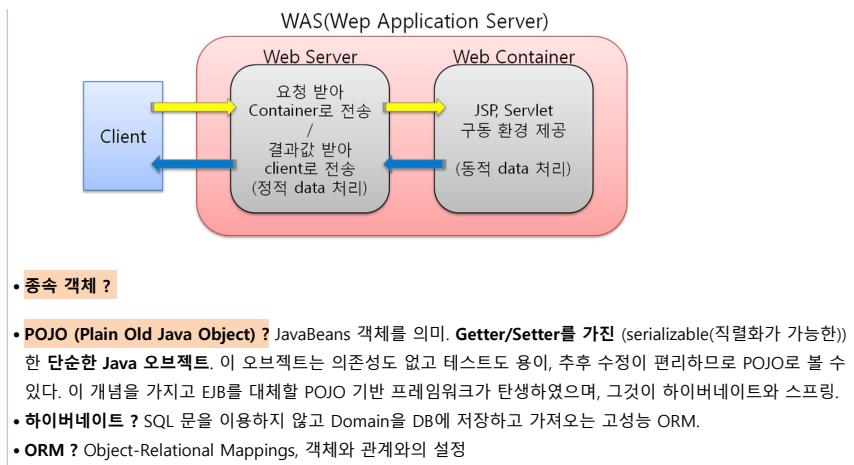
• Java는 실행 코드로 컴파일 되어 실행되는 것이 아니며, 인터프리터 방식을 채택, 즉, Text 형태 Source파일이 Class파일로 컴파일 된 후 이 Class파일을 JVM이 해석하여 실행하는 방식.

### • Open Source ?

1. No, 무료가 아닌 자유 Software, 기술 공개 차원, '자신이 필요에 따라 해당 소스 이용할 수 있다'의 개념.
2. 공개적 협업 (Public Collaboration)

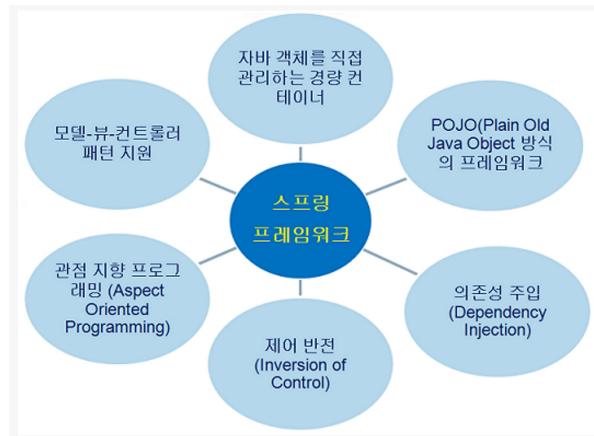
### • Framework ?

1. SW의 구체적인 부분에 해당하는 설계와 구현을 재사용이 가능하게끔 일련의 협업화된 형태로 클래스들을 제공하는 것
  2. 자주 쓰일 만한 기능들을 모아 놓은 유틸리티(클래스)들의 모음집
- JSP ? 자바서버 페이지(JavaServer Pages, JSP)는 HTML내에 자바 코드를 삽입하여 웹 서버에서 동적으로 웹 페이지를 생성하여 웹 브라우저에 돌려주는 언어
  - Java Beans ? Java Code들로 작성된 컴포넌트들, JSP page에서 비즈니스 로직을 제거하기 위한 방법으로 제공하는 기술. 일반 Java 객체라 보면 된다. EE에서만 부르는 용어이다.
  - JMS ? 자바 메시지 서비스, 네트워크를 통해 데이터를 송수신하는 자바 API. 엔터프라이즈 애플리케이션에 포함되어 있다.
  - Apache ? GET, POST, DELETE 등 이용해 요청을 하면 결과를 돌려주는 기능을 한다. 즉, 웹서버, HTTP웹서버, 정적인 HTML / JPG 등의 이미지를 HTTP Protocol을 통해 웹 브라우저에 제공한다.
  - WAS Server ? Web Application Server.
  - Tomcat ? WAS, 웹서버와 웹 컨테이너의 결합으로 다양한 기능을 컨테이너에 구현하여 다양한 역할을 수행할 수 있는 서버. DB와 연결되어 데이터를 주고 받거나 P/G으로 데이터 조작이 필요한 경우 활용해야 한다.



### 3) Spring 특징

- 크기 / 부하 측면에서 경량.
- 제어 역행(IoC) 기술을 통해 Application의 느슨한 결합을 도모.
- AOP 관점지향 Programming을 위한 지원
- Application Object의 Life Cycle과 설정을 포함하고 관리 --> 일종의 컨테이너(Container)
- 간단한 컴포넌트로 복잡한 Application 구성 / 설정 가능
- 동적 웹 사이트 개발을 위한 프레임워크
- 대한민국 전자정부 표준 프레임워크의 기반 기술



- 연관 개념

• **IoC** ? 제어의 역전 (Inversion of Control), 어떠한 일을 하도록 만들어진 프레임워크에 제어의 권한을 넘김으로서 Client 코드가 신경 써야 할 것을 줄이는 전략. (제어가 역전되었다.)  
일반적으로는 Client P/G에서 Library 메소드를 호출해서 사용하는 것을 의미, 역전이라 함은 그 반대, 즉, 프레임워크의 메소드가 Client 코드를 호출한다는 데 있다. Ex. 인터페이스 - 인터페이스 구현클래스  
방법은 나의 메소드를 프레임워크의 event, delegate에 등록하는 것, 또 다른 방법은 프레임워크 상에 정의된 interface / abstract를 나의 Code에서 구현, 상속한 후 프레임워크에 넘겨주는 것.

**• 의존성 ?**

**• 의존성 주입 (Dependency Injection) ?**

• **델리게이트 (Delegate)** ? 메소드를 대신해서 호출하는 역할, 메소드의 대리인

**• AOP ? 관점 지향 프로그래밍**

• **대한민국 전자정부** ? 정보기술을 활용하여 행정/공공기관의 업무를 전자화하여 기관 상호 간의 행정 업무 및 국민에 대한 행정 업무를 효율적으로 수행하는 정부

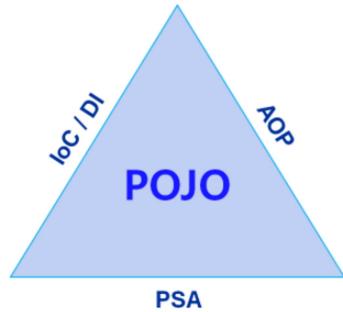
• **컨테이너(Container)** ? 기능들의 집합

## 2 Major Elements & DI

### 1) Spring Core Concept

- DI
- IoC
- AOP & AOP Proxy
- AOP in Spring

- a. 주요 구성 요소 (IoC / DI, AOP, PSA)



- 연관개념

- **PSA** ? Portable Service Abstractions, (쉬운) 서비스 추상화, 성격이 비슷한 여러 종류의 기술을 추상화하고 일관된 방법으로 사용할 수 있도록 지원하는 개념  
예) 트랜잭션 서비스 추상화 : 여러 DB를 사용한다고 하면 Global 트랜잭션 방식을 사용
- **트랜잭션**
  1. DB 내에서 한꺼번에 수행되어야 할 일련의 연산들 (전부 되거나 안되거나)
  2. DBMS 또는 유사 시스템에서의 상호작용의 단위
  3. DB 상태를 변화시키기 위해서 수행하는 작업의 단위

모든 연산은 반드시 한꺼번에 완료가 되어야 하며 그렇지 않은 경우는 모두 취소되어야 하는 원자성을 가짐
- **원자성** ? 분리할 수 없는 하나의 단위, 즉, 작업은 모두 완료 또는 취소 되어야 한다.

## 2) DI(Dependency Injection, 의존성 주입)

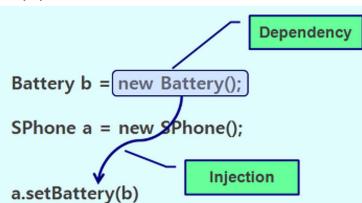
- 일체형

- **Composition** : HAS-A 관계
- A가 B를 생성자에서 생성하는 관계

- 분리 / 도킹(부착)형

- **Association** 관계
- A 객체가 다른 사람이 만든 B 객체를 사용
- Code의 예  
Battery b = new Battery();  
Phone p = new Phone();  
p.setBattery(b);

- 도식화



- 정리

- 분리형의 경우 A, B를 개별적으로 세팅해줘야 하며, 내가 원하는 다른 것으로 바꾸어 부착 가능하다. 이것이 곧 DI 개념.  
분리형으로 개발 시 객체 간 결합도를 낮출 수가 있다. 즉, DI 사용목적은 객체 간의 결합도를 낮추기 위함이다.

