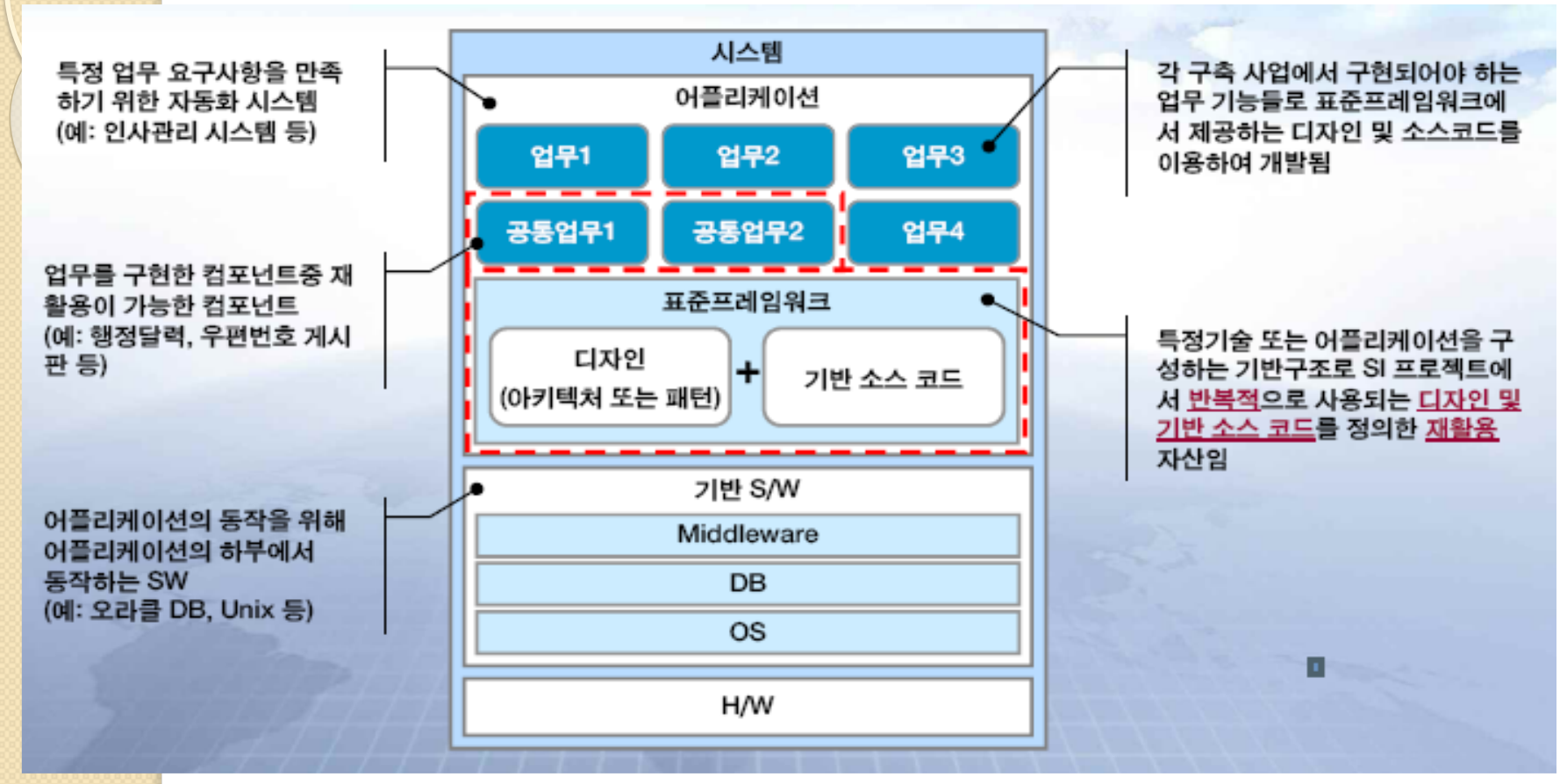
Spring

강사 강태광

1-1. 어플리케이션 프레임워크 개념

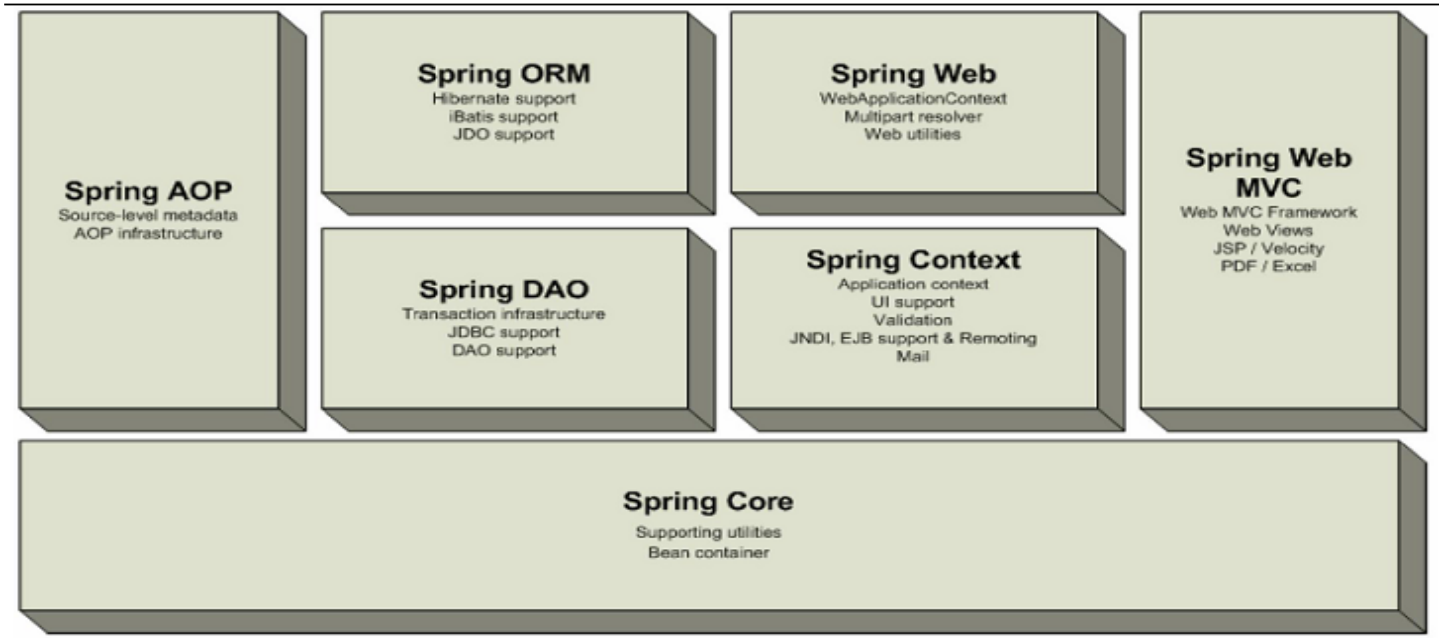
1. 프로그래밍에서 특정 운영체제를 위한 응용프로그램 표준구조를 구현하는클래스와 라이브러리 모임



