스프링(Spring) 프레임워크 기본 개념

1. 스프링의 이해

1.1) 스프링 정의

- 간단히 스프링이라 지칭하지만 정확하게는 스프링 프레임워크 (Spring Framework) 라고 하는 것이 정확한 표현.'

*** 자바(JAVA) 플랫폼을 위한 오픈 소스(Open Source) 애플리케이션 프레임워크(Framework)**

*** 자바 엔터프라이즈 개발을 편하게 해주는 오픈 소스 경량급 애플리케이션 프레임워크**

*** 자바 개발을 위한 프레임워크로 종속 객체를 생성해 주고, 조립해 주는 도구**

*** 자바로 된 프레임워크로 자바SE로 된 자바 객체(POJO)를 자바EE에 의존적이지 않게 연결해 주는 역할.**

- POJO 란 Plain Old Java Object 의 약자로 특별한 뜻을 담고 있는 용어는 아니며, 단순히 평범한 자바빈즈(Javabeans) 객체를 의미함.

1.2) 스프링 특징

*** 크기와 부하의 측면에서 경량.**

*** 제어 역행(IoC)이라는 기술을 통해 애플리케이션의 느슨한 결합을 도모.**

*** 관점지향(AOP) 프로그래밍을 위한 풍부한 지원을 함.**

*** 애플리케이션 객체의 생명 주기와 설정을 포함하고 관리한다는 점에서 일종의 컨테이너(Container)라고 할 수 있음.**

*** 간단한 컴포넌트로 복잡한 애플리케이션을 구성하고 설정할 수 있음.**

- 스프링의 특징을 좀 더 상세히 말하자면 -

1) 경량 컨테이너로서 자바 객체를 직접 관리.

각각의 객체 생성, 소멸과 같은 라이프 사이클을 관리하며 스프링으로부터 필요한 객체를 얻어올 수 있다.

2) 스프링은 POJO(Plain Old Java Object) 방식의 프레임워크.

일반적인 J2EE 프레임워크에 비해 구현을 위해 특정한 인터페이스를 구현하거나 상속을 받을 필요가 없어 기존에 존재하는 라이브러리 등을 지원하기에 용이하고 객체가 가볍다.

3) 스프링은 제어 반전(IoC : Inversion of Control)을 지원.

컨트롤의 제어권이 사용자가 아니라 프레임워크에 있어서 필요에 따라 스프링에서 사용자의 코드를 호출한다.

4) 스프링은 의존성 주입(DI : Dependency Injection)을 지원

각각의 계층이나 서비스들 간에 의존성이 존재할 경우 프레임워크가 서로 연결시켜 준다.

5) 스프링은 관점 지향 프로그래밍(AOP : Aspect-Oriented Programming)을 지원

따라서 트랜잭션이나 로깅, 보안과 같이 여러 모듈에서 공통적으로 사용하는 기능의 경우 해당 기능을 분리하여 관리할 수 있다.

6) 스프링은 영속성과 관련된 다양한 서비스를 지원

iBatis나 Hibernate 등 이미 완성도가 높은 데이터베이스 처리 라이브러리와 연결할 수 있는 인터페이스를 제공한다.

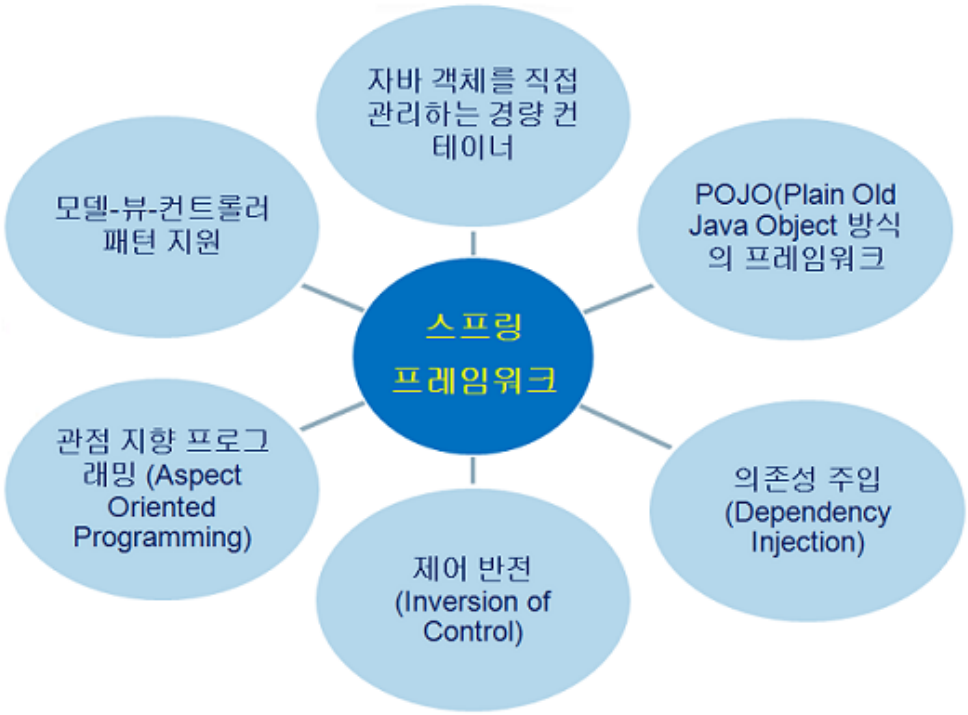
7) 스프링은 확장성이 높음.

스프링 프레임워크에 통합하기 위해 간단하게 기존 라이브러리를 감싸는 정도로 스프링에서 사용이 가능하기 때문에 수많은 라이브러리가 이미 스프링에서 지원되고 있고 스프링에서 사용되는 라이브러리를 별도로 분리하기도 용이하다.

- 또 다른 특징 -

*** 동적(Dynamic) 웹 사이트 개발을 위한 프레임워크**

*** 대한민국 전자정부 표준 프레임워크의 기반 기술**



1.3) 스프링 역사

*** Rod Johnson 이 2002년 출판한 자신의 저서 Expert One-on-One J2EE Design and Development에 선보인 코드가 시초.**

*** 이 프레임워크는 2003년 6월 최초 아파치 2.0 라이선스로 공개.**

*** 2004년 3월에 1.0 버전이 릴리즈 되고, 2015년 8월에 현재 최신 버전인 4.2 버전이 공개됨**

(2) 주요 구성 요소 & DI

1. 스프링의 핵심 개념

*** DI**

*** IoC**

*** AOP & AOP Proxy**

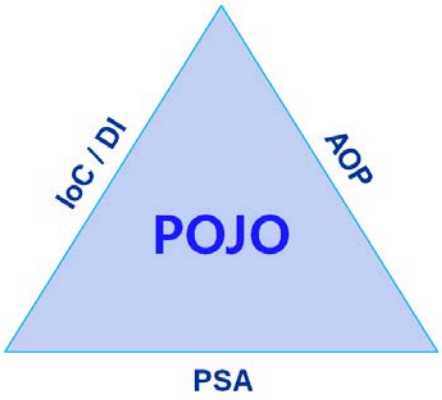
*** AOP in Spring**

1) 주요 구성 요소

*** IoC / DI**

*** AOP**

*** PSA**



- 용어 설명 -

*** Plain Old Java Object 혹은 POJO**는 처음에 javax.ejb 인터페이스를 상속받지 않은, 무거운 EJB와는 반대로 경량의 자바 객체를 지칭하는 용어로 소개. 즉, POJO라는 이름은 특별하지도 않고, 특히 Enterprise JavaBean이 아닌 자바 객체를 강조하는 의미로 사용.

이 이름은 Martin Fowler, Rebecca Parsons와 Josh MacKenzie에 의해 2000년 9월에 만들어졌으며, 프레임워크에 종속된 복잡한 객체에 반대되는 간단한 객체를 설명하기 위한 용어의 필요성이 많아짐에 따라 POJO라는 이름이 점점 널리 쓰여짐.