- 연관개념

- 인터페이스는 기능적이 아닌 의미적으로 구성한다. (ex. 회원가입)
- "!"라는 인터페이스를 구현하고 있는 A, B가 있다라고 가정  
기준에 잘못 설계된 setB(B b)가 있다. 이것의 문제점은 B클래스 객체 말고 다른 객체는 받을 수가 없다는 것이다.  
즉, 파라미터 클래스 타입이 B가 아니라 인터페이스 "!"였다면 A, B 둘다 받을 수 있었을 것이다.  
구조를 개선하고자 한다면 B클래스를 확장한다. 즉, B클래스를 상속받는 C를 하나 더 만들고 실제 구현은 C 내에서 한다.

### a. DI 종류

- i. **Setter** Injection
- ii. **Construction** Injection

### b. Spring에서의 DI



- i. 소단위 부품(모듈)을 활용하여 제품(Component)을 조립 --> 스프링(Spring)
  - ii. 생성자 내에서 생성하는 것이 아니라 필요한 부품을 생성
  - iii. 작은 것(작은 부품, 작은 모듈)에서 큰 것(큰 부품, 큰 모듈)로 이동한다. --> IoC 개념  
(기본적인 완제품 제작 순서와는 다르게 작은 부품에서 큰 부품으로,
- 제품 생성 순서가 역순 (IoC), 스프링은 컨테이너라는 곳에 담아서 처리하여 스프링을 IoC 컨테이너라 한다.)
- 정리 : 스프링에서의 DI의 의미 --> 부품 생성 및 제품 조립하는 공정과정을 대신해 주는 라이브러리.  
생성하기 원하는 객체를 XML에 기술하고, 그 것과 의존성을 보관하는 일을 처리.  
이러한 데이터를 보관하는 공간을 컨테이너. (IoC 컨테이너)

#### c. DI 구현

- 객체 생성, 도킹(부착)에 대한 내용이 소스 코드 상에 있는 것이 아닌 별도의 XML 설정 파일에 분리하여 존재한다.
- 즉, Java 소스 compile 작업없이 XML 변경만으로 내용 변경 가능

- Source Code

• Java (DI) - 이해를 돋기 위한 예제

```

Record record = new SprRecord();
RecordView view = new SprRecordView();
view.setRecord(record); // Injection
view.input();
view.print();

```

• XML (Spring DI)

1. view에 대한 객체만을 요청했을 뿐, 실제 내부적인 사항은 Java Code상에 드러나지 않는다.
  2. 새로운 클래스의 bean 객체를 만들어 XML에 주입만 해줘도 기존 소스 변경 없이 새로운 형태의 객체 적용이 가능
- ```

<bean id="record" class="di.SprRecord"></bean> // Bean 객체 생성
<bean id="view" class="di.SprRecordView">           // Bean 객체 생성
    <property name="record" ref="record"></property> // setRecord() 호출
</bean>

```
3. Bean 객체는 반드시 클래스 사용. 인터페이스 / 추상 클래스는 객체 생성 불가

• Java(XML --> Parsing --> Container)

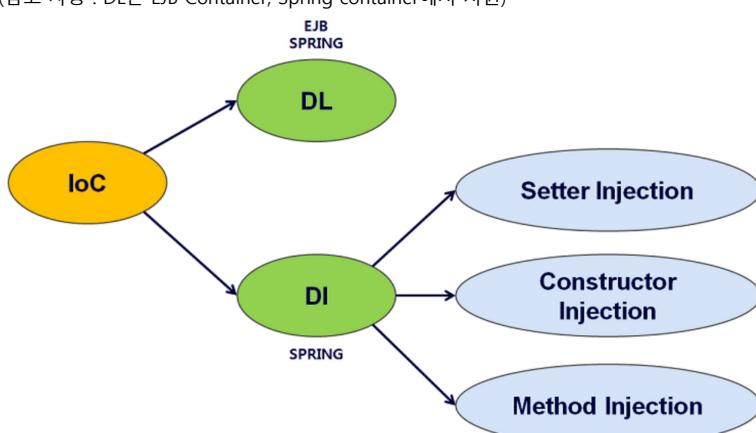
```

ApplicationContext ctx = new ClassPathXmlApplicationContext("config.xml");
RecordView = (RecordView) ctx.getBean("view");

```

### 3 IoC Container (Spring Container)

- 제어의 역전 IoC (Inversion Of Control) --> 외부 (컨테이너)에서 제어 한다. 거꾸로 가는 순서를 의미.
  - 빈(Bean) : 스프링이 제어권을 가지고 직접 만들고 관계를 부여하는 오브젝트
  - 빈 팩토리(Bean Factory) : 빈(오브젝트)의 생성과 관계 설정 제어를 담당하는 IoC 오브젝트
  - IoC(Inversion Of Control), DL(Dependency Lookup), DI(Dependency Injection) 관계
- (참고 사항 : DL은 EJB Container, Spring container에서 지원)



- 연관개념

- EJB (Enterprise Java Bean) ? 효율적으로 서버 관리를 하고 프로그램 관련 문제들을 알아서 처리한다는 개념.  
(연결 관계가 복잡, 무겁고, 독립적이지 못하다.)
- DI 3 가지
  1. Setter Injection
  2. Constructor Injection
  3. Method Injection ? Singleton 인스턴스와 non Singleton 인스턴스의 의존관계를 연결시킬 필요가 있을 때 사용하는 방법.

## 4 AOP (Aspect Oriented Programming)

- Definition

- Aspect를 만드는 프로그램 방법
- Aspect 지향 프로그램
- 관심 지향 프로그래밍

- 도입 배경 ?

- 주 업무가 아닌 부가적인 업무가 강한 응집력을 가지고 있는 경우, 소스 관리 및 개발 업무 진행이 복잡해지고 어려워진다.
- 보조 업무 코드를 주 업무 코드에서 별도로 분리하여 작성하고, 필요할 때에만 Docking하여 사용하는 것은 어떨까? 라는 발상

- 사용 이유 ?

- 트랜잭션(핵심 & 부가 기능)의 분리
- 클래스들이 공통으로 갖는 기능이나 절차 등을 하나의 것으로 묶어 빼내어 별도로 관리하려는 목적
- 영향받는 객체 간의 결합도를 낮추는 것.

- Spring Framework에서 Aspect ?

- 주 업무가 아닌 업무 (보조업무 : 로그, 트랜잭션, 보안처리)

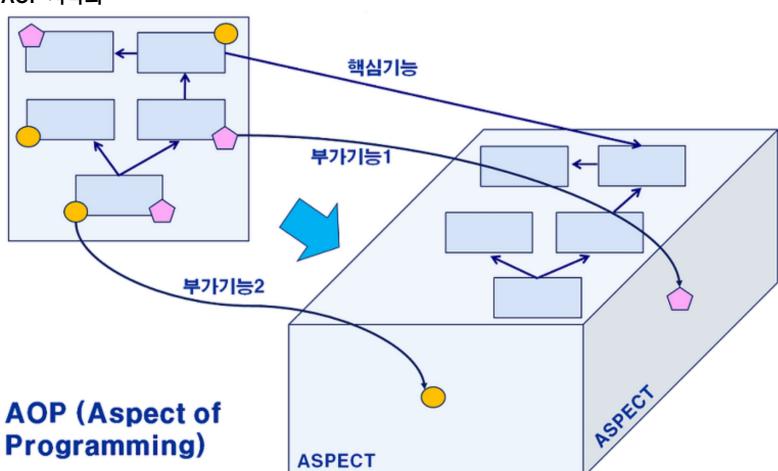
- AOP의 구현이란?

- 보조 업무를 주 업무를 처리하는 코드에서 분리하는 것.

- AOP 장점

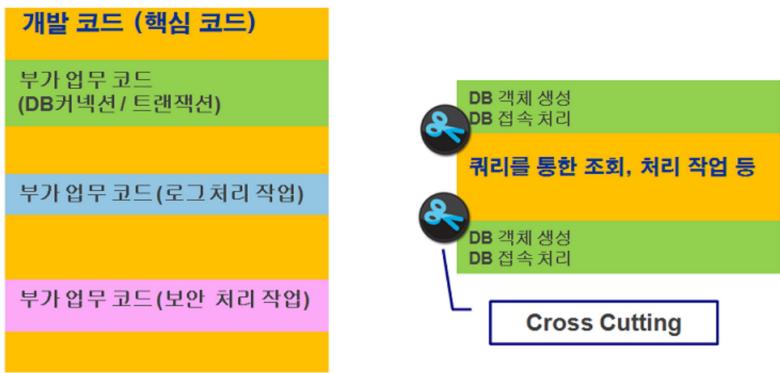
- 전체 코드 곳곳에 흩어져 있는 다양한 관심 사항이 한 곳으로 응집된다.
- 코드가 깔끔해지고 가독성이 높아진다.