2. Spring 설치
   - 사이트 : https://spring.io/tools/sts/all

1-2. Spring 프레임워크

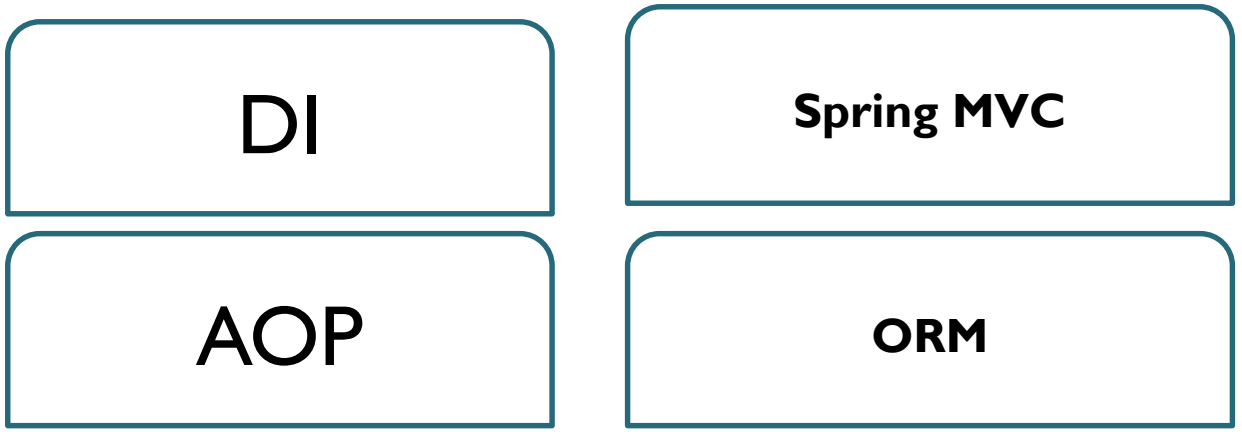
1. 대표적 오픈소스기반의 어플리케이션 프레임워크
2. EJB 의 복잡성및 빈약한 데이터 모델을 해결 하기위한 POJO 기반의 OSS 프레임워크



1) CORE: IoC, DI, DDD를 기반으로하는 디자인 패턴
2) MVC : 웹어플리케이션 제작을 위한 기반제공
3) AOP : 프록시기반의 AOP 기반 인프라 제공
4) ORM : Hibernate, iBatis의3rdParty 플러그인 제공
5) DAO: 데이터를 액세스하기 위한 기반제공

1-2. Spring 프레임워크

1. 대표적 오픈소스기반의 어플리케이션 프레임워크
2. EJB 의복잡성및 빈약한 데이터 모델을 해결 하기위한 POJO 기반의 OSS 프레임워크



1) CORE: IoC, DI, DDD를 기반으로하는 디자인 패턴
2) MVC : 웹어플리케이션 제작을 위한 기반제공
3) AOP : 프록시기반의 AOP 기반 인프라 제공
4) ORM : Hibernate, iBatis의3rdParty 플러그인 제공
5) DAO: 데이터를 액세스하기 위한 기반제공

2-1. IOC

1) IOC 개념
- 기존의 프로그래밍에서 객체의 생성, 제어, 소멸 등의 객체의 라이프 사이클을 개발자가 관리 하던 것을 컨테이너 에게 그 제어권을 위임하는 프로그래밍 기법
- 특정 작업을 수행하기 위해 필요한 다른 컴포넌트들을 직접 생성하거나 획득하기 보다는 이러한 **의존성들을 외부 에 정의하고 컨테이너에 의해 공급**받는 프로그래밍 기법. 객체의 생성에서부터 생명주기의 관리까지 모든 객체에 대한 제어권이 바뀌었다는 것을 의미

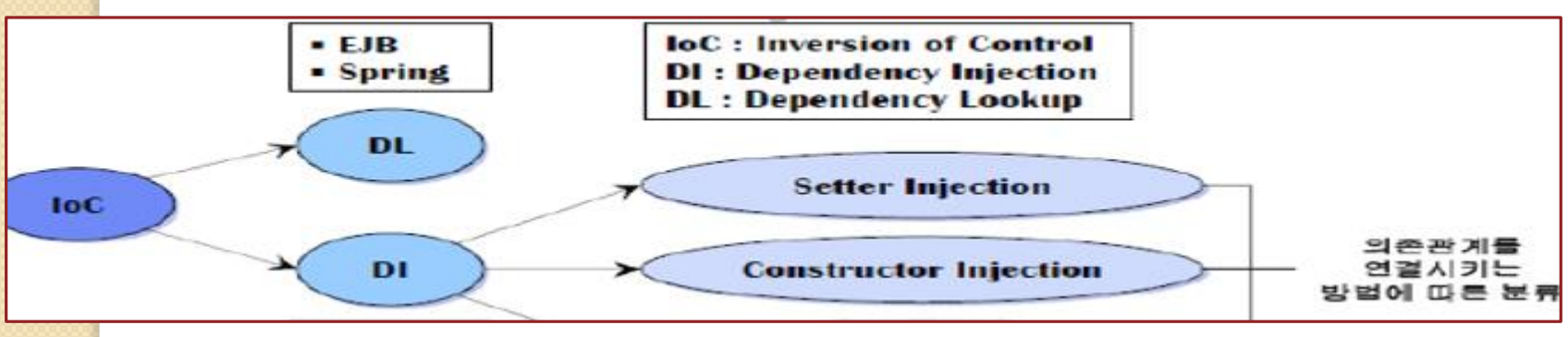
2) IoC 적용 장점
- 유지보수 용이성: Loosely coupling을 통해 코드 변경에 쉽게 대처가 가능하여 유지보수가 용이
- 용이한 환경설정: 어플리케이션 로직으로부터 의존관계를 분리하여 상황에 따라 자유로운 환경 설정이 가능
- 재사용 용이성: 의존관계를 일일이 Lookup할 필요없이 외부에서 조립하여 재사용성이 강화됨
- 테스트 용이성: 의존관계에 대한 고려를 최소화하여 컴포넌트를 개별적으로 테스트할 수 있음. Mock 객체를 사용하여 DB를 사용하지 않고도 테스트 가능