POJO 대표적인 예로, JavaBean을 들 수 있는데, JavaBean은 기본 생성자와 멤버 필드에 접근할 수 있는 getter/setter 메소드를 가진 serializable(직렬화가 가능한)한 객체를 의미. POJO를 이용한 디자인이 널리 쓰임에 따라 POJO를 기본으로 하는 스프링이나 하이버네이트와 같은 프레임워크에서도 생겨남. 요즘에는 POJO는 (EJB 뿐만 아니라) 별도로 종속되지 않는 자바 객체를 통칭하여 의미한다.

*** PSA (Portable Service Abstractions) – (쉬운) 서비스 추상화**

성격이 비슷한 여러 종류의 기술을 추상화하고 이를 일관된 방법으로 사용할 수 있도록 지원.

트랜잭션 서비스 추상화 : 여러 가지의 DB를 사용한다고 하면 Global Transaction 방식을 사용.
자바는 JDBC 외에 이런 글로벌 트랜잭션을 지원하는 트랜잭션 매니져를 지원하기 위한 API인 JTA(Java Transaction Api)를 제공.
높은 응집도와 낮은 결합도를 준수.

2) DI (Dependency Injection, 의존성 주입)

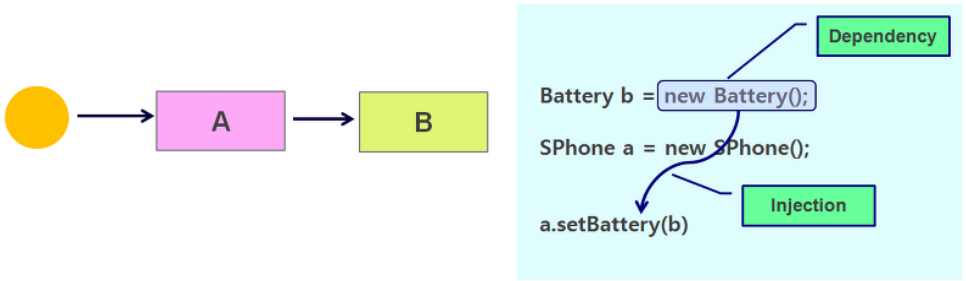
DI는 스프링(Sping)을 통해서 특별히 생겨난 용어는 아님. 단 DI를 잘 지원해 주는 게 스프링.

*** 일체형**

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

*** 분리 / 도킹(부착) 형**

- Association 관계
- A 객체가 다른 녀석이 만든 B 객체를 사용.



예제 설명

A를 스마트폰, B를 배터리리라 하면,
일체형 스마트폰 (아이폰)은 바로 전원을 켜도 되지만,
배터리 탈부착 형태의 스마트폰은 여기서는 배터리를 넣고, 전원을 넣어야 함.

일체형은 A라는 객체의 내부 프로세스에 대해 신경 쓸 필요가 없으며, 분리형은 A와 B를 개별적으로 세팅해 주어야 함. 단, 분리형은 내가 원하는 것(다른 배터리)으로 바꾸어 부착할 수 있음. 이것을 DI의 개념이라 보면 됨.

위의 예제 처럼 분리/도킹(부착) 형으로 개발을 하게 되면 각 객체(또는 애플리케이션) 간의 결합도를 낮출 수 있으며, DI를 사용하는 목적이 이러한 결합도를 낮추기 위함.

2-1) DI의 종류

*** Setter Injection (세터 주입)**
*** Construction Injection (생성자 주입)**

Setter Injection

```
B b = new B();
A a = new A();
a.setB(b);
```

Construnction Injection

```
B b = new B();
A a = new A(b);
```

위의 코드는 JAVA 개발자라면 친숙한 개념으로 DI는 JAVA에서도 많이 사용해 왔던 개념.
(객체 지향 프로그래밍(OOP)에 있던 개념)
단지, 스프링에서는 이러한 일련의 과정을 (동적으로) 자동화 함.

2-2) 스프링에서의 DI

*** 명세서에 따라서 자동적으로 부품을 활용하여 제품을 조립 => 스프링**

*** 일체형 프로그램과는 반대로 제품을 생성.**
*** 즉, 작은 부품부터 시작하여 큰 부품으로 이동하며 조립.**
=> Inversion of Control (IoC)

기본적인 완제품 제작 순서와는 다르게 작은 부품부터 큰 부품으로, 제품을 만드는 순서가 역순 (Inversion of Control).
그래서 IoC라는 이름이 붙었으며, 이러한 일련의 작업을 스프링은 컨테이너라는 곳에 담아서 처리하여 스프링을 IoC컨테이너라 함.
스프링은 완성품을 만드는 것보다 부품을 모아서 조립하는 것을 도와줌. 즉, 이러한 DI 구조를 개발자가 좀 더 쉽고, 간편하게 사용할 수 있도록 지원해 주는 프레임워크 중의 하나가 스프링.

스프링에서의 DI의 의미
- 부품들을 생성하고, 제품을 조립해주는 공정과정을 대신해 주는 라이브러리 (역할자)

* 개발 핵심 처리 루틴의 수정 없이 제품(객체)를 다른 제품(객체)로 쉽게 대체하여 생성 가능하도록 하는 역할을 함.
* 명세서에 따라서 자동적으로 부품을 활용하여 제품을 조립.
* 생성하기 원하는 객체를 명세서(XML)에 기술하고, 그 부품과 의존성(Dependency)들을 보관하는 일을 처리. 그러한 데이터를 보관하는 공간을 컨테이너라 함. (IoC 컨테이너)

제품	스프링
주문서	설정파일(XML)
일반적인 순서의 공정	역방향 조립

2-3) DI 구현

객체의 생성과 도킹에 대한 내용이 소스 코드 상에 있는 것이 아닌 별도의 텍스트 파일(XML 설정 파일)에 분리하여 존재.
(JAVA 소스 컴파일 없이 XML 변경만으로 내용 변경 가능)

JAVA(DI) : Property 항목은 실제 값을 record에 바로 주입하는 것이 아닌 setRecord() 함수를 호출하여 주입함.

• JAVA (DI)

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

XML (스프링 DI) : 객체 생성 시, 패키지명을 포함한 풀 클래스 네임 작성.
XML에 작성된 명세서를 보고, IoC컨테이너가 각 객체를 생성하고, 값을 주입해줌.
여기서 ApplicationContext 가 IoC컨테이너 역할을 함.

• XML (스프링 DI) ***config.xml***

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

• **JAVA**

```
// XML을 파싱하여 컨테이너에 담는 작업
ApplicationContext ctx = new ClassPathXmlApplicationContext("config.xml");
RecordView = (RecordView) ctx.getBean("view");
```

* XML을 활용(스프링 DI)하는 경우는 VIEW에 대한 객체만을 요청했을 뿐, 실제 내부적인 사항은 JAVA코드 상에 드러나지 않음.
* 새로운 클래스의 bean객체를 만들어 XML에 주입만 시켜줘도, 기존 소스 변경 없이 새로운 형태의 객체 적용이 가능함.

2-4) XML (Bean) Sample

* 빈(Bean) 객체는 반드시 클래스를 사용. 인터페이스나 추상클래스는 객체 생성이 불가능함.
* 빈 객체 생성, 객체 초기화, 객체 값(또는 레퍼런스) 주입.