- AOP 시작화



- AOP는 실제 코드에서 어떻게 처리되는가?

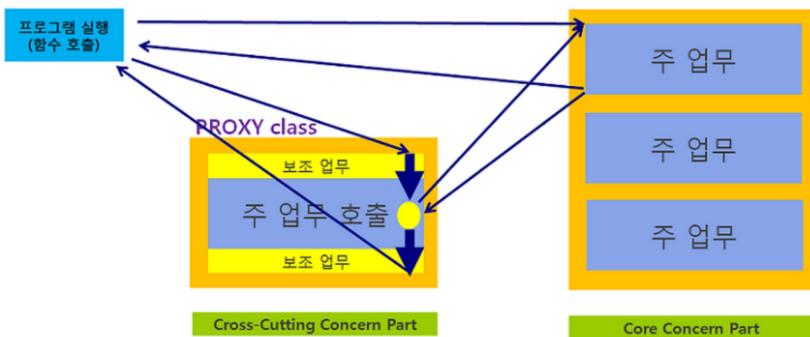




- 기존 개발방식(왼쪽)은 보조 업무를 담당하는 코드가 주 업무 코드 사이사이에 포함되어 있었다.
- 기존 개발방식의 문제점
  - 동일한 작업 반복(DB 객체 생성, 접속처리..)
  - 보조 업무 코드 변경 시 해당 보조 업무를 사용하는 모든 주 업무 코드의 소스 수정 불가피
  - 주 업무 코드보다 더 많은 양의 보조 업무 코드 (Ex. DB 객체 생성, 접속처리, 예외처리, DB Close)
- 핵심 코드(핵심 관심사) - **Core(Primary) Concern**, 부가/보조 업무 코드(횡단 관심사) - **Cross-Cutting Concern**

## 5 AOP Proxy

- Client 요청을 받아주어 처리하는 **대리자**
- 함수 호출자는 주요 업무가 아닌 보조 업무를 프록시에게 맡기고, 프록시는 내부적으로 이러한 보조업무를 처리한다.
- 필요에 따라 해당 프록시를 추가해서 보조 업무를 수행하면 된다. (보조 업무의 탈부착이 쉬워진다.)
- 호출 및 처리 순서 : 클라이언트 --> 프록시 --> Target
- AOP Proxy 시작화



- 그림 설명
  - 프록시 호출
  - 보조 업무 처리
  - 프록시 처리 메소드가 실제 구현 메소드(주 업무) 호출
  - 제어권이 다시 프록시 메소드로 넘어오고 나머지 보조 업무 처리
  - 처리 작업 완료 후 호출 메소드로 반환

- Proxy 사용 목적
  - 클라이언트가 타깃(Target)에 접근하는 방법을 제어.
  - 타깃에 부가적인 기능을 부여.

- Proxy 구현
  - AOP 개념은 스프링에 한정되어 사용되는 개념이 아님.

```
// 사칙 연산을 정의하는 인터페이스
package aoptest;

public interface Calculator {
    public int add(int x, int y);
    public int subtract(int x, int y);
    public int multiply(int x, int y);
    public int divide(int x, int y);
}

// 사칙 연산을 구현하는 클래스
package aoptest;

public class myCalculator implements Calculator {
```

```

@Override
public int add(int x, int y) {
    return x + y;
}

@Override
public int subtract(int x, int y) {
    return x - y;
}

@Override
public int multiply(int x, int y) {
    return x * y;
}

@Override
public int divide(int x, int y) {
    return x / y;
}
}

// 실제 위의 사칙연산 클래스를 사용하는 예제 코드
public static void main(String[] args) {
    Calculator cal = new myCalculator(); // 다형성
    System.out.println(cal.add(3, 4)); // add 메서드를 호출하여 3 + 4 결과를 출력
}

```

#### [가정]

- 현재 구현 상태에서 실제 사칙 연산을 하는데에 있어서 실제 소요되는 시간 측정이라는 기능이 필요하다고 가정.

위의 예제 코드에서 cal.add(3, 4) 를 사용하고 있으니, 이 시간 측정을 위한 코드를 일단 myCalculator 클래스의 add 메서드에 추가하면 아래와 같다

```

public class myCalculator implements Calculator {

    @Override
    public int add(int x, int y) {

        // 보조 업무 (시간 측정 시작 & 로그 출력)
        Log log = LogFactory.getLog(this.getClass());
        StopWatch sw = new StopWatch();
        sw.start();
        log.info("Timer Begin");

        int sum = x + y; // 주 업무 (덧셈 연산)

        // 보조 업무 (시간 측정 끝 & 측정 시간 로그 출력)
        sw.stop();
        log.info("Timer Stop – Elapsed Time : " + sw.getTotalTimeMillis());

        return sum;
    }
    ...
}

```

#### [문제점]

- 시간 측정을 위한 이러한 코드를 add 메서드 뿐만 아니라, subtract, multiply, divide 메서드에도 추가해 주어야 하고, 설령 측정 방식이 달라지거나 로그 출력 내용이 변경 되기라도 한다면 모든 메서드를 수정해야 한다. 게다가 연산을 위한 주 업무 코드를 수정하는 것도 아니다.

주 업무와 보조 업무를 분리(크로스 컷팅, Cross Cutting)하고 보조 업무를 프록시(Proxy)에게 넘긴다

```

/* 보조 업무를 처리할 프록시 클래스 */
public class LogPrintHandler implements InvocationHandler {
    // InvocationHandler Interface를 구현한 객체는 invoke 메소드를 구현해야 한다.
    private Object target; // 객체에 대한 정보

    public LogPrintHandler(Object target) { // 생성자
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // 보조 업무 (시간 측정 및 로그 출력)
        Log log = LogFactory.getLog(proxy.getClass());
        StopWatch sw = new StopWatch();
        sw.start();
        log.info(method.getName() + " Method Start");

        Object result = method.invoke(target, args);

        log.info(method.getName() + " Method End – Elapsed Time : " + sw.getTotalTimeMillis());
        sw.stop();

        return result;
    }
}

```

```

}

@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    Log log = LogFactory.getLog(this.getClass());
    StopWatch sw = new StopWatch();
    sw.start();
    log.info("Timer Begin");

    int result = (int) method.invoke(target, args); // 주 업무를 invoke 함수를 통해 호출

    sw.stop();
    log.info("Timer Stop - Elapsed Time : " + sw.getTotalTimeMillis());
    return result;
}

public static void main(String[] args) {
    Calculator cal = new myCalculator(); // 대형성

    // Proxy.newInstance Parameter : 1. Loader, 2. Interface, 3. Handler (보조 업무)
    Calculator proxy_cal = (Calculator) Proxy.newInstance(
        cal.getClass().getClassLoader(), // Loader
        cal.getClass().getInterfaces(), // Interface
        new LogPrintHandler(cal)); // Handler (보조 업무를 구현하고 있는 실제 클래스)

    System.out.println(proxy_cal.add(3, 4)); // 주 업무 처리 클래스의 add 메서드를 호출
}

```

#### [주의 클래스와 메서드에 대한 설명]

- 1. InvocationHandler 인터페이스를 구현한 객체는 invoke 메서드를 구현해야 함.
- 해당 객체에 의하여 요청 받은 메서드를 리플렉션 API를 사용하여 실제 타깃이 되는 객체의 메서드를 호출해준다.
- (실제 LogPrintHandler 클래스의 invoke 메서드 내에서 `method.invoke(target, args)` 메서드를 호출하는데, 이는 주 업무의 메서드를 호출하는 것. main 메서드에서 Proxy.newInstance 를 통해 주 업무를 처리할 클래스 (myCalculator)와 보조 업무를 처리할 Proxy 클래스를 결합).
- `cal(변수)` 실제 객체를 `proxy_cal(변수)` 객체에 핸들러를 통해서 전달. (`LogPrintHandler()`)
- `getClassLoader()` : 동적으로 생성되는 다이내믹 프록시 클래스의 로딩에 사용할 클래스 로더
- `getInterfaces()` : 구현할 인터페이스
- `LogPrintHandler(cal)` : 부가 기능과 위임 코드를 담은 핸들러

#### - Example

```

Hello proxiedHello = (Hello)Proxy.newInstance(
    getClass().getClassLoader(), --> 동적으로 생성되는 다이내믹프록시 클래스의 로딩에 사용할 클래스 로더
    new Class[] {Hello.class}, <-- 구현할 인터페이스
    new UppercaseHandler(new HelloTarget())); --> 부가기능과 위임코드를 담은 핸들러

```

2. 주 업무 클래스의 메서드를 호출하게 되면 프록시 클래스의 invoke 메서드가 호출되어 자신의 보조 업무를 처리하고, 주 업무의 메서드를 호출한다.
3. invoke() : 메서드를 실행시킬 대상 오브젝트와 파라미터 목록을 받아서 메서드를 호출한 뒤에 그 결과를 Object 타입으로 돌려준다.