2-1. IOC

3) IoC (Inversion of Control)의 구현 방법

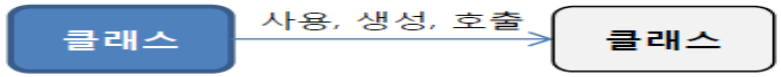
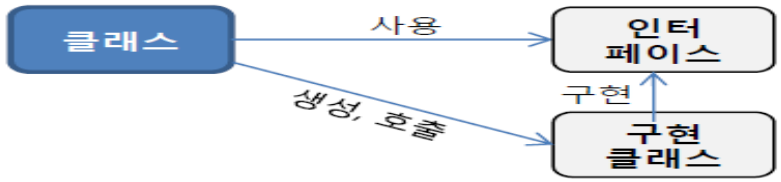
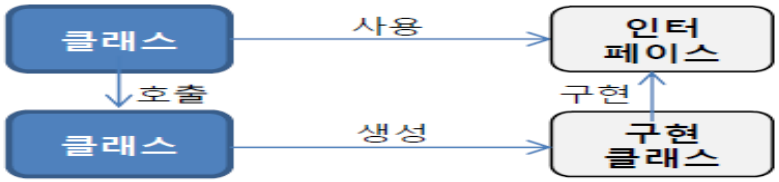
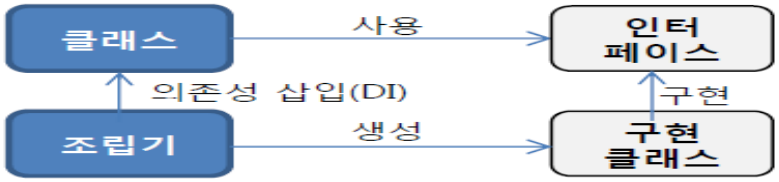
[1] DL(Dependency Lookup)
-의존성 검색. 저장소에 저장되어 있는 빈(Bean)에 접근하기 위하여 개발자들이 컨테이너에서 제공하는 API를 이용하여 사용하고자 하는 빈(Bean)을 Lookup하는 것

[2] DI(Dependency Injection)
- 의존성 주입. 각 계층 사이, 각 클래스 사이에 필요로 하는 의존 관계를 컨테이너가 자동으로 연결해 주는 것
- 각 클래스 사이의 의존 관계를 빈 설정(Bean Definition) 정보를 바탕으로 컨테이너가 자동으로 연결해 주는 것
- DL 사용 시 컨테이너 종속성이 증가하여, 이를 줄이기 위하여 DI를 사용
- 종류 : Setter Injection, Constructor Injection, Method Injection



2-1. IOC

4) DI관점의 클래스 호출방식을 통한 IoC의 이해

호출 방식	도해	설명
일반적인 클래스 호출		클래스 내에 선언과 구현이 한몸이기 때문에 다양한 형태로 변화가 불가능
인터페이스를 이용한 클래스 호출		클래스를 인터페이스와 구현 클래스로 분리 구현클래스 교체가 용이하여 다양한 형태로 변화가 가능하지만 구현클래스 교체시 호출 클래스의 소스를 수정
팩토리패턴을 이용한 클래스 호출		팩토리방식은 팩토리가 구현 클래스를 생성하므로 클래스는 팩토리를 호출하는 코드로 충분 구현클래스 변경 시 호출클래스에는 영향을 미치지 않고, 팩토리만 수정하면 됨
IoC를 이용한 클래스 호출		팩토리 패턴의 장점을 더하여 어떠한 것에도 의존하지 않는 형태로 구성 가능 실행 시점에 클래스 간의 관계가 형성 의존성이 삽입된다는 의미로 IoC를 DI(Dependency Injection)라는 표현으로 사용