XML (스프링 DI) config.xml

```
 1)
<bean id="record" name="r1,r2 r3;r4" class="di.SprRecord">
    <property name="kor" value="20"></property>
</bean>
 2)
 <bean id="record" name="r1,r2 r3;r4" class="di.SprRecord">
    <constructor-arg value="20"></constructor-arg>
</bean>
 3)
 <bean id="record" name="r1,r2 r3;r4" class="di.SprRecord">
    <constructor-arg name="kor" value="20"></constructor-arg>
</bean>
 4)
 <bean id="record" " name="r1,r2 r3;r4" class="di.SprRecord"
    p:kor="50" p:eng="60" p:math="70">
 5)
 <bean id="view" class="di.SprRecordView">
    <property name="record" ref="record"></property>
</bean>
```

id : 빈 객체 고유 이름 (접근 가능자)
name : 객체의 이름(별칭)
class : 생성할 클래스
constructor-arg : 초기값 설정 (생성자 함수 사용)
property : 초기값 설정 (Setter함수 사용)

1) 이름이 record인 빈 객체 생성 / 별명 4개 : r1,r2,r3,r4 / SprReocrd 클래스 객체 생성.
초기값으로 kor 라는 프로퍼티에 20값 대입 (set함수가 존재해야 위와 같은 프로퍼티 설정이 가능).
2) 이름이 record인 빈 객체 생성 / 생성자(인자가 하나인)를 통해서 값 대입 & 생성.
3) 생성자 중에서 kor 값을 입력받는 생성자를 통해서 20값 대입하고, 생성.
4) 3개의 인자를 받는 생성자를 통해 kor = 50, eng = 60, math = 70 대입 & 생성.
5) 생성된 record 객체를 set함수를 통해 프로퍼티에 저장하고 SprRecordView를 생성.

참고로 값을 대입하는 경우에는 value, 참조(레퍼런스)를 대입하는 경우에는 ref 를 사용.

3) IoC 컨테이너 (스프링 컨테이너)

*** 제어의 역전 – 외부(컨테이너)에서 제어를 함.**

> 빈(Bean) :
스프링이 제어권을 가지고 직접 만들고 관계를 부여하는 오브젝트.

> 빈 팩토리(Bean Factory) :
빈(오브젝트)의 생성과 관계 설정 제어를 담당하는 IoC오브젝트.
좀 더 확장한 애플리케이션 컨텍스트(application context)를 주로 사용.

> 애플리케이션 컨텍스트: (IoC 컨테이너 or 스프링 컨테이너)
DI를 위한 빈 팩토리에 엔터프라이즈 애플리케이션을 개발하는 데 필요한 여러 가지 컨테이너 기능을 추가한 것.

> 설정정보/설정 메타정보
구성정보 or 형상정보 (XML)

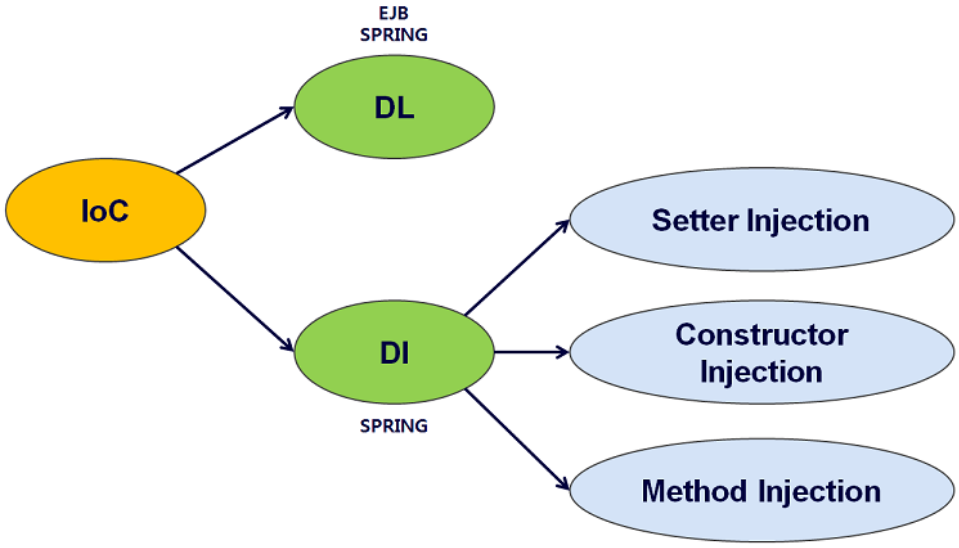
> 스프링 컨테이너(IoC 컨테이너) :
IoC 방식으로 빈을 관리한다는 의미에서 애플리케이션 컨텍스트나 빈 팩토리를 의미.

IoC컨테이너는 다른 용어로 빈 팩토리(Bean Factory), 애플리케이션 컨텍스트(Application Context)라고도 불림.
스프링의 IoC 컨테이너는 일반적으로 애플르케이션 컨텍스트를 말함.

* 빈 팩토리를 애플리케이션 컨텍스트 또는 IoC컨테이너라 말하기도 하지만, 사실 애플리케이션 컨텍스트는 빈을 좀 더 확장한 개념.
* 애플리케이션 컨텍스트는 그 자체로 IoC와 DI를 위한 빈 팩토리(Bean Factory)이면서 그 이상의 기능을 가짐.
* 빈팩토리와 어플리케이션컨텍스트는 각각 BeanFactory, ApplicationContext 두 개의 인터페이스로 정의
* ApplicationContext 인터페이스는 BeanFactory 인터페이스를 상속한 서브인터페이스.
* 실제로 스프링 컨테이너 또는 IoC 컨테이너라고 말하는 것은 바로 이 ApplicationContext 인터페이스를 구현한 클래스의 오브젝트.
* 컨테이너가 본격적인 IoC 컨테이너로서 동작하려면 POJO클래스와 설정 메타정보가 필요.
  (스프링 애플리케이션 : POJO 클래스와 설정 메타정보를 이용해 IoC 컨테이너가 만들어주는 오브젝트의 조합.)

*** IoC 컨테이너**

어떠한 객체의 명세서를 작성하고, 스프링 라이브러리는 해당 명세대로 객체를 생성. 생성된 객체(그리고 디펜던시)들을 보관하는 공간을 의미.



*** EJB** : Enterprise Java Bean (엔터프라이즈 자바빈) – 효율적으로 서버 관리를 해주고, 또 프로그램 관련 문제들을 알아서 처리해준다는 개념.
(연결 관계가 복잡하고, 무겁고, 독립적이지 못하다)
*** DL** : Dependency Lookup
*** DI** : Dependency Injection

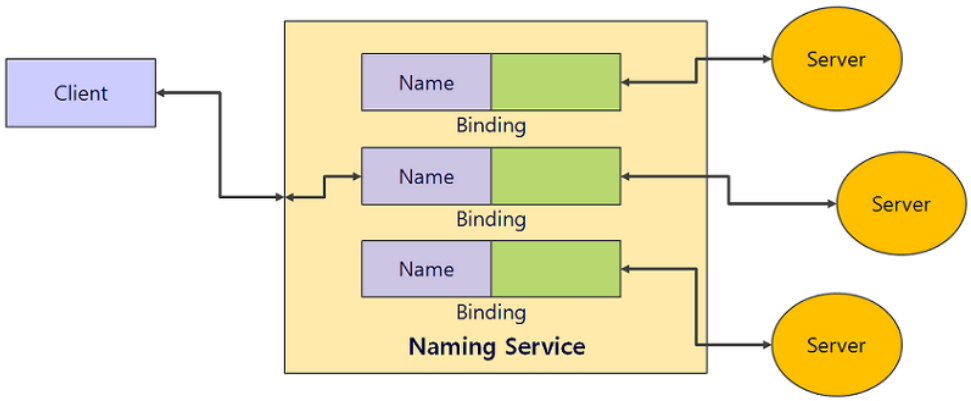
* **DL(Dependecy Lookup)** - JNDI 같은 저장소에 의하여 관리되고 있는 bean을 개발자들이 직접 컨테이너(Container)에서 제공하는 API를 이용하여 lookup하는 것을 말함. 따라서 container와의 종속성이 생김. (JNDI 컨테이너에 의존성이 강하다.)
오브젝트간에 디커플링(decoupling)을 해주는 면에서 장점이 있지만 이렇게 만들어진 오브젝트는 컨테이너 밖에서 실행 할 수 없고 JNDI외의 방법을 사용할 경우 JNDI관련 코드를 오브젝트내에 일일히 변경해 줘야 하며 테스트하기 매우 어렵고 코드 양이 매우 증가하고 매번 Casting해야 하고 NamingException같은 checked exception을 처리하기 위해서 exception처리구조가 매우 복잡해지는 단점이 있음.
EJB container, Spring container 에서 지원.