#### [실제 프록시 및 주 업무(타깃) 처리 순서]

- 호출 및 처리 순서 : 클라이언트 ---> 프록시 ---> 타깃

- AOP 개념은 스프링에 한정되어 사용되는 개념이 아님.
- 실제 위의 코드는 AOP Proxy의 구현(객체를 생성하고, 값을 주입)하는 것을 순수 JAVA 코드 단에서 처리한 것임. (스프링 개념 없이)
- 스프링을 통해서라면 AOP 는 XML과 어노테이션(Annotation)을 통해서 더 쉽게 구현할 수 있음.

## 6 Dynamic Proxy (동적 프록시)

### • 프록시(Proxy)의 단점

1. 매 번 새로운 클래스 정의 필요
  - 실제 프록시 클래스(Proxy Class)는 실제 구현 클래스와 동일한 형태를 가지고 있기 때문에 구현 클래스의 Interface를 모두 구현해야 함.
2. 타깃의 인터페이스를 구현하고 위임하는 코드 작성의 번거로움.

- 부가기능이 필요없는 메소드도 구현해서 타깃으로 위임하는 코드를 일일이 만들어줘야 함.
- 복잡하진 않지만 인터페이스의 메소드가 많아지고 다양해지면 상당히 부담스러운 작업
- 타깃 인터페이스의 메소드가 추가되거나 변경될 때마다 함께 수정해줘야 한다는 부담도 있음.

### 3. 부가기능 코드의 중복 가능성.

- 프록시를 활용할 만한 부가기능, 접근제어 기능 등은 일반적으로 자주 활용되는 것들이 많기 때문에 다양한 타깃 클래스와 메소드에 중복되어 나타날 가능성이 높음.  
(특히 트랜잭션(Transaction) 처리는 데이터베이스를 사용하는 대부분의 로직에서 적용되어야 할 필요가 있음.)
- 메소드가 많아지고 트랜잭션 적용의 비율이 높아지면 트랜잭션 기능을 제공하는 유사한 코드가 여러 메서드에 중복돼서 나타날 수 있음.

#### • 해결책

- 다이내믹 프록시(Dynamic Proxy)를 이용하는 것! (JDK Dynamic Proxy)
- 다이내믹 프록시란?  
  - > 런타임 시 동적으로 만들어지는 오브젝트
  - > 리플렉션 기능을 이용해서 프록시 생성 (java.lang.reflect)
  - > 타깃 인터페이스와 동일한 형태로 생성.
  - > 팩토리빈(FactoryBean)을 통해서 생성.
- 스프링의 빈은 기본적으로 클래스 이름(Class name)과 프로퍼티(Property)로 정의.
- 스프링은 지정된 클래스 이름을 가지고 리플렉션을 이용해서 해당 클래스의 오브젝트를 생성.
- 참고로 스프링은 지정된 클래스 이름을 가지고 리플렉션을 이용하여 해당 클래스의 오브젝트를 생성함.
- 이 외에도 팩토리빈(FactoryBean)과 프록시 팩토리빈(Proxy FactoryBean)을 통해 오브젝트를 생성할 수 있음. 팩토리빈 이런 스프링을 대신해서 오브젝트의 생성 로직을 담당하도록 만들어진 특별한 빈을 뜻함.

#### ○ [추가 정리]

데코레이터(Decorator) 또는 프록시 패턴(Proxy Pattern)을 적용하는 데에 따른 어려움은 부가기능을 구현할 클래스가 부가기능과 관계없는 메서드들도 인터페이스에 선언된 메서드라면 전부 구현해야 하는 번거로움과 부가기능과 관련된 메소드들에 구현되는 코드 중복을 들 수 있음.

--> 이는 자바의 리플렉션(Reflection)에서 제공하는 다이내믹 프록시(Dynamic Proxy)를 활용하여 해결할 수 있음.

1. Proxy.newProxyInstance() 를 통한 프록시 생성.
  2. Proxy.newProxyInstance() 를 호출할 때 전달하는 InvocationHandler 인터페이스의 단일 메소드인 invoke()에 부가 기능을 단 한번만 구현함으로써 코드 중복 해결.
- => 다이나믹 프록시 오브젝트는 클래스 파일 자체가 존재하지 않음.  
=> 빈 오브젝트로 등록 불가.  
=> 팩토리빈 인터페이스 활용. (팩토리빈 인터페이스를 구현한 클래스를 빈으로 등록)

## 7 Patterns

- 패턴은 말 그대로 어떤 일정한 형태나 양식 또는 유형
- 패턴이라는 개념은 스프링 프레임워크에 한정된 것이 아니라 개발 디자인(Development Design)에 대해 사용되는 일반적인 개념 중의 하나.
- 개발 디자인 패턴에는 다음의 2가지가 존재

#### • 데코레이터 패턴

1. 타깃의 코드에 손 대지 않고, 클라이언트가 호출하는 방법도 변경하지 않은 채로 새로운 기능을 추가할 때 유용한 방법.
2. 핵심 코드에 부가적인 기능을 추가하기 위해서 런타임시 다이나믹하게 추가되는 프록시를 사용. 즉 동일한 인터페이스를 구현한 여러 개의 객체를 사용하는 것.
3. 주어진 상황 및 용도에 따라 어떤 객체에 책임을 덧붙이는 패턴. 기능 확장이 필요할 때 서브클래스(Subclassing) 대신 쓸 수 있는 유연한 대안이 될 수 있음.
4. 동적으로 객체의 추가적인 기능들을 가진 객체를 덧붙여 꾸밈.

#### • 프록시 패턴

1. 타깃의 기능 자체에는 관여하지 않으면서 접근하는 방법을 제어해주는 프록시를 이용하는 방법.

위의 2가지 패턴의 차이점은 프록시의 경우는 실제 실행될 타깃을 확장하거나 기능을 추가하는 것이 아니라, 단지 타깃에 접근하는 방법 자체를 프록시를 통하여 가능하게 하는 것이고, 데코레이터는 실행 타깃의 확장을 의미함.

#### • 데코레이터 패턴 예제

- 윈도우에 대한 인터페이스(interface Window)와 클래스(class SimpleWindow),
- 그리고 수직 스크롤바가 있는 윈도우 클래스(class VerticalScrollBarDecorator), <- 데코레이터
- 수평 스크롤바가 있는 윈도우 클래스(class HorizontalScrollBarDecorator) <- 데코레이터
- 기존 윈도우(Simple Window) 클래스를 감싸(implements) 스크롤이 추가된 클래스를 새로 재정의함.  
(기존 클래스에 장식(데코레이팅)된 형태로 클래스를 정의 )

```

// Windows Interface & Simple Window Class
interface Window {
    public void draw(); // draws the Window
    // returns a description of the Window
    public String getDescription();
}

class SimpleWindow implements Window {
    public void draw() {
        // draw window
    }
    public String getDescription() {
        return "simple window";
    }
}

// Decorators
abstract class WindowDecorator implements Window {
    protected Window decoratedWindow; // the Window being decorated
    public WindowDecorator (Window decoratedWindow) {
        this.decoratedWindow = decoratedWindow;
    }
}

class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        drawVerticalScrollBar();
        decoratedWindow.draw();
    }
    private void drawVerticalScrollBar() { // draw the vertical scrollbar
    }
    public String getDescription() {
        return decoratedWindow.getDescription() + ", including vertical scrollbars";
    }
}

class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        drawHorizontalScrollBar();
        decoratedWindow.draw();
    }
    private void drawHorizontalScrollBar() { // draw the horizontal scrollbar
    }
    public String getDescription() {
        return decoratedWindow.getDescription() + ", including horizontal scrollbars";
    }
}

// Decorator Pattern Example
public class DecoratedWindowTest {
    public static void main(String[] args) {
        // create a decorated Window with horizontal and vertical scrollbars
        Window decoratedWindow = new HorizontalScrollBarDecorator (
            new VerticalScrollBarDecorator(new SimpleWindow()));

        // print the Window's description
        System.out.println(decoratedWindow.getDescription());
    }
}

```

- o **데코레이터 패턴의 단점**

- 잡다한 클래스가 많아지고, 겹겹이 에워싼 형태의 구조로 구조가 복잡해지면 객체의 정체를 알기 어려움.

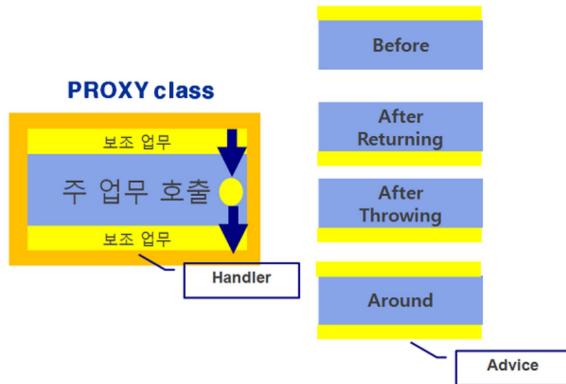
- o **데코레이터 패턴의 장점**

- 기존 코드는 수정하지 않고, 확장 및 추가가 가능함.