2-1. IOC

5) DI(Dependency Injection)의 종류

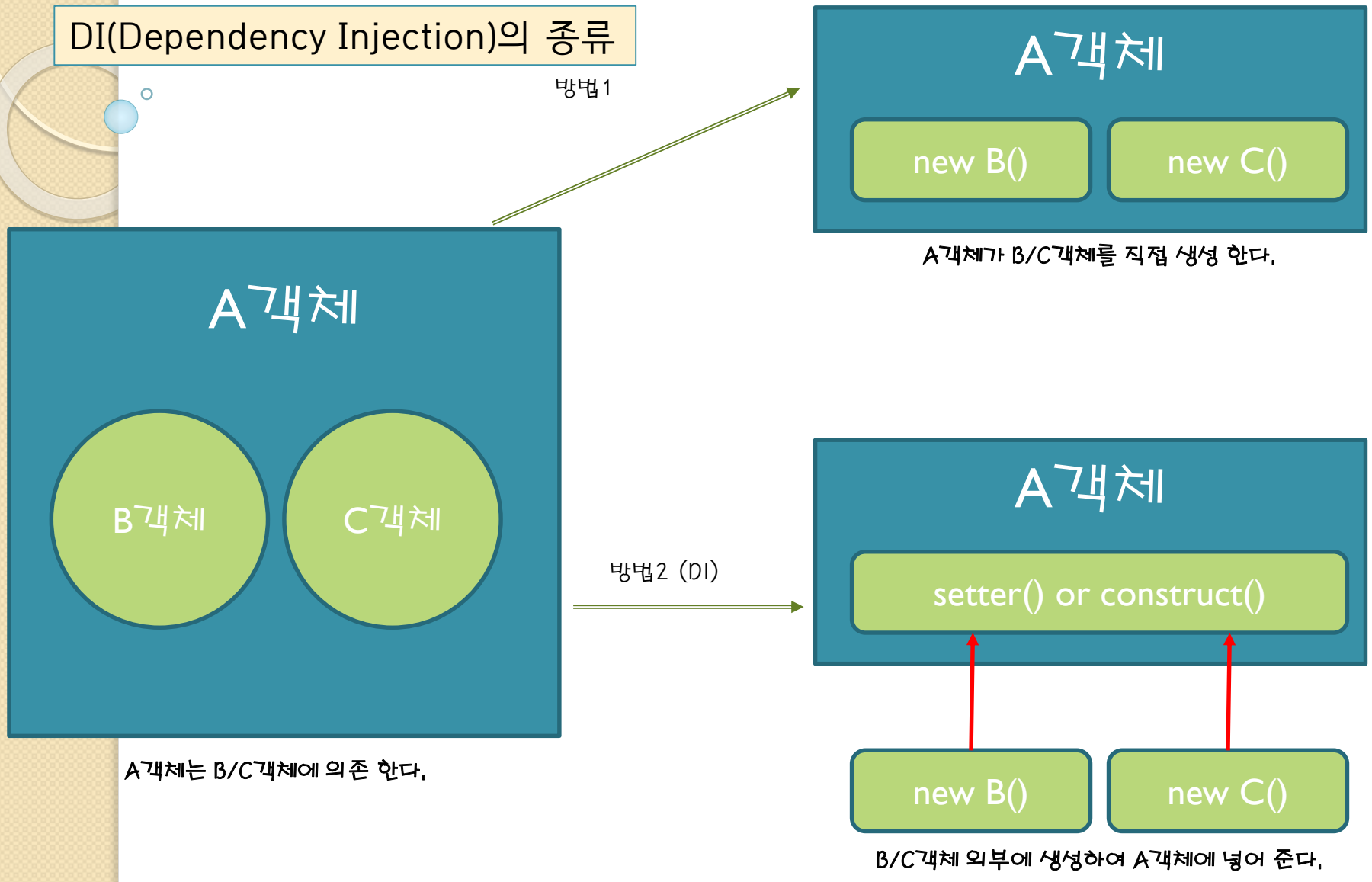
1) Setter Injection
- 인자가 없는 생성자나 인자가 없는 static factory 메소드가 bean를 인스턴스화하기 위해 호출된 후 bean의 Setter 메소드를 호출하여 실체화하는 방법
- 객체를 생성 후 의존성 삽입 방식이기에 구현 시에 좀더 유연하게 사용
- 세터를 통해 필요한 값이 할당되기 전까지 객체를 사용할 수 없음
- Spring 프레임워크의 빈 설정 파일에서 property 사용

2) Constructor Injection
- 생성자를 이용하여 클래스 사이의 의존 관계를 연결
- Setter메소드를 지정함으로 생성하고자 하는 객체가 필요로 하는 것을 명확하게 알 수 있음
- Setter메소드를 제공하지 않음으로 간단하게 필드를 불변 값으로 지정이 가능
- 생성자의 파라미터가 많을 경우 코드가 복잡해 보일 수 있음
- 조립기 입장에서는 생성의 순서를 지켜야 하기에 상당히 불편함
- Spring 프레임워크의 빈 설정 파일에서 Constructor-arg 사용

Setter Injection 장점	Constructor Injection 장점
① 생성자 Parameter 목록이 길어 지는 것 방지 ② 생성자의 수가 많아 지는 것 방지 ③ Circular dependencies 방지	① 강한 의존성 계약 강제, ② Setter 메소드 과다 사용 억제 ③ 불필요한 Setter 메소드를 제거함으로써 실수로 속성 값을 변경하는 일을 사전에 방지

2-1. IOC

DI(Dependency Injection)의 종류



부품을 생성하고 조립하는 라이브러리 집합체.

2-1. IOC

DI(Dependency Injection)의 적용

```
<bean id="myInfo"
class="com.oracle.DI02.MyInfo">
<property name="name">
<value>홍길동</value>
</property>
<property name="height">
<value>170</value>
</property>
<property name="weight">
<value>72</value>
</property>
<property name="hobbys">
<list>
<value>바둑</value>
<value>낚시</value>
<value>대화</value>
</list>
</property>
<property name="bmiCalculator">
<ref bean="bmiCalcaulator"/>
</property>
</bean>
```

기초데이터

List 타입

다른 빈객체 참조

```
private String name;
private double height;
private double weight;
private ArrayList<String> hobbys;
private BMICalculator bmiCalculator;
```

```
public void setBmiCalculator(BMICalculator bmiCalculator) {
    this.bmiCalculator = bmiCalculator;
}
public void setName(String name) {
    this.name = name;
}
public void setHeight(double height) {
    this.height = height;
}
public void setWeight(double weight) {
    this.weight = weight;
}
public void setHobbys(ArrayList<String> hobbys) {
    this.hobbys = hobbys;
}
```

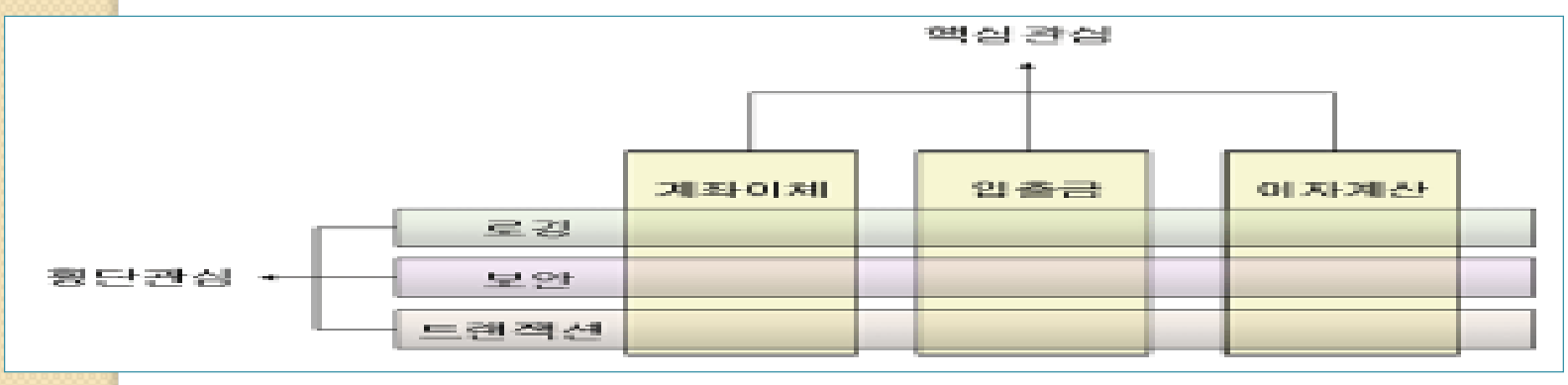
2-2. AOP(Aspect Oriented Programming)