* **DI (Dependecy Injection)** – 각 class 사이의 의존관계를 빈 설정 정보를 바탕으로 containe가 자동적으로 연결해 주는 것을 말함. 따라서 looku과 관련된 코드들이 오브젝트 내에서 완전히 사라지고 컨테이너에 의존적이지 않은 코드를 작성할 수 있음.
단지 빈 설정 파일에서 의존관계가 필요하다는 정보를 추가하면 됨.
1) Setter Injection – 클래스 사이의 의존관계를 연결시키기 위하여 setter 메소드를 이용하는 방법. property tag 사용.

2) Constructor Injection – 클래스 사이의 의존관계를 연결시키기 위하여 constructor를 이용하는 방법. constructor-arg tag 사용.
3) Method Injection – 싱글톤(Singleton) 인스턴스와 non singleton 인스턴스의 의존관계를 연결시킬 필요가 있을 때 사용하는 방법.

참고) JNDI (Java Naming and Directory Interface)

* 엔터프라이즈 애플리케이션을 위한 네이밍과 디렉토리 서비스 표준 인터페이스
* Java 소프트웨어 클라이언트가 이름(Name)을 이용하여 데이터 및 객체를 찾을 수 있도록 도와주는 디렉토리 서비스에 대한 Java API.



[Naming Service 구조]

Naming Service (네이밍 서비스)

* Java Naming and Directory Interface(JNDI) API를 이용하여 자원(Resource)을 찾을 수 있도록 도와주는 서비스. Naming 서비스를 지원하는 Naming 서버에 자원을 등록하여 다른 어플리케이션에서 사용할 수 있도록 공개하고, Naming 서버에 등록되어 있는 자원을 찾아와서 이용할 수 있게 함.
* DNS서버 같은 개념.

JNDI(Java Naming and Directory Interface)

- 디렉토리 서비스에서 제공하는 데이터 및 객체를 발견하고 참고하기 위한 자바 API.
- J2EE 플랫폼의 일부.
- 여러 대의 서버 간에 JNDI를 이용하여 객체를 등록, 참조하여 이용.
- javax.naming 패키지 안에 존재.
- 여러 웹 서버(톰캣, 웹로직, 제우스 등)에서 사용.
- 기본 네임스페이스는 java:com/env.

4) AOP

IoC , DI 서비스 추상화와 더불어 스프링의 3대 기반 기술 중의 하나.

AOP는 Aspect Oriented Programming 의 약자인데 그 의미는 무엇인가?

4-1) Aspect Oriented Programming의 의미는?

-> Aspect 를 만드는 프로그램 방법.
-> Aspect 지향 프로그램.

우리말로 풀이하면 관점 지향 프로그래밍

* 여기서 Aspect는 측면 또는 관점이라는 의미로 볼 수도 있고, 부가적인 업무를 의미하기도 함.

"전통적인 객체지향기술의 설계방법으로는 독립적인 모듈화가 불가능한 트랜잭션 경계설정과 같은 부가 기능을 어떻게 모듈화 할 것인가"
"트랜잭션(핵심 기능과 부가 기능)의 분리"

횡단 관심사와 이에 영향 받는 객체 간의 결합도를 낮추는 것!

AOP(관점 지향 프로그래밍)는 왜 사용하는가?

- 주 업무가 아닌 부가적인 업무가 강한 응집력을 가지고 있는 경우, 소스 관리 및 개발 업무 진행의 복잡해지고, 어려워짐.
- 즉 서비스 추상화가 어려워짐. 이러한 문제를 해결하기 위한 프로그래밍 기법으로 OOP(Object Oriented Programming)의 보완적 개념.

- 횡단 관심사와 이에 영향 받는 객체 간 결합도를 낮추는데 목적이 있다. 쉽게 말해 클래스들이 공통으로 갖는 기능이나 절차 등을 하나의 것으로 묶어 빼내어 별도로 관리하려는 목적.
- 이러한 부가적인 업무의 예로 로그인(Login), 트랜잭션(Transaction), 보안(Security), 캐싱(Caching)과 같은 내부 처리(비지니스, Business) 작업이 있다.

AOP는 메인 프로그램의 비즈니스 로직으로부터 2차적 또는 보조 기능들을 고립시키는 프로그램 패러다임(관점 지향 프로그래밍 패러다임)이며, 애플리케이션의 핵심적인 기능에서 부가적인 기능을 분리해서 애스펙트(Aspect)로 정의하고 설계하여 개발하는 방식.

위에서도 언급했지만, 스프링 프레임워크에서의 애스펙트(Aspect)란
- **주업무가 아닌 업무.**
- **보조업무 : 로그, 트랜잭션, 보안처리.** 를 의미함.

4-2) AOP의 구현이란?

- 주 업무가 아닌 보조적인 업무를 주 업무를 처리하는 코드에서 분리하는 것.

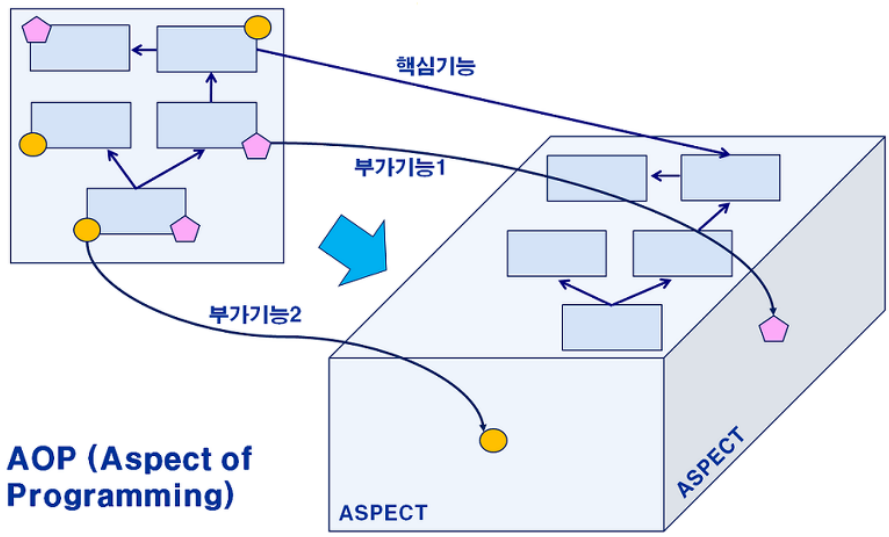
4-3) AOP의 장점은 무엇인가?

- 전체 코드 곳곳에 흩어져 있는 다양한 관심 사항이 하나의 장소로 응집.
- 여타 서비스 모듈이 자신의 주요 관심 사항(또는 핵심 기능)에 대한 코드만 포함하고 그 외의 관심 사항은 모두 Aspect로 옮겨지므로 코드가 깔끔해지고, 가독성이 높아짐.

관점 지향 프로그래밍 패러다임 : 이것은 수평으로 관심항목의 분리를 허용하고, 관점 지향 소프트웨어 개발의 기초를 형성하여 모듈화를 시키려 하고자 하는 것이다. 관점 지향 소프트웨어 개발이 모든 엔지니어링 분야에 관련되므로, 관점 지향 프로그래밍은 소스코드 레벨에서 관심사항들의 모듈화를 지원하는 프로그래밍 기술과 툴들을 포함한다.
관점 지향 프로그래밍은 프로그램을 명확한 부분으로 나누는 것을 수반한다. 모든 프로그래밍 패러다임은 이들 관심사항들을 구현, 추상화, 구성하는 추상적 개념을 제공하는 분리되고, 독립적인 통로들을 통해 Grouping의 같은 레벨과 관심사항들의 캡슐화(Encapsulation)를 지원한다. 그러나 어떤 관심사항들은 구현의 이런 형태를 거역하고, 이들이 프로그램 내에서 다중 추상적 개념들에 영향을 끼치기 때문에 횡단관심사(cross-cutting concerns)라고 불린다.