## 8 스프링에서의 AOP + Advice

### 1. 개념

- 스프링은 4가지 형태의 핸들러(어드바이스, Advice)를 제공.
- 스프링에서는 핸들러를 어드바이스(Advice)라는 개념으로 사용.



- JAVA 레벨이 아닌 XML을 통해 객체를 생성하고 인젝션(Injection)함.

```
package aoptest;
public class LogPrintAroundAdvice implements MethodInterceptor {  
    public static void main(String[] args) {  
        Calculator cal = new TestCalculator();  
        Calculator proxy = (Calculator) Proxy.newProxyInstance(  
            cal.getClass().getClassLoader(),  
            cal.getClass().getInterfaces(),  
            new LogPrintHandler(cal));  
  
        System.out.println(proxy.add(3, 4));  
    }  
}
```

```
<?xml version="1.0" encoding="UTF-8"?>  
  
<bean id="cal" class="aoptest.TestCalculator"></bean> // Target  
  
<bean id="logPrintAroundAdvice" class="aoptest.LogPrintAroundAdvice"/> // Advice  
  
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">  
    <property name="proxyInterfaces">  
        <list>  
            <value>aoptest.Calculator</value>  
        </list>  
    </property>  
    <property name="target" ref="cal"></property>  
    <property name="interceptorNames">  
        <list>  
            <value>logPrintAroundAdvice</value>  
        </list>  
    </property>  
</bean>
```

#### [코드 간략 설명]

- 스프링 프레임워크에서 제공해주는 핸들러(어드바이스) : MethodInterceptor
- MethodInterceptor : Around Advice 사용할 때, 구현해야 하는 인터페이스.
- MethodInterceptor 는 InvocationHandler 와 동일한 기능을 수행하나 차이점은 InvocationHandler 의 경우에는 프록시할 타깃 객체를 InvocationHandler 인터페이스 구현 객체가 알고 있어야 하며, MethodInterceptor 의 경우에는 MethodInvocation 객체가 타깃 객체와 호출된 메소드 정보를 모두 알고 있음.
- 인터페이스(Interface)는 스프링 프레임워크에서 알아서 감지(Auto Detecting)하여 연결해줌. 그러므로 위의 Proxy Bean 객체 XML 코드에서 aoptest.Calculator 부분(파란색)은 제거해도 무방함.
- 스프링 프레임워크는 Invoke 대신 proceed 메서드를 호출하여 주 업무 프로세스를 호출함. / 또한 스프링 프레임워크는 핸들러(Handler) 대신 어드바이스(Advice)를 생성.

#### [어드바이스(Advice)란?]

- 타깃 오브젝트(Target Object)에 적용할 부가 기능을 담은 오브젝트
- 메인 업무에 보조적으로 추가될 보조 업무.

[XML 레벨로 보조 업무 연결 코드가 이동하면서 JAVA 코드가 단순해짐]

```
public static void main(String[] args) {  
  
    /*Calculator cal = new TestCalculator();  
    Calculator proxy = (Calculator) Proxy.newProxyInstance(  
        cal.getClass().getClassLoader(),  
        cal.getClass().getInterfaces(),  
        new LogPrintHandler(cal)); */  
  
    ApplicationContext ctx = new ClassPathXmlApplicationContext("config.xml");  
    Calculator proxy = (Calculator) ctx.getBean("proxy");  
  
    System.out.println(proxy.add(3, 4));  
}
```

- 위의 X 표시된 코드는 주석처리하고 AOP를 활용하여 Advice를 적용한 JAVA 코드
- 객체의 생성과 조립을 메인 함수에서 코드에서 관리하였지만, 부가적인 파트(객체 생성과 의존성)는 XML 파일로 이동시킴.
- XML 빈 객체의 이름 및 속성을 변경함으로써, 기존 코드에 바로 적용 가능.
- 부가 코드와 주 코드를 분리하여 필요할 때 도킹(Docking)하여 사용 가능.
- 위와 같은 동적 프록시(Dynamic Proxy)는 ApplicationContext의 getBean 메서드를 통해서 개체를 얻어옴.

## 2. 스프링 프레임워크에서 4가지 형태의 Advice

Before	<b>MethodBeforeAdvice – before Method</b> 주 업무 이전에 처리해야 할 필요가 있는 부가 작업을 처리 <code>public void before(Method method, Object[] args, Object target)</code>
After Returning	<b>AfterReturningAdvice – afterReturning Method</b> 주 업무 이전에 처리해야 할 필요가 있는 부가 작업을 처리 <code>public void afterReturning(Object returnValue, Method method, Object[] args)</code>
After Throwing	<b>ThrowsAdvice – afterThrowing Method</b> 예외가 발생한 경우, 그 예외를 어떻게 처리해야 하는지에 대한 공통 처리 모듈 작성 <code>public void afterThrowing(IllegalArgumentException e)</code>
Around	<b>MethodInterceptor – invoke Method</b> 주 업무 시작과 마지막 부분에 처리해야 할 필요가 있는 부가 작업을 처리 <code>public Object invoke(Object proxy, Method method, Object[] args)</code>

## 9 PointCut, Join Point (포인트 컷 & 조인포인트)

- **위빙 (Weaving)** : 보조업무가 프록시(Proxy)를 통해서 주 업무에 주입되는 것.  
즉, 타깃 객체에 애스펙트(Aspect)를 적용해서 새로운 프록시 객체를 생성하는 절차.
- **조인포인트 (Joint Point)** : 위빙하게 되는 함수. 그 지점.
- **포인트 컷 (PointCuts)** : 객체의 특정 함수만 조인포인트 역할을 하도록 하는 것.
- **추가 설명** : 스프링 AOP는 특정 객체(클래스)를 대상으로 타깃(Target)을 설정하였는데, 그렇기 때문에 프록시 적용이 해당 객체 단위 전체에 적용이 됨.  
즉 타깃에 속한 모든 조인포인트에 보조 업무(Proxy Class)가 적용됨. 하지만, 포인트 컷을 활용하면 특정 클래스의 특정 함수에만 보조 업무를 삽입할 수 있음.
- 예제

```
<bean id="cal" class="aoptest.TestCalculator"></bean>
<bean id="logPrintAroundAdvice" class="aoptest.LogPrintAroundAdvice"/>
<bean id="logPrintBeforeAdvice" class="aoptest.LogPrintBeforeAdvice"/>
<bean id="logPrintAfterReturningAdvice" class="aoptest.LogPrintAfterReturningAdvice"/>
<bean id="logPrintAfterThrowingAdvice" class="aoptest.LogPrintAfterThrowingAdvice"/>

<bean id="namePointcut" class="org.springframework.aop.support.NameMatchMethodPointcut">
<property name="mappedNames">
  <list>
    <value>add</value>
    <value>subtract</value>
  </list>
</property>
</bean>

<bean id="nameAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
  <property name="pointcut" ref="namePointcut"></property>
  <property name="advice" ref="logPrintAroundAdvice"></property>
</bean>

<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="cal"></property>
  <property name="interceptorNames">
    <list>
      <value>nameAdvisor</value>
      /* <value>logPrintAroundAdvice</value>
      <value>logPrintBeforeAdvice</value>
      <value>logPrintAfterReturningAdvice</value>
      <value>logPrintAfterThrowingAdvice</value> */
    </list>
  </property>
</bean>
```

```
</bean>
```

#### [코드 설명]

1. 특정 함수(add, subtract) 만 조인포인트 함수가 되도록 포인트 컷 지정.
2. 특정 포인트 컷에 특정 어드바이스(핸들러) 지정.

어드바이저 = 포인트 컷 + 어드바이스

3. 특정 어드바이스(Advice) 대신 어드바이저(Advisor) 지정 가능
    - **TestCalculator** : 주 업무(타깃) 클래스 (AOP처리가 필요한 클래스)
    - **LogXXXAdvice** : 어드바이스 클래스 (핸들러 클래스 – 주 업무 처리 이전, 이후 등 부가적인 업무를 처리할 클래스)
    - **PointCut(mappedNames)** : 주 클래스의 함수 중에 내가 어드바이스를 적용을 한정(포인트 컷 할)시킬 함수를 지정 (특정 함수에만 국한지어 적용)
    - **Advisor(nameAdvisor)** : 포인트컷과 어드바이스를 합친 어드바이저를 생성. 어드바이스 대신 어드바이저 사용 가능.
    - **Proxy** : 타깃 클래스를 지정하고, 어드바이저(또는 어드바이스)를 적용할 것(interceptorNames)을 지정한 후, 대리 클래스 생성. ProxyFactoryBean을 통한 동적 프록시 생성.
- 참고로 어드바이저에는 여러 개의 어드바이스와 포인트 컷이 추가 될 수 있기 때문에 개별적으로 등록 시 어떤 어드바이스(부가 기능)에 대해 어떤 포인트컷(메소드 선정)을 적용 할지 애매해진다. 그래서 어드바이스와 포인트 컷을 묶은 오브젝트를 어드바이저라고 부른다.