1. 기능 외적인 관점의 용이한 적용을 위한 패러다임. AOP의 개념
 1) 관점 지향 프로그램(Aspect Oriented Programming, AOP)의 정의
  - 핵심 관심사(Core Concerns)에 대한 관점과 횡단 관심사(Cross-cutting Concerns)에 대한 관점들로 프로그램을 분해해 객체지향 방식(OOP)에서 추구하는 모듈을 효과적으로 지원하도록 하는 프로그래밍 기법.
 2) AOP의 등장배경

구분	등장배경
기능의 분산 (Scattering)	OOP의 SRP 원칙 실 세계에서 지키기 어렵다 그로 인해 객체지향으로 설계한 모듈에 보안이나 모니터링 기능이 분산해서 존재한다
코드의 혼란 (Trangling)	Infrastructure Service(Tracing, Logging, Monitoring 등)의 많은 요구로 인해 최초 OOP방식으로 작성된 코드가 지저분한 코드로 바뀌게 된다
OOP 코드 유지	OOP에 충실한 모듈의 구현

2-2. AOP(Aspect Oriented Programming)

2. AOP의 등장배경
 1) 기존 OOP 한계
 - 하나의 클래스에 핵심과 횡단 관심사가 혼재되어 프로그램의 가독성 및 재활용성에 비효율이 발생, 관심사를 분리/단순화 하여 코드의 재활용과 유지보수의 효율성을 극대화 .
 2) AOP의 시스템 적용 예시
 - OOP의 구조는 계좌이체 클래스, 입출금 클래스, 이자계산 클래스로 나뉘게 되고, 로깅, 보안, 트랜잭션 기능이 분산해서 존재
 - 타 프로젝트에서 기 구성 시스템 중 보안기능만을 분리한다고 할 경우 기존의 OOP방식으로는 이 보안 역할만을 별도 분리 불가.



2-2. AOP(Aspect Oriented Programming)

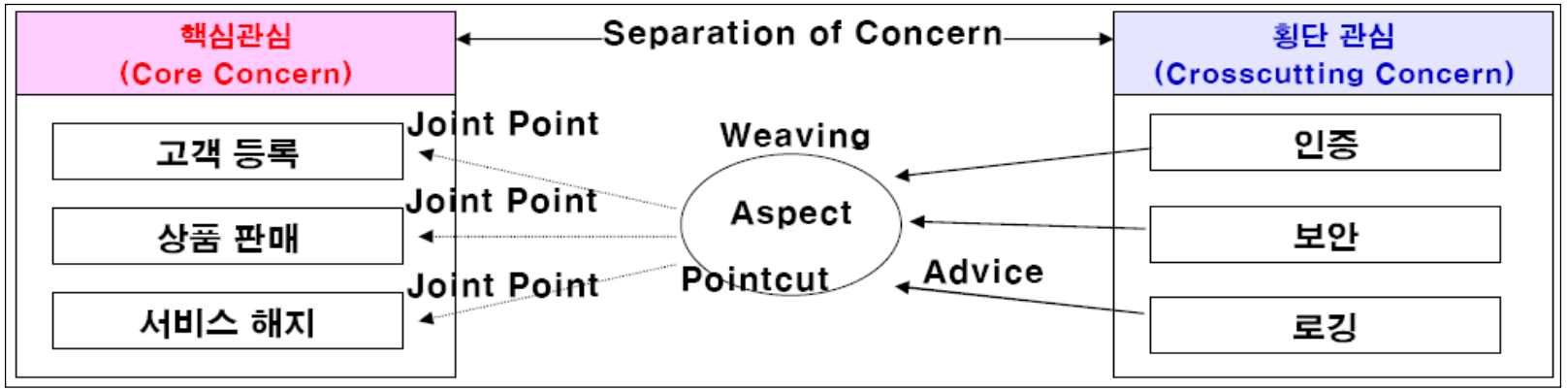
3. AOP의 특징

구분	특징
모듈화	횡단 관심사를 포괄적이고 체계적으로 모듈화
캡슐화	횡단 관심사는 Aspect라는 새로운 단위로 캡슐화하여 모듈화가 이루어짐
단순화	핵심 모듈은 더 이상 횡단 관심사의 모듈을 직접 포함하지 않으며 횡단 관심사의 모든 복잡성은 Aspect로 분리.

2-2. AOP(Aspect Oriented Programming)

4. AOP의 개념도 및 주요 요소

1) AOP의 개념도



- 핵심과 횡단의 분리를 이루고, AOP가 핵심 관심 모듈의 코드를 직접 건드리지 않고 필요한 기능을 작동하는 데는 weaving 또는 cross-cutting 작업 필요

2-2. AOP(Aspect Oriented Programming)

4. AOP의 개념도 및 주요 요소
1) AOP의 주요 요소

구분	특징
핵심 관심 (Core Concern)	**-시스템이 추구하는 핵심 기능 및 가치** - Business 업무
횡단 관심 (Cross-cutting)	- **핵심 관심에 공통적으로 적용되는 부가적인 요구사항** - **보안, 인증, 로그작성, 정책 적용** 등 - Cross-cutting Concern
Joint Point	- **관심사를 구현한 코드에 끼워 넣을수 있는 프로그램의 Event**
Point-Cut	- 관심사가 주 프로그램의 어디에 횡단할 것인지를 나타내는 위치 - aop:pointcut id="pub1" expression="within(com.oracle.aop1.St*)" />
Advice	- **관심사를 구현하는 코드, 결합점에 삽입되어 동작할수 있는 코드** - Point-cut에 의해 매칭된 joint point에 실행할 작업 - BEFORE, AROUND, AFTER의 실행 위치 지정
Aspect	- **프로그램의 핵심관심사에 걸쳐 적용되는 공통 프로그램의 영역** - 특정 상황(point-cut)과 그 상황에서 수행할 작업 (advice)의 집합 - Point-cut 과 Advice를 합쳐 놓은 클래스 형태의 코드 - 특정 관심사에 관련된 코드만을 캡슐화
Weaving	- **Joint Point 에 해당하는 Advice를 삽입하는 과정** - Aspect와 핵심 관심사를 엮는(weave)것을 의미