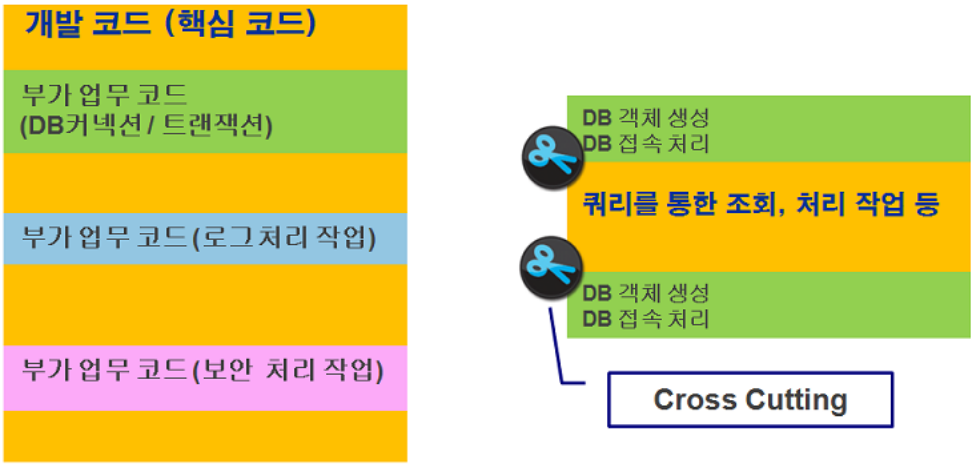
4-4) AOP의 시각화

* 핵심적인 기능에서 부가적인 기능을 분리하여 애스팩트라는 독특한 모듈로 만들어서 설계하고 개발하는 방법.



* 독립 애스펙트를 이용한 부가기능의 분리와 모듈화. (애스팩트를 모두 독립시킴) - 위의 그림 참조

4-5) AOP는 실제 코드에서 어떻게 처리되는가?



기존의 개발 방식은 보조 업무를 담당하는 코드가 주 업무 코드 사이사이에 포함되어 있음.

보조 업무가 주 업무 코드에 포함될 때에는 다음과 같은 일이 발생.

*** 동일한 작업 반복.**

*** 보조 업무의 작업 코드가 변경될 시, 해당 보조 업무를 사용하는 모든 주 업무 코드의 소스 수정 필요.**

*** 주 업무 코드보다 더 많은 양의 보조 업무 코드 – 특히 DB 객체 생성 및 접속, 예외 처리, DB 닫기 등.**

AOP는 이러한 보조 업무 코드를 주 업무 코드에서 별도로 분리하여 작성하고, 필요할 때에만 도킹(Docking)하여 사용하는 것은 어떨까? 하는 발상에서 나온 개념.

* 여기서 핵심 코드(핵심 관심사)는 **Core(Primary) Concern**.

* 부가/보조 업무 코드(횡단 관심사)는 **Cross-Cutting Concern**.

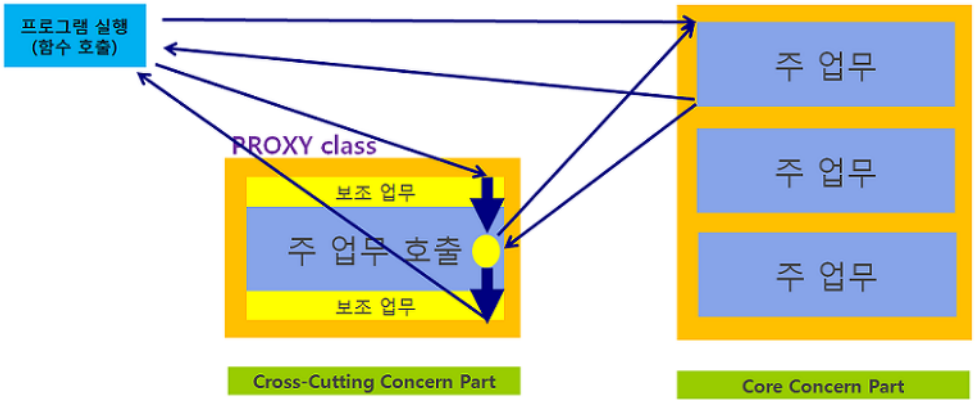
위의 오른쪽 그림에서 보듯이 주 업무에서 보조 업무를 횡단으로 잘라내었다는 의미로 Cross-Cutting 이라 불림.

5) AOP Proxy (AOP 프록시)

5-1) 프록시(Proxy)란?

- 클라이언트가 사용하려고 하는 실제 대상인 것처럼 위장하여 클라이언트의 요청을 받아주어 처리하는 대리자 역할.
프록시의 단어 자체로는 '대리인'이라는 의미를 내포하고 있음.
스프링 AOP에서의 프록시란 말 그대로 대리하여 업무를 처리. 함수 호출자는 주요 업무가 아닌 보조 업무를 프록시에게 맡기고, 프록시는 내부적으로 이러한 보조 업무를 처리.

이렇게 함으로써, 주 업무 코드는 보조 업무가 필요한 경우, 해당 Proxy 만 추가하면 되고, 필요없게 되면 Proxy를 제거하면 됨.
보조 업무의 탈 부착이 쉬워지고, 그리하여 주 업무 코드는 보조 업무 코드의 변경으로 인해서 발생하는 코드 수정 작업이 필요 없게 됨.



- 프록시의 호출 및 처리 순서 -

1) Proxy 호출
2) 보조 업무 처리
3) Proxy 처리 함수(메서드) 가 실제 구현 함수(메서드) 호출 및 주 업무 처리
4) 제어권이 다시 Proxy 함수(메서드)로 넘어오고 나머지 보조 업무 처리
5) 처리 작업 완료 후, 호출 함수(메서드)로 반환.

5-2) 프록시의 사용 목적

* 클라이언트가 타깃(Target)에 접근하는 방법을 제어.
* 타깃에 부가적인 기능을 부여.

(실제 이러한 목적으로 어떻게 사용되는지는 아래에서 보자!)

5-3) 프록시 구현

* 먼저 사칙 연산을 위한 인터페이스(Interface)와 실제 기능을 구현한 클래스(Class) 정의.

```
// 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
}
```

여기서, 실제 사칙 연산을 하는 데에 있어서 실제 소요되는 시간 측정 이라는 기능이 필요하다고 가정하자. 그렇다면 어떻게 해야 할까?
위의 예제 코드에서 cal.add(3, 4) 를 사용하고 있으니, 이 시간 측정을 위한 코드를 일단 myCalculator 클래스의 add 메서드에 추가하여 보자!

```
package aoptest;

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;
    }

...

}
```

위와 같이 시간을 측정하기 위한 업무(실제 연산 업무가 아니기 때문에 보조 업무)가 add 메서드에 추가됨. 하지만, 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.

그렇다면 좀 더 코드를 깔끔하게 하고, 관리도 쉽게 하고, 직관적인 프로그래밍을 위해서는 어떻게 변화를 줄 수 있을까?
바로 방금(또 이전 포스팅에서) 배운 주 업무와 보조 업무를 분리(크로스 컷팅, Cross Cutting)하고 보조 업무를 프록시(Proxy)에게 넘긴다!

AOP Proxy 를 구현한 예제 코드

아래의 클래스와 메서드에 대한 설명.

* (1) InvocationHandler 인터페이스를 구현한 객체는 invoke 메소드를 구현해야 함.
 해당 객체에 의하여 요청 받은 메소드를 리플렉션 API를 사용하여 실제 타깃이 되는 객체의 메소드를 호출해준다. (실제 LogPrintHandler 클래스의 invoke 메서드 내에서 **method.invoke(target, args)** 메서드를 호출하는데, 이는 주 업무의 메서드를 호출하는 것. main 메서드에서 Proxy.newProxyInstance 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.newProxyInstance(
    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) 그런다음 주 업무 클래스의 메서드를 호출하게 되면 프록시 클래스의 invoke 메서드가 호출되어 자신의 보조 업무를 처리하고, 주 업무의 메서드를 호출한다.