어드바이저 = 포인트컷(메소드 선정 알고리즘) + 어드바이스(부가 기능, 애스펙트의 한가지 형태)

```
<bean id="nameAdvisor" class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
    <property name="pointcut" ref="namePointcut"></property>
    <property name="mappedNames">
        <list>
            <value>add</value>
            <value>subtract</value>
        </list>
    </property>
    <property name="advice" ref="logPrintAroundAdvice"></property>
</bean>
```

// JAVA 정규식 표현을 활용한 포인트 컷 패턴 설정.

```
<bean id="nameAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="pointcut" ref="namePointcut"></property>
    <property name="patterns">
        <list>
            <value>.*a.*</value>
            <value>.*b</value>
        </list>
    </property>
    <property name="advice" ref="logPrintAroundAdvice"></property>
</bean>
```

- mappedName : 적용시킬 함수를 지정.
- mappedNames : 적용시킬 함수들을 지정.
- 예제 클래스에는 2개의 함수(add, subtract) 밖에 없지만, 함수가 상당히 많은 경우, 패턴을 사용하는 것이 더 효율적일 수 있다.  
그룹핑을 통해 어떤 어드바이스(Advice)만 실행되게 할 것인지도 한정할 수도 있다.

#### [정규식 특수 문자 의미]

- 1.. : 어떠한 문자든 상관없다. (한개의 문자가 옴)
- 2.\* : 임의의 문자가 0개 이상 올 수 있다. (한개 이상의 문자가 옴)

#### • 최종 정리

- **위빙** : 크로스 컷팅 되어 있는 관심사(Concern)를 다시 결합하는 것.
- **조인포인트(Join Point)** : 위빙하는 함수들이 있는 곳 지점, 시점 (스프링에서는 메소드만을 조인 포인트로 사용함.)
- **포인트 컷(PointCut)** : 부가 기능 적용 대상 메서드 선정 방법. 내가 원하는 메서드만 골라서 조인포인트 역할을 하도록 함.  
(단, 포인트 컷을 통해 특정 타깃 함수와 어드바이스를 매 하나씩 연결해야 한다면 설정 파일의 용량도 커지고, 함수가 많아지게 되면 관리도 어려워질 수 있음.)

#### • 스프링에서의 조인 포인트 특징

1. 분리하고 도킹하는 것이 자유로움.
2. 조인포인트를 쓸 수 있는 것이 **Method**에 한정.

## 10 빈 후처리기 (Bean Post Processor)

- **빈 후처리기란 (Bean Post Processor) ?**

- 빈(Bean)의 설정을 후처리(postprocessing)함으로써 빈의 생명 주기와 빈 팩토리의 생명주기에 관여.
- 빈의 초기화 되기 전, 초기화 된 후 2개의 기회를 제공.
- 빈 프로퍼티의 유효성 검사 등에 사용.
- 다른 초기화 메소드인 afterPropertiesSet()과 init-method가 호출되기 직전과 직후에 호출되어짐.
- 예제

```
public interface BeanPostProcessor {
    Object postProcessBeforeInitialization(Object bean, String name) throws BeansException;
    Object postProcessAfterInitialization(Object bean, String name) throws BeansException;
    ...
}

package com.sample;
public class Foo implements BeanPostProcessor {
    public void method1() {
        // 어떠한 작업
    }
    Object postProcessBeforeInitialization(Object bean, String name) throws BeansException {
        // 초기화 되기 전 처리
    }
    Object postProcessAfterInitialization(Object bean, String name) throws BeansException;
        // 초기화 된 후 처리
    }

Resource resource = new FileSystemResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
BeanPostProcess foo = new Foo();
Factory.addBeanPostProcessor(foo);

// beans.xml 개체
<bean id="foo" class="com.sample.Foo" />

Bean Factory를 IoC Container 로 사용하는 경우에는 단순히 addBeanPostProcessor() 메서드를 이용하여 프로그램 방식으로만
빈 후처리기(Bean Post Processor)를 등록함.
```

- **빈 후처리 과정(Process of Bean Post Processing)**

- 빈 객체(Bean Instance) 생성 ( 생성자(Constructor) 또는 팩토리 메서드(Factory Method) 사용 )
  - 빈 프로퍼티(Bean Property)에 값과 빈 래퍼런스 설정.
  - Aware Interface에 정의된 Setter 메서드 호출
  - 빈 인스턴스(Bean Instance)를 각 빈 후처리기(Bean Post Processor)의 postProcessBeforeInitialization() 에 전달.
  - 초기화 Callback 메서드 호출
  - 빈 인스턴스(Bean Instance)를 각 빈 후처리기(Bean Post Processor)의 postProcessAfterInitialization() 에 전달.
  - 빈 사용 준비 완료
  - 컨테이너 종료 후, 소멸(Destructor) Callback 메서드 호출
- [BeanFactory 인터페이스]
    - 빈 객체를 관리하고 각 빈 객체 간의 의존 관계를 설정해주는 기능 제공.
    - 가장 단순한 컨테이너.
  - [XmlBeanFactory 클래스]
    - 외부 자원으로부터 설정 정보를 읽어 와 빈 객체를 생성하는 클래스
  - 추가 설명

스프링의 빈 객체는 기본적으로 싱글톤(Singleton)으로 생성되는데, 그 이유는 사용자의 요청이 있을 때마다 애플리케이션 로직까지 모두 포함하고 있는 오브젝트를 매번 생성하는 것은 비효율 적이기 때문임.  
 하나의 빈 객체(Bean Object)에 동시에 여러 스레드가 접근할 수 있기 때문에 상태 값을 인스턴스 변수에 저장해 두고 사용할 수 없음.  
 따라서 싱글톤의 필드에는 의존 관계에 있는 빈에 대한 참조(Reference)나 읽기 전용(Read-Only) 값만을 저장해 두고,  
 실제 오브젝트의 변화 상태를 저장하는 인스턴스 변수는 두지 않음. 애플리케이션 로직을 포함하고 있는 오브젝트는 대부분 싱글톤 빈(Singleton Bean)으로 만들면 충분하다.

하지만, 하나의 빈 설정으로 여러 개의 오브젝트를 만들어서 사용하는 경우도 발생하게 되는데, 이 때는 싱글톤이 아닌 빈을 생성해야 함. 싱글톤이 아닌 빈은 프로토타입 빈(Prototype Bean)과 스코프 빈(Scope Bean)이 있다.

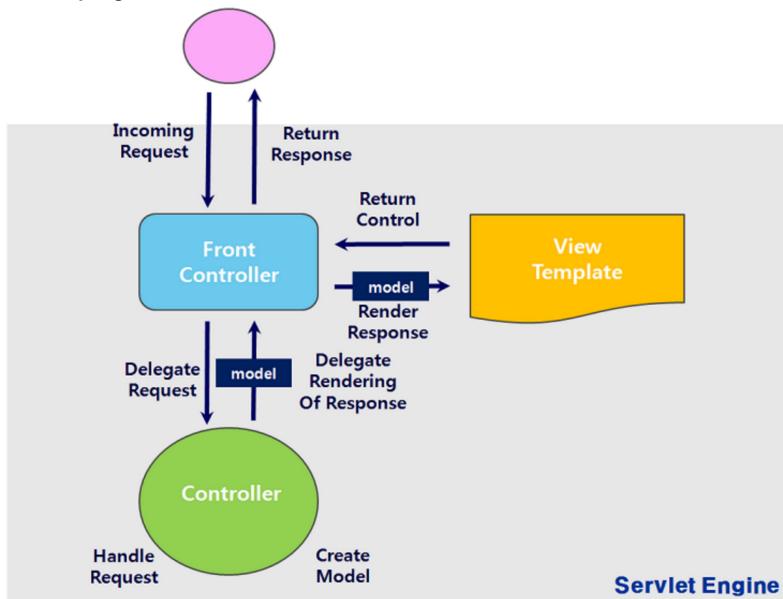
## 11 Spring MVC (Model2)

- Front Controller : URL매핑을 담당.  
 (어떠한 컨트롤러가 어떤 URL을 쓸것인가에 대한 설정 파일 포함) - 설정 파일을 통해서 적절한 컨트롤러를 호출.