2-2. AOP(Aspect Oriented Programming)

4. Spring AOP의 특징

특징	설명
표준자바 클래스	스프링의 경우 스프링 내에서 작성되는 모든 Advice는 표준 자바 클래스로 작성
Runtime 시점에서의 Advice 적용	Spring에서는 자체적으로 런타임 시에 위빙하는 "프록시 기반의 AOP"를 지원
AOP 연맹의 표준 준수	스프링은 AOP연맹의 인터페이스를 구현
메소드 단위 조인포인트만 제공	필드 단위의 조인포인트 등 다양한 조인포인트를 제공해 주는 AspectJ와 다르게 메소드 단위 조인포인트만 제공

스프링에서 AOP 구현 방법 : proxy를 이용

2-2. AOP(Aspect Oriented Programming)

5. Spring AOP Advice 종류(*)

Advice 종류	XML 스키마 기반 POJO 클래스 이용	@Aspect 애노테이션 기반	설 명
Before	<aop:before>	@Before	target 객체의 메소드 호출시 호출 전에 실행
AfterReturning	<aop:after-returning>	@AfterRetuning	target 객체의 메소드가 예외 없이 실행된 후 호출
AfterThrowing	<aop:after-throwing>	@AfterThrowing	target 객체의 메소드가 실행하는중 예외가 발생한 경우에 실행
After	<aop:after>	@After	target 객체의 메소드를 정상 또는 예외 발생 유무와 상관없이 실행 try의 finally와 흡사
Around	<aop:around>	@Around	target 객체의 메소드 실행 전, 후 또는 예외 발생 시점에 모두 실행해야 할 로직을 담아야 할 경우

2-2. AOP(Aspect Oriented Programming)

5. Spring AOP 구현 예시

1) <aop:config>태그를 이용한 config 작성

```
<bean id="common" class="spring.study.aop.CommonAdvice" />

<bean id="data" class="spring.study.aop.DataImpl"></bean>

<aop:config>
   <aop:aspect id="commonAdvice" ref="common">

      <aop:pointcut id="aroundPush" expression="within(spring.study.aop.*)"/>

      <aop:around pointcut-ref ="aroundPush" method="around" />

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

2-2. AOP(Aspect Oriented Programming)

5. Spring AOP 구현 예시

2) 횡단관심 Advice 구현

```
import org.aspectj.lang.ProceedingJoinPoint;

public class CommonAdvice {

    /* 1. around advice
     * 2. 실행 로직  - Advice가 적용될 target 객체를 호출하기 전화 후를 구해서 target 객체의
     *                메소드 호출 실행 시간 출력*/
    public void around(ProceedingJoinPoint point) throws Throwable{
        String methodName = point.getSignature().getName();
        String targetName = point.getTarget().getClass().getName();
        System.out.println("target 클래스명 : " + targetName + " 및 메소드명 : " + methodName);

        long startTime = System.nanoTime();
        System.out.println("[Log]핵심 로직 메소드 호출전 --> "+ methodName+" time check start");

        Object obj = point.proceed();

        long endTime = System.nanoTime();
        System.out.println("[Log]핵심 로직 메소드 호출 후 --> " + methodName+" time check end");
        System.out.println("[Log] "+ methodName+" 실행 소요 시간 "+(endTime - startTime)+"ns");

        System.out.println(targetName + " 메서드 실행후 리턴된 데이터 : " + obj);
    }
```

2-2. AOP(Aspect Oriented Programming)

5. Spring AOP 구현 예시

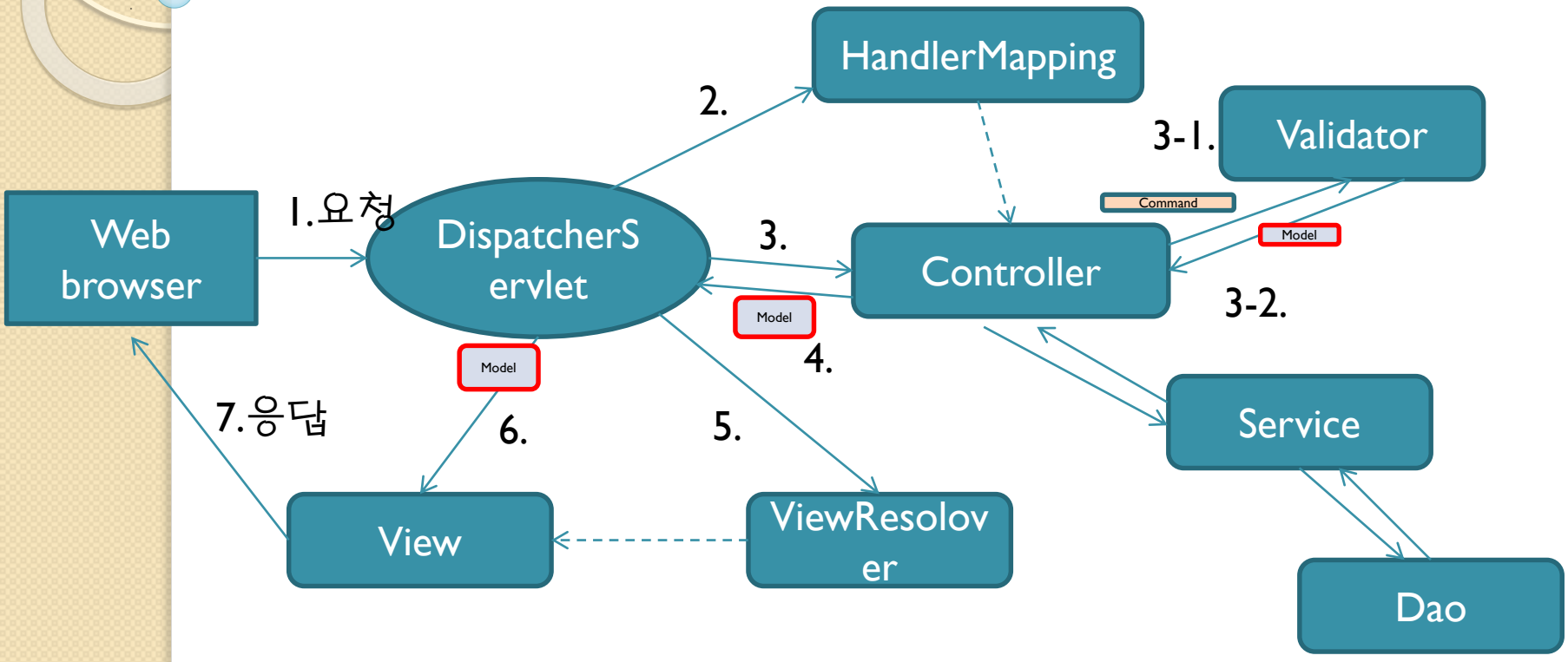
3) Annotation 적용 구현

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.annotation.Around;
@Aspect
public class CommonAdvice {
    @Pointcut("within(spring.study.aop.*)")
    private void adviceMethod(){}

    @Around("adviceMethod()")
    public void around(ProceedingJoinPoint point) throws Throwable{
        String methodName = point.getSignature().getName();
        String targetName = point.getTarget().getClass().getName();
        System.out.println("target 클래스명 : " + targetName + " 및 메소드명 : " + methodName);

        long startTime = System.nanoTime();
        System.out.println("[Log]핵심 로직 메소드 호출전 --> "+ methodName+" time check start");

        Object obj = point.proceed();

        long endTime = System.nanoTime();
        System.out.println("[Log]핵심 로직 메소드 호출 후 --> " + methodName+" time check end");
        System.out.println("[Log] "+ methodName+" 실행 소요 시간 "+(endTime - startTime)+"ns");

        System.out.println(targetName + " 메서드 실행후 리턴된 데이터 : " + obj);
    }
}
```