* (3) invoke() : 메소드를 실행시킬 대상 오브젝트와 파라미터 목록을 받아서 메소드를 호출한 뒤에 그 결과를 Object 타입으로 돌려준다.

```
 // 보조 업무를 처리할 프록시 클래스 정의 (여기서는 LogPrintHandler 라이름 지었음)
 public class LogPrintHandler implements InvocationHandler { // 프록시 클래스 (핸들러)
     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 {
         Log log = LogFactory.getLog(this.getClass());
         StopWatch sw = new StopWatch();
         sw.start();
         log.info("Timer Begin");
         int result = (int) method.invoke(target, args); // (3) 주업무를 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();

     Calculator proxy_cal = (Calculator) Proxy.newProxyInstance( // (1)
         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)); // (2) 주 업무 처리 클래스의 add 메서드를 호출
 }
```

[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 것. (스프링 개념 없이)
스프링을 통해서라면 AOP 는 XML과 어노테이션(Annotation)을 통해서 더 쉽게 구현할 수 있음.

6) Dynamic Proxy (동적 프록시)

*** 프록시의 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 작업이 될 수 있음.
 - 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.)
 - 메서드가 많아지고 트랜잭션 적용의 비율이 높아지면 트랜잭션 기능을 제공하는 유사한 코드가 여러 메서드에 중복돼서 나타날 수 있음.

*** 해결책은?**

- 다이내믹 프록시(Dynamic Prox)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 이란 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 관계된 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 한번 만 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) 패턴(Patterns)

패턴은 말 그대로 어떤 일정한 형태나 양식 또는 유형을 뜻함.
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 :**

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

*** 프록시 패턴 :**

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 패턴에 대한 예제 코드 : 출처 - 위키피디아
(http://en.wikipedia.org/wiki/Decorator_pattern)

```
// 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());
    }
}
```

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 재정의 함.
(기존 클래스에 장식(데코레이팅)된 형태로 클래스를 정의)

데코레이터 패턴의 단점 :

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

데코레이터 패턴의 장점 :
- 기존 코드는 수정하지 않고, 확장 및 추가가 가능함.

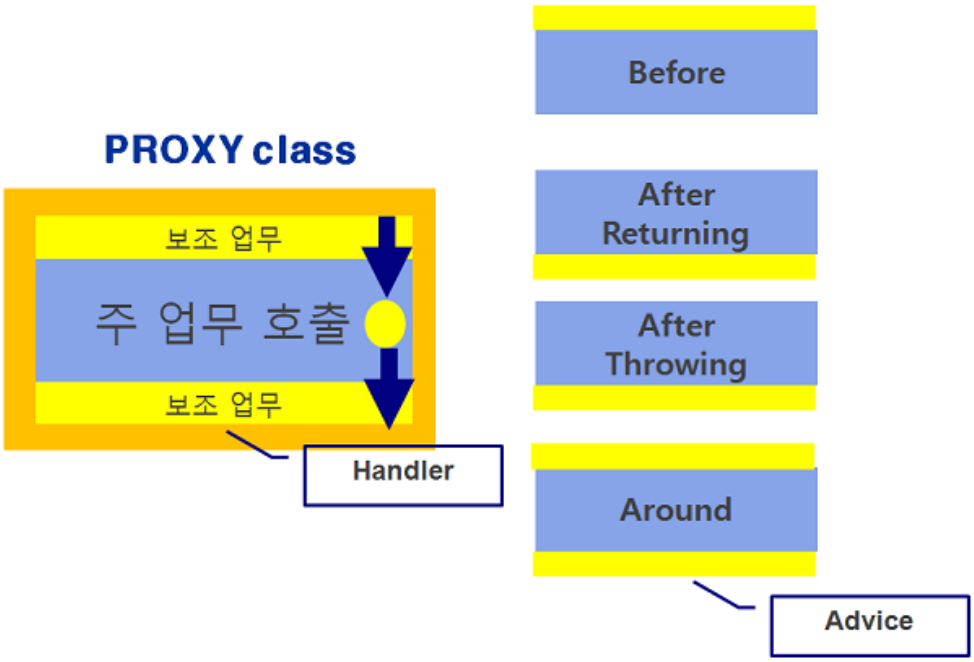
패턴의 종류에는 옵저버 패턴, 데코레이터 패턴, 프록시 패턴, 팩토리 패턴, 싱글턴 패턴, 커맨드 패턴, 어댑터 패턴, 퍼사드 패턴, 템플릿 패턴 등등 다양하며 좀 더 자세히 알고 싶은 경우에 **Head First Design Patterns (저자 에릭 프리먼)** 서적을 추천함.

데코레이터 패턴 위키피디아 : http://en.wikipedia.org/wiki/Decorator_pattern

8) 스프링에서의 AOP (Aspect-Oriented Programming)

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

```
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));
}
```

```
package aoptest;
public class LogPrintAroundAdvice implements MethodInterceptor {// Around Advice

    @Override
    public Object invoke(MethodeInvocation method) throws Throwable {
        Log log = LogFactory.getLog(this.getClass());
        StopWatch sw = new StopWatch();
        sw.start();
        log.info("Timer Begin");
        Object result = method.proceed(); // 주 업무 호출
        sw.stop();
        log.info("Timer Stop – Elapsed Time : "+ sw.getTotalTimeMillis());
        return result;
    }
}
```

```
<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) : MethodeInterceptor
- MethodeInterceptor : Around Advice 사용할 때, 구현해야 하는 인터페이스.
- MethodeInterceptor 는 InvocationHandler 와 동일한 기능을 수행하나 차이점은 InvocationHandler 의 경우에는 프록시할 타깃 객체를 InvocationHandler 인터페이스 구현 객체가 알고 있어야 하나, 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.

```
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));
}
```

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

*** 위의 X 표시된 코드는 주석처리하고 AOP를 활용하여 Advice 를 적용한 JAVA 코드**

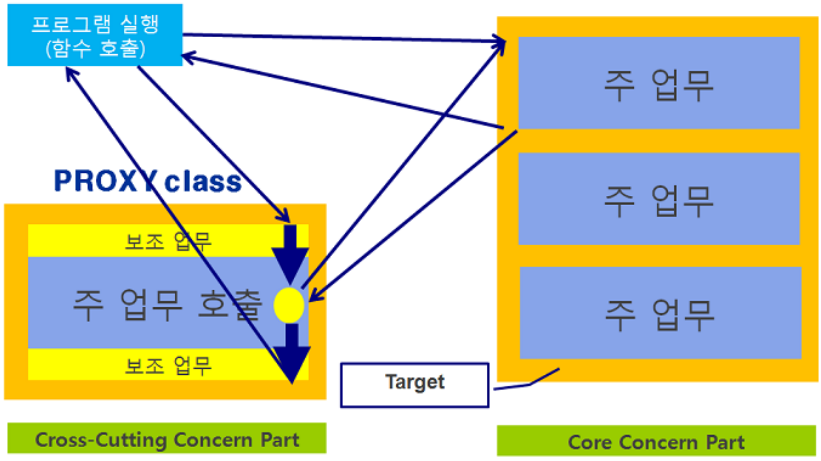
* 객체의 생성과 조립을 메인 함수에서 코드에서 관리하였지만, 부가적인 파트(객체 생성과 의존성)는 XML 파일로 이동시킴.
* XML 빈 객체의 이름 및 속성을 변경함으로써, 기존 코드에 바로 적용 가능.
* 부가 코드와 주 코드를 분리하여 필요할 때 도킹(Docking)하여 사용 가능.
* 위와 같은 동적 프록시(Dynamic Proxy)는 ApplicationContext의 getBean 메서드를 통해서 개체를 얻어옴.

2) 스프링 프레임워크에서의 4가지 형태의 Advice

9) 스프링 AOP (Aspect Oriented Programming)

*** 포인트 컷 & 조인포인트**

- 위빙 (Weaving) : 보조업무가 프록시(Proxy)를 통해서 주 업무에 주입되는 것. 즉, 타깃 객체에 애스펙트를 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.



*** 예제 코드**