(Front Part : Servlet (DispatcherServlet.class))

- Controller : 화면 단에서 보여줄 데이터들을 미리 만들어 놓는 것. 그 데이터는 모델(Model)이라 함.
- Model : 데이터(Data)
  - > Model, Controller, Front Controller, View 를 잘 구조화하는 것이 MVC 모델을 제어하고 있는 라이브러리가 할 역할.

- 스프링(Spring) MVC의 개념도



- 예제 코드

```
// web.xml
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>*.oz</url-pattern>
</servlet-mapping>

// dispatcher-servlet.xml // 어떤 URL을 어떤 클래스에 매핑할 것인가?
<bean name="customer/notice.oz" class="controllers.customer.NoticeController"></bean>

// NoticeController.java
package controllers.customer;
public class NoticeController implements Controller { // 스프링이 제공 (Controller)
    @Override
    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        ModelAndView mv = new ModelAndView("notice.jsp");
        mv.addObject("test", "hello");
        return mv;
    }
}

// notice.jsp
<div id="main"><h2>테스트 값 : ${test}</h2></div>

// 실행 및 결과
URL Call : localhost/customer/notice.oz => 화면 출력 : hello
```

- 예제 코드 설명

예제 코드는 게시판 MVC (스프링 MVC – 서블릿 DispatcherServlet)로 공지사항 리스트를 관리하고, 특정 번호의 공지사항을 열람하기 위한 공지사항 페이지를 구현한 것임. (스프링의 MVC의 개념을 설명하는데에 있어서 게시물 등록, 수정, 삭제 처리 험수는 크게 다른게 없으므로 생략.)

1. **ModelAndView** : Controller의 처리결과를 보여줄 View와 View에서 사용할 값을 전달하는 클래스.
2. ModelAndView의 mv 오브젝트는 web.xml에서 설정했던 DispatcherServlet이 받아서 처리함.
3. \${test} : 표현 언어 (JAVA 레벨(서버))의 변수를 Front(클라이언트)에 출력하기 위한 표현 언어

#### [WEB.XML 이란?]

- Web Application 의 환경 파일(Deployment Description)
- XML 형식(XML Schema)의 파일로써, WEB-INF 폴더에 위치
- <web-app> 태그로 시작하고 종료하는 문서로써 web.xml 이 정의 된 Web Application의 동작과 관련된 다양한 환경 정보를 태그 기반으로 설정하는 파일.
- Servlet 2.3 까지 DTD 파일, 2.4 부터 XML Schema 파일 형태로 변경.

#### • WEB.XML 파일의 구성 내용

- a. ServletContext의 초기 파라미터
- b. Session의 유효시간 설정
- c. Servlet/JSP에 대한 정의
- d. Servlet/JSP 매핑
- e. Mime Type 매핑
- f. Welcome File list
- g. Error Pages 처리
- h. Listen/Filter 설정
- i. 보안

## 12 Spring MVC (AOP with XML)

#### • 예제

```
// NoticeController.java // 공지사항 리스트
package controllers.customer;

public class NoticeController implements Controller { // 스프링이 제공 (Controller)

    private NoticeDao noticeDao;

    public void setNoticeDao(NoticeDao noticeDao) { // setter
        this.noticeDao = noticeDao;
    }

    @Override
    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        String _page = request.getParameter("pg");
        String _field = request.getParameter("f");
        String _query = request.getParameter("q");

        int page = 1;
        String field = "TITLE";
        String query = "%"%

        if(_page == null && _page.equals(""))
            page = Integer.parseInt(_page);
        if(_field == null && _field.equals(""))
            field = _field;
        if(_query == null && _query.equals(""))
            query = _query;

        List<Notice> list = noticeDao.getNotices(page, field, query); // 페이지번호, 검색항목, 검색어
        ModelAndView mv = new ModelAndView("notice.jsp");
        mv.addObject("listData", list);

        return mv;
    }
}
```

실행 호출 : notice.tst?pg={param.pg}&f={param.f}&q={param.q}

실행 결과 : 공지사항 리스트 정보 (listData)

#### [코드 설명]

- NoticeDao 클래스 인스턴스를 별도로 생성. (기존 예제에서는 handleRequest 함수 이내에서 처리함)
- Setter 함수도 생성.
- 이전의 고전적인 JSP 방식에서는 DB처리를 위한 noticeDao 클래스를 만들어 그 인스턴스 객체를 사용하여 작업하였지만, 여전히, 객체를 생성, 호출하는 부분이 분리되지 않음. 매번 만들어야 함. 또한 함수 내에 있는 NoticeDao는 빈(Bean) 객체로 생성할 수도 없고, XML에서 접근 불가.
- XML을 활용하면 noticeDao의 생성과 초기화, 디펜던시 등을 분리하여 관리할 수 있음. DB등의 정보가 변경되어도 XML파일만 수정.

```
// NoticeDetailController.java // 공지사항 상세 내용
package controllers.customer;
```

```

public class NoticeDetailController implements Controller { // 스프링이 제공 (Controller)

    private NoticeDao noticeDao;

    public void setNoticeDao(NoticeDao noticeDao) { // setter
        this.noticeDao = noticeDao;
    }

    @Override
    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        String seq = request.getParameter("seq");
        Notice notice = noticeDao.getNotice(seq);

        ModelAndView mv = new ModelAndView("noticeDetail.jsp");
        mv.addObject("noticeData", notice);

        return mv;
    }
}

```

실행 호출 : noticeDetail.tst?pg={param.pg}&f={param.f}&q={param.q}  
 실행 결과 : noticeData 객체를 페이지에 전달하여 적절한 내용을 출력함.

```

// dispatcher-servlet.xml ( 객체를 생성해주고, 주입(Injection)함 )
<bean name="noticeDao" class="dao.NoticeDao">

// 빈 객체 생성하고 값(레퍼런스)를 주입.
// setNoticeDao() 호출을 통한 값(레퍼런스) 배정 (value or ref )
<bean name="customer/notice.tst" class="controllers.customer.NoticeController">
    <property name="noticeDao" ref="noticeDao"></property>
</bean>

<bean name="customer/noticeDetail.tst" class="controllers.customer.NoticeDetailController">
    <property name="noticeDao" ref="noticeDao"></property>
</bean>

```

XML에서 noticeDao 객체를 각 빈(Bean) noticeController와 NoticeDetailController에 생성하고, 주입.

```

// noticeDetail.jsp // 공지사항 상세 정보 표시 페이지
<div id=notice-article-detail>
    <dl class="article-detail-row">
        <dt class="article-detail-title">제목</dt>
        <dd class="article-detail-data">
            ${notice.title}
        </dd>
    </dl>
    <dl class="article-detail-row">
        <dt class="article-detail-title">작성일</dt>
        <dd class="article-detail-data">
            ${notice.regDate}
        </dd>
    </dl>
    ...
</div>

```

#### • 이러한 AOP 구현에 있어서 XML 방식의 문제점

- XML에 URL 수만큼 Bean객체를 생성하여 객체 값을 설정 해야하는 어려움이 있음.
- 문제가 되는 이유는 Controller 클래스에는 URL에 매핑되는 오버라이드(Override)된 handleRequest 함수가 하나씩만 배정되어 있음.
- URL에 반응하는 Controller가 함수 하나 하나마다 매번 캡슐(java 페이지) 생성해야 하므로, 코드 작성의 어려움이 있음. 즉 소스 페이지(JSP, XML, JAVA 등)가 여러 곳에 산재하게 됨.  
--> 웹 서비스 규모에 따른 너무 많은 개체 수 관리.

#### • 해결 방법

- 만약 URL을 처리할 수 있는 험수가 하나의 캡슐(Controller)에 여러 개의 매핑 메서드(Method)로 담을 수 있다면?

b. 즉 클래스(컨트롤러) 하나에 여러개의 URL-메서드 매핑(Mapping)이 가능하다면??

--> **애노테이션 (Annotation)**

## 13 AOP with Annotation

### • 애노테이션(Annotation) & 오토와이어링(Autowiring)

- 하나의 컨트롤러(Controller)와 하나의 페이지를 매핑하는 것은 작업도 번거롭고, 관리도 힘듬.

그렇다면 하나의 함수에 하나의 URL을 어떻게 매핑할 것인가?

--> 애노테이션(Annotation)을 사용

#### - 애노테이션은 무엇인가?

- o 컴파일러를 거친 후에도 코드에 남아있는 특수한 주석!!
- o 애노테이션은 특수한 주석으로 자체적으로 어떠한 기능을 수행하는 것은 아니지만, 애노테이션이 가리키는 자시어(정보)를 기반으로 컴파일러는 애노테이션이 적용된 클래스 또는 매서드, 프로퍼티 등에 적절한 기능을 부여한다.
- o 애노테이션은 @(골뱅이)로 주석을 시작하며, 클래스, 매서드, 변수 등에 적용됨.