proxy대상의 실제 메소드 호출

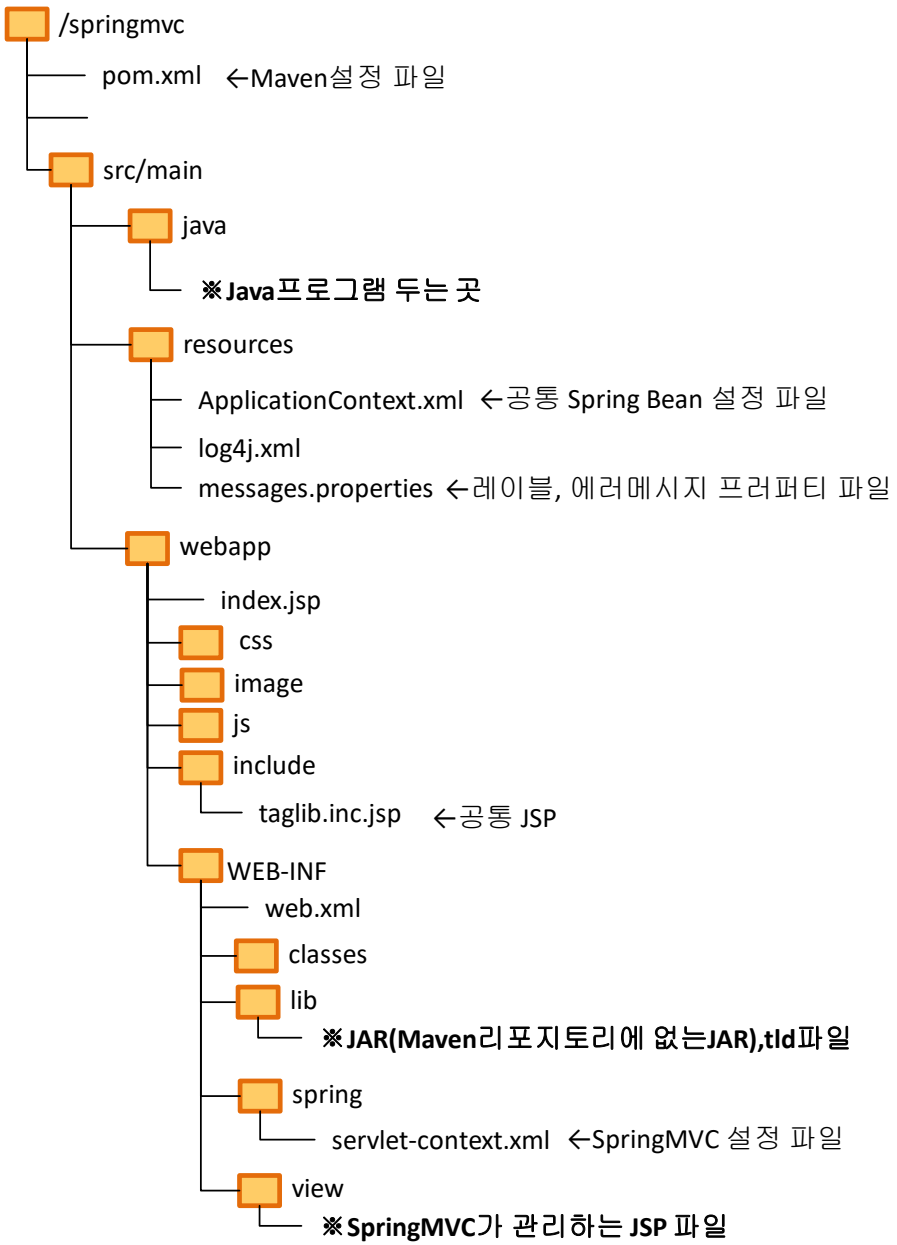
3. Spring MVC

1) MVC 흐름



3. Spring MVC

2) 파일구성



3. Spring MVC

1) 프로젝트 관리도구 MAVEN
Maven은 자바용 프로젝트 관리도구로 Apache Ant의 대안으로 만들어졌다.
- Maven은 Ant와 마찬가지로 프로젝트의 전체적인 라이프 사이클을 관리하는 도구 이며, 많은 편리함과 이점이 있어 널리 사용.
(프로젝트의 작성부터 컴파일, Test등 프로젝트 라이프사이클에 포함되는 각 테스트를 지원.)

- Maven은 필요한 라이브러리를 특정 문서(pom.xml)에 정의해 놓으면 내가 사용할 라이브러리 뿐만 아니라 해당 라이브러리가 작동하는데에 필요한 다른 라이브러리들까지 관리하여 NW를 통해서 자동으로 다운.
- Maven은 중앙 저장소를 통한 자동 의존성 관리를 중앙 저장소(아파치재단에서 운영 관리)는 라이브러리를 공유하는 파일 서버라고 볼 수 있고, 메이븐은 자기 회사만의 중앙 저장소를 구축

3. Spring MVC(Model 과 MoelAndView)

2) Spring MVC에서 model 과 **MoelAndView**

① Spring MVC에서 model
- Model은 파라미터 방식으로 메소드에 (Model model) 파라미터를 넣어주고 String형태로 리턴
- Model은 값을 넣을 때 addAttribute()를 사용

② ModelAndView는 컴포넌트 방식으로 ModelAndView 객체를 생성해서 객체형태로 리턴
- MoelAndView는 말그대로 Model과 View를 합쳐놓은 것으로,값을 넣을때 addObject()를 사용
- setViewName()으로 보낼 곳 View를 세팅

3. Spring MVC

3) 컨트롤러 패턴

- 단순 View(Jsp)로 이동하는 경우
- 단순 View(Jsp)로 이동하는 경우(모델 존재)
- 다른 URL로 이동하는 경우
- URL 리퀘스트로부터 값을 받는 경우
- Form으로부터 값을 받는 경우
- Form으로부터 값을 받아, 세션에 저장하는 경우

3. Spring MVC

3) 컨트롤러 패턴(단순 View(Jsp)로 이동하는 경우)

```
@Controller
public class SampleController {

    @RequestMapping("/sample")
    public String sample() {

        return "/toView";
    }
}
```

3. Spring MVC

3) 컨트롤러 패턴(단순 View(Jsp)로 이동하는 경우[모델 존재)])

```
@Controller
public class SampleController {

    @RequestMapping("/sample")
    public ModelAndView sample() {

        // View 설정
        ModelAndView mav = new ModelAndView("/toView");

        // 모델을 취득해서, 메시지 설정
        mav.addObject("message1",  " 메시지1");

        return mav;
    }
}
```

3. Spring MVC

3) 컨트롤러 패턴(다른 URL로 이동하는 경우)

```
@Controller
public class SampleController {
    // 포워드
    @RequestMapping("/sample1")
    public String forward() {
        return "forward:/hello.html";
    }

    // 리다이렉트
    @RequestMapping("/sample2")
    public String redirect() {
        return "redirect:/hello.html";
    }
}
```

3. Spring MVC

3) 컨트롤러 패턴(URL 리퀘스트로부터 값을 받는 경우)

```
@Controller
public class SampleController {
@RequestMapping(value="/sample", method={RequestMethod.GET})
        public ModelAndView sample(@RequestParam(value="userCd", required=true) Integer id,
                                                    @RequestParam String token) {

            // 바인드 시 에러처리
            if(bindingResult.hasErrors()) {
                ・・・생략
            }

            ModelAndView mav = new ModelAndView("/toView");
            return mav;
        }
    }
```

3. Spring MVC

3) 컨트롤러 패턴(Form으로부터 값을 받는 경우)

```
    @Controller
public class SampleController {

    @RequestMapping(value="/sample", method={RequestMethod.POST})
    public ModelAndView sample(@ModelAttribute LoginCommand command,
BindingResult bindingResult) {

        // 바인드시 에러처리
        if(bindingResult.hasErrors()) {
            ・・・생략
        }
        ModelAndView mav = new ModelAndView("/toView");
        return mav;
    }
}
```

3. Spring MVC

3) 컨트롤러 패턴(Form으로부터 값을 받아, 세션에 저장하는 경우)

```
@Controller
public class SampleController {

    @RequestMapping(value="/sample", method={RequestMethod.POST})
    public ModelAndView sample(WebRequest request, @ModelAttribute
LoginCommand command, BindingResult bindingResult) {
        // 바인드시 에러처리
        if(bindingResult.hasErrors()) {
            ・・・생략
        }
        // 세션에 데이터 저장
        request.setAttribute("loginUser", "hogehoge",
RequestAttributes.SCOPE_SESSION);

        ModelAndView mav = new ModelAndView("/toView");
        return mav;
    }
}
```

3. Spring MVC

4) 트랜잭션 전파 속성

2개 이상의 트랜잭션이 작동할 때, 기존의 트랜잭션에 참여하는 방법을 결정하는 속성
PlatformTransactionManager 인터페이스 보다 더욱 많이 사용되는 TransactionTemplate

PROPAGATION_REQUIRED(0)	DEFAULT : 전체 처리
PROPAGATION_SUPPORTS(1)	기존 트랜잭션에 의존
PROPAGATION_MANDATORY(2)	트랜잭션에 꼭 포함 되어야 함. – 트랜잭션이 있는 곳에서 호출해야 됨.
PROPAGATION_REQUIRES_NEW(3)	각각 트랜잭션 처리
PROPAGATION_NOT_SUPPORTED(4)	트랜잭션에 포함 하지 않음 – 트랜잭션이 없는 것과 동일 함.
PROPAGATION_NEVER(5)	트랜잭션에 절대 포함 하지 않음. – 트랜잭션이 있는 곳에서 호출하면 에러 발생

4. Security 설정

1) Pom 설정(보안 관련 라이브러리 추가)

```
<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-config</artifactId>
<version>3.2.5.RELEASE</version>
</dependency>

<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-core</artifactId>
<version>3.2.5.RELEASE</version>
</dependency>

<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-web</artifactId>
<version>3.2.5.RELEASE</version>
</dependency>

<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-taglibs</artifactId>
<version>3.2.4.RELEASE</version>
</dependency>
```

4. Security 설정

2) web.xml 설정

```
<!-- The definition of the Root Spring Container shared by
all Servlets and Filters -->

<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>
        /WEB-INF/spring/root-context.xml
        /WEB-INF/spring/appServlet/security-context.xml
</param-value>
</context-param>

<filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>
                        org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

4. Security 설정

3) security-context.xml

```
<security:http auto-config="true">
<security:intercept-url pattern="/login.html*" access="ROLE_USER"/>
<security:intercept-