```
<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">
<proporty 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>logPrintBeforAdvice</value>
           <value>logPrintAfterReturningAdvice</value>
           <value>logPrintAfterThrowingAdvice</value> */
        </list>
     </property>
 </bean>
```

- 코드 설명 -

1) 특정 함수(add, subtract) 만 조인포인트 함수가 되도 록 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 사용가능.
* **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(부가기능)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.NameMatchMeth
odPointcutAdvisor">
     <property name="pointcut" ref="namePointcut"></property>
     <proporty name="mappedNames">
         <list>
             <value>add</value>
             <value>substract</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.RegexpMethodPo
intcutAdvisor">
     <property name="pointcut" ref="namePointcut"></property>
     <proporty 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, substract) 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]
. : 어떠한 문자든 상관없다. (한 개의 문자가 옴)
*** :** 임의의 문자가 0개 이상 올 수 있다. (한 개 이상의 문자가 옴)

*** 위빙, 조인포인트, 포인트컷을 다시 한 번 정리하자면**
위빙 : 크로스 컷팅 되어 있는 관심사(Concern)를 다시 결합하는 것.
조인포인트 : 위빙하는 함수들이 있는 곳 지점, 시점 (스프링에서는 메소드만을 조인 포인트로 사용함.)
포인트 컷 : 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!**
- 분리하고 도킹하는 것이 자유로움.
- 조인포인트를 쓸 수 있는 것이 Method에 한정.

10) 빈 후처리기(Bean Post Processor)

*** 빈 후처리기(Bean Post Processor)란?**

- 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.

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

```
// 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 Postprocessing)**

1) 빈 객체(Bean Instance) 생성 (생성자(Constructor) 또는 팩토리 메서드(Factory Method) 사용).
2) 빈 프로퍼티(Bean Property)에 값과 빈 레퍼런스 설정.
3) Aware Interface에 정의된 Setter 메서드 호출
4) 빈 인스턴스(Bean Instance)를 각 빈 후처리기
   (Bean Post Processor)의 postProcessBeforeInitialization() 에 전달.
5) 초기화 Callback 메서드 호출
6) 빈 인스턴스(Bean Instance)를 각 빈 후처리기(Bean Post Processor)의
   postProcessAfterInitialization() 에 전달.
7) 빈 사용 준비 완료
8) 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

스프링의 빈(bean) 객체는 기본적으로 싱글톤(Singleton)으로 생성되는데, 그 이유는 사용자의 요청이 있을 때마다 애플리케이션 로직까지 모두 포함하고 있는 오브젝트를 매번 생성하는 것은 비효율적이다. 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 저장하게 되는 인스턴스 변수를 두지 않음. 애플리케이션 로직을 포함하고 있는 오브젝트는 대부분 싱글톤 빈(Singleton Bean)으로 만들면 충분함.

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) 스프링MVC (Model2)

스프링 MVC (Model2)

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

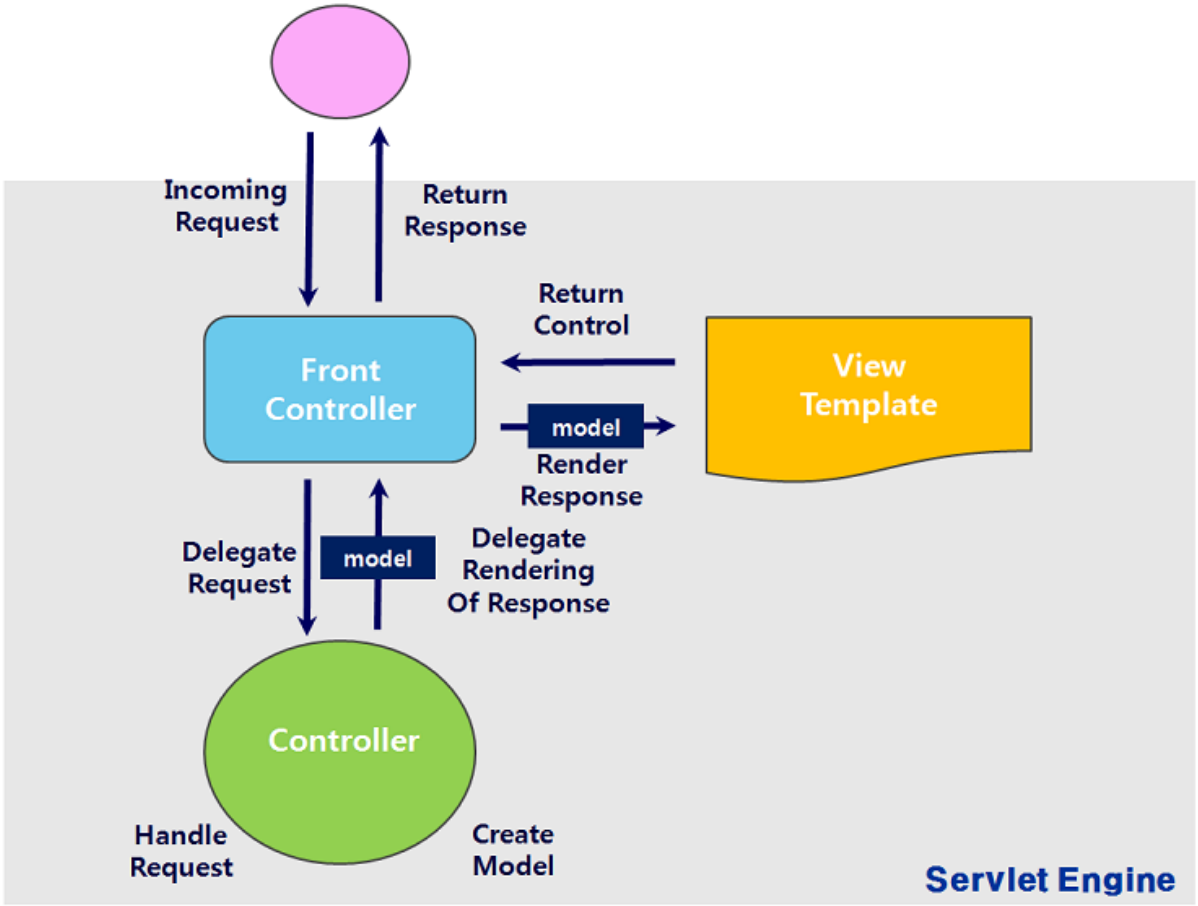
(Front Part : Servlet (DispatcherServlet.class))

*** Controller :** 화면 단에서 보여줄 데이터들을 미리 만들어 놓는 것. 그 데이터는 모델(Model)이라 함.

*** Model :** 데이터

- Model, Controller, Front Controller, View 를 잘 구조화하는 것이 MVC 모델을 제어하고 있는 라이브러리가 할 역할.

[스프링 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>
</server-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      제
공 (Controller)
    @Override
    public ModelAndView handleRequest(HttpServletRequest request, HttpSer
vletResponse 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 코드는 게시판 MVC(스프링 MVC – 서블릿 DispatcherServlet)로 공지사항 리스트를 관리하고, 특정 번호의 공지사항을 열람하기 위한 공지사항 페이지를 구현. (스프링의 MVC의 개념을 설명하는 데에 있어서 게시물 등록, 수정, 삭제 처리 함수는 크게 다른 게 없으므로 생략.)

ModelAndView : Controller의 처리결과를 보여줄 View와 View에서 사용할 값을 전달하는 클래스.
ModelAndView의 mv 오브젝트는 web.xml에서 설정했었던 DispatcherServlet이 받아서 처리함.
${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
 1) ServletContext의 초기 파라미터
 2) Session의 유효시간 설정
 3) Servlet/JSP에 대한 정의
 4) Servlet/JSP 매핑
 5) Mime Type 매핑
 6) Welcome File list
 7) Error Pages 처리
 8) Listen/Filter 설정
 9) 보안

12) - 스프링MVC (고전적인 JSP 코딩)

* 스프링 MVC 샘플 코드를 보기 이전에 고전적인 JSP 코딩 방식을 보도록 하겠습니다.

```
// NoticeController.java (1단계 DB 제어 클래스를 통한 DB데이터 가져오기)
package controllers.customer;
public class NoticeController implements Controller { // 스프링이 제공 (Controller)

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

        String _page = request.getParameter("pg");
        String _field = request.getParameter("f");
        String _query = request.getParameter("q");

        int page = 0;
        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;

        NoticeDao dao = new NoticeDao();
        List<Notice> list = dao.getNotices(page, field, query); // 페이지번호, 검색항목, 검색어

        ModelAndView mv = new ModelAndView("notice.jsp");
        mv.addObject("listData", list);

        return mv;
    }
}
```

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

- 코드 설명 -

* NoticeDao 클래스는 DB 커넥션 개체를 생성하고, DB와 연결하며, 쿼리를 통해 데이터를 가져오는 작업을 처리하여 주는 클래스.