### • 예제

```
// dispatcher-servlet.xml (객체를 생성해주고, Injection해줌)
<context:component-scan base-package="controller"></context:component-scan>
<bean name="noticeDao" class="dao.NoticeDao"> // Autowired 어노테이션을 통해서 자동 매핑됨.

// CustomerController.java
package controllers;

@Controller
@RequestMapping("/customer/*")
public class CustomerController {

    private NoticeDao noticeDao;

    @Autowired
    public void setNoticeDao(NoticeDao noticeDao) {
        this.noticeDao = noticeDao;
    }

    @RequestMapping("notice.oz")
    public String notices(String pg, String f, String q, Model model) throws ClassNotFoundException {
        int page = 1;
        String field = "TITLE";
        String query = "%";

        if(pg == null && pg.equals(""))
            page = Integer.parseInt(pg);
        if(f == null && f.equals(""))
            field = _field;
        if(q == null && q.equals(""))
            query = _query;

        List<Notice> list = noticeDao.getNotices(page, field, query); // 페이지번호, 검색항목, 검색어
        mv.addObject("listData", list);

        return "notice.jsp"; // VIEW로 사용되는 페이지 문자열(String)
    }

    @RequestMapping("noticeDetail.oz") // 게시물 상세 내용 보기
    public String noticeDetail(String seq, Model model) throws ClassNotFoundException {
        Notice notice = noticeDao.getNotice(seq);
        model.addAttribute("notice", notice);
        return "noticeDetail.jsp";
    }

    @RequestMapping("noticeReg.oz") // 게시물 등록
    public String noticeReg(Notice n /* String title, String content */) throws ClassNotFoundException {

        /* Notice n = new Notice();
        n.setTitle(title);
        n.setContent(content); */
        noticeDao.insert(n);
        return "redirect:notice.oz";
    }
}
```

```

@RequestMapping("noticeEdit.oz") // 게시물 수정
public String noticeEdit(Notice n /* String title, String content */) throws ClassNotFoundException {

    /* Notice n = new Notice();
    n.setTitle(title);
    n.setContent(content); */
    noticeDao.update(n);
    return "redirect:noticeDetail.oz?seq=" + n.getSeq();
}

```

#### [코드 설명]

- **Context:component-scan** : Controller 패키지에서 URL-메소드 매핑된 정보를 자동으로 검색(Auto-Scan)함.  
단 Controller 애노테이션된 클래스에서 해당 메서드를 찾음. 위에서 /customer/notice.oz 서브 URL이 호출되면 notices 메서드가 수행됨.  
(기존 XML 방식에서는 특정 페이지의 URL과 컨트롤러를 하나씩 배정했지만, context:component-scan을 통해서는 사용자로부터 URL이 입력된 경우 @Controller 애노테이션을 가지고 있는 클래스 내부의 RequestMapping 정보를 자동으로 스캔하여 처리 함수를 찾아 연결하여 줌.)
- **@RequestMapping** – 표시된 URL에 해당 메서드를 매핑 (지정된 URL요청이 들어오면 해당 메서드를 실행함)
  - XML을 사용하지 않으므로 URL-Controller 맵핑 정보(빈(Bean) 객체) 자체가 XML 파일에서 사라지게 되는데 이렇게 되면 빈(Bean) 객체를 생성하고, 프로퍼티(AOP관련 포함) 값을 세팅해줄 수가 없음.
    - 이 문제는 @Autowired 애노테이션을 통해 해결 가능
    - component-scan을 사용하는 경우 빈(Bean) 객체를 세팅하고 값을 할당해 줄 수 없지만, @Autowired를 사용하여 자동으로 객체와 변수를 연결 및 설정해 줌.
- 메서드에서 반환하는 모델이 있다면 메서드 내에 파라미터 Model model을 선언만 해주면 됨.  
해당 Model 객체가 생성되어 참조가 자동으로 됨.

#### [동일한 페이지를 다르게 처리]

```

@RequestMapping(value="/customer/noticeReg.oz", method=RequestMethod.GET)
@RequestMapping(value="/customer/noticeReg.oz", method=RequestMethod.POST)

```

#### [매핑된 컨트롤러의 함수가 처리하는 JSP페이지와는 다른 페이지로 결과를 넘길때]

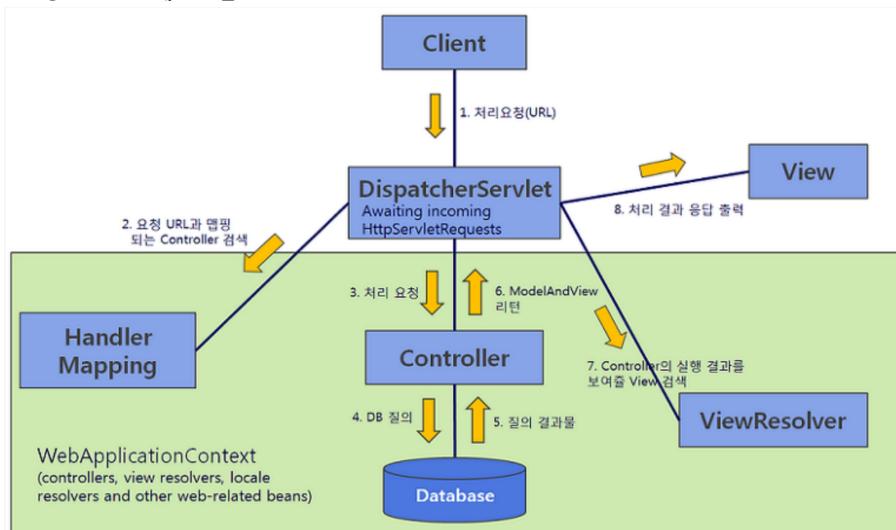
```
return "redirect:notice.oz" // redirect 처리
```

- Request로부터 넘어오는 인자는 1:1로 함수 매개변수로 대입 가능.

더 나아가 Request를 통해서 넘어오는 개별 어트리뷰트(Attribute) 인자의 이름(변수명)이 처리 함수의 매개 변수의 레퍼런스 클래스의 인스턴스(속성 값)의 이름(변수 명)과 동일하다면 이를 자동으로 맵핑(대입)하여 줌.

## 14 Spring MVC Process

#### • 스프링 MVC 프로세스 모델



#### • 스프링 MVC의 주요 구성 요소

- DispatcherServlet** : 클라이언트의 요청을 전달 받는 역할. 컨트롤러(Controller)에게 클라이언트의 요청을 전달하고, 컨트롤러가 리턴한 결과 값을 뷰(View)에 전달하여 알맞은 응답을 생성하도록 함. (스프링 제공)
- HandlerMapping** : 클라이언트의 요청 URL을 어떤 컨트롤러가 처리할지를 결정. (스프링 제공)
- Controller** : 클라이언트의 요청을 처리한 뒤, 그 결과를 DispatcherServlet에 알려준다. (실제 로직을 담당)
- SQL 쿼리를 통한 데이터 질의문 전달.**

- e. 데이터베이스가 반환(Return)한 데이터 획득.
- f. **ModelAndView** : 컨트롤러가 처리한 결과 정보 및 뷰 선택에 필요한 정보를 담음.
- g. **ViewResolver** : 컨트롤러의 처리 결과를 생성할 뷰를 결정. (스프링 제공)
- h. **View** : 컨트롤러의 처리 결과 화면을 생성 (Freemarker 등)

- **스프링 MVC 구조 및 처리 순서**

1. 클라이언트의 요청이 DispatcherServlet에 전달됨.
2. DispatcherServlet은 HandleMapping을 사용하여 클라이언트의 요청을 처리할 컨트롤러(Controller)를 검색.
3. 매치되는 컨트롤러가 있다면 요청을 해당 컨트롤러에게 전달.
  - DispatcherServlet은 컨트롤러 객체의 handleRequest() 메서드를 호출하여 클라이언트 요청을 처리 .
4. DB 관련 처리 (DB 질의 요청)
5. DB 관련 처리 (DB 질의 결과)
6. 컨트롤러의 handlerRequest()메서드는 처리 결과 정보를 담은 ModelAndView 객체를 리턴.
7. DispatcherServlet은 ViewResolver로부터 응답 결과를 생성할 뷰 객체를 구함.
  - JSP, Velocity, Freemarker 등 다양한 뷰를 지원.
    - AbstractCachingViewResolver
    - XmlViewResolver
    - ResourceBundleViewResolver
    - UrlBasedViewResolver
    - InternalResourceViewResolver
    - FreeMarkerViewResolver
    - ViewResolvers Chaining
8. 뷰는 클라이언트에 전송할 응답을 생성.

## 15 References

- 스프링 4 프로그래밍
- 토비의 스프링 3.1
- 위키피디아
- 스프링 애플리케이션 개발