* AOP개념이라고 볼 수 있는데 NoticeDao 를 통해서 DB 개체 생성, 접속, 쿼리 실행, 종료 등에 대한 처리를 위임한 클래스 개체를 만들어 사용.

* Controller Interface 구현

* Controller의 handleRequest 함수 구현 (XML에 매핑된 함수임)

```
// notice.jsp
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>    //    core
library
<table>
<tbody>
<c:forEach var="n" items="${listData}">
    <tr>
        <td class="seq">${n.seq}</td> // n.getSeq();
        <td
class="title"><a href="noticeDetail.bugs?seq=${n.seq}">${n.title}</td>
        <td class="writer">${n.writer}</td>
        <td class="writer">${n.regDate}</td>
        <td class="writer">${n.hit}</td>
    </tr>
</c:forEach>
</tbody>
</table>
```

* 참고로 JSP페이지에 스크립트릿, 표현식, HTML 등의 코드가 섞이게 되면 가독성이 떨어지게 되는데, 보다 가독성을 높이기 위한 언어 및 태그를 표현언어(EL, Expression Language) / JSTL 코드를 사용.

* ${n.seq} 는 실제 n.getSeq() 함수를 호출하는 것이며, 이 setter 함수는 이미 정의되어 있어야 함.
* EL은 $와 {}를 사용하여 값을 표현

13) 스프링MVC (AOP with XML)

*** AOP 를 적용하여 코딩한 예제입니다.**

```
// 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,bugs?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,bugs?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.bugs" class="controllers.customer.NoticeController">
    <property name="noticeDao" ref="noticeDao"></property>
</bean>

<bean
name="customer/noticeDetail.bugs" class="controllers.customer.NoticeDetail
Controller">
    <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 해야 하는 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 방법은?

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

* 즉 클래스(컨트롤러) 하나에 여러 개의 URL-메서드 매핑(Mapping)이 가능하다면??

-> 애노테이션! (Annotation)

14) AOP with Annotation(애노테이션)

*** 애노테이션(Annotation) & 오토와이어링(Autowiring)**

- 하나의 컨트롤러(Controller)와 하나의 페이지를 매핑하는 것은 작업도 번거롭고, 관리도 힘듬.
  그렇다면 하나의 함수에 하나의 URL을 어떻게 매핑할것인가?
  -> 애노테이션(Annotation)을 사용하자!

- 애노테이션은 무엇인가?
  -> 컴파일러를 거친 후에도 코드에 남아있는 특수한 주석(Comments)

애노테이션은 특수한 주석으로 자체적으로 어떠한 기능을 수행하는 것은 아니지만, 애노테이션이 가리키는 지시어(정보)를 기반으로 컴파일러는 애노테이션이 적용된 클래스 또는 매서드, 프로퍼트 등에 적절한 기능을 부여한다.

-> 애노테이션은 @(골뱅이) 로 주석을 시작하며, 클래스, 매서드, 변수 등에 적용됨.

```
// dispatcher-servlet.xml (객체를 생성해 주고, Injection해 줌)
<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
@RequestMapper("/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); // 페이지 번호, 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-메쏘드 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)
* 그런데! 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!
-> 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 파라메터 Model model을 선언만 해주면 됨. 해당 Model 객체가 생성되어 참조가 자동으로 됨.

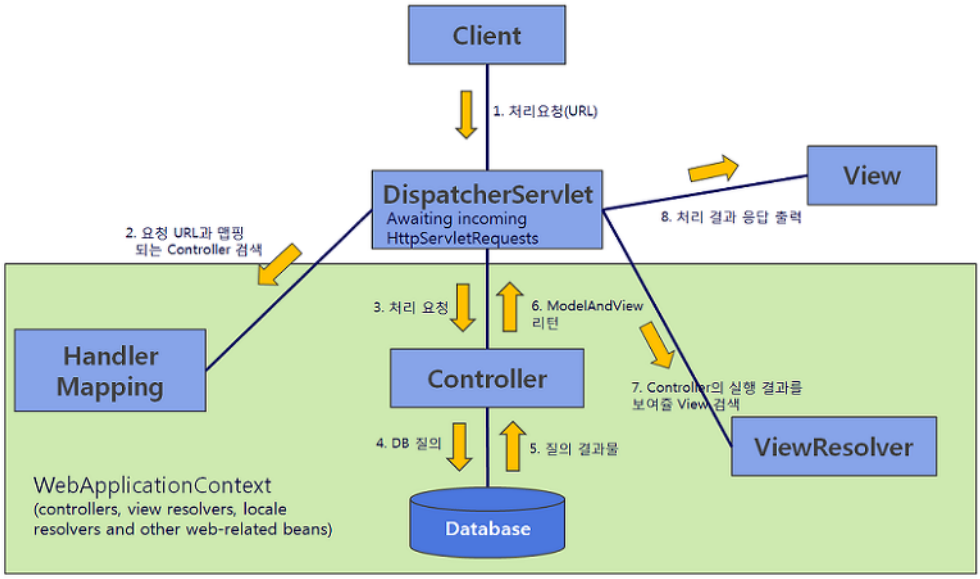
[동일한 페이지를 다르게 처리]
@RequestMapping(value={"/customer/noticeReg.oz"}, method=RequestMethod.GET)
@RequestMapping(value={"/customer/noticeReg.oz"}, method=RequestMethod.POST)

[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 변수 레퍼런스 클래스의 인스턴스(속성 값)의 이름(변수 명)과 동일하다면 이를 자동으로 맵핑(대입)하여 줌.

15) 스프링 MVC 프로세스

*** 스프링 MVC 프로세스 모델**



*** 스프링 MVC의 주요 구성 요소**

1) 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)

2) HandlerMapping : 클라이언트의 요청 URL을 어떤 컨트롤러가 처리할지를 결정. (스프링 제공)

3) Controller : 클라이언트의 요청을 처리한 뒤, 그 결과를 DispatcherServle에 알려준다. (실제 로직을 담당)

4) SQL 쿼리를 통한 데이터 질의문 전달.

5) 데이터베이스가 반환(Return)한 **데이터 획득**.

6) ModelAndView : 컨트롤러가 처리한 결과 정보 및 뷰 선택에 필요한 정보를 담음.

7) ViewResolver : 컨트롤러의 처리 결과를 생성할 뷰를 결정. (스프링 제공)

8) View : 컨트롤러의 처리 결과 화면을 생성 (Freemarker 등)

*** 스프링 MVC 구조 및 처리 순서**

1) 클라이언트의 요청이 **DispatcherServlet**에 전달됨.

2) DispatcherServlet은 **HandleMapping**을 사용하여 클라이언트의 요청을 처리할 컨트롤러(Controller)를 검색.

3) 매치되는 컨트롤러가 있다면 **요청을 해당 컨트롤러에게 전달.**

DispatcherServlet은 컨트롤러 객체의 **handleRequest() 메서드를 호출**하여 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 **ViewResolve**r로부터 응답 결과를 생성할 뷰 객체를 구함.

JSP, Velocity, Freemarker 등 다양한 뷰를 지원.

- AbstractCachingViewResolver
- XmlViewResolver
- ResourceBundleViewResolver
- UrlBasedViewResolver
- InternalResourceViewResolver
- FreeMakerViewResolver
- ViewResolvers Chaining

8) 뷰는 클라이언트에 전송할 응답을 생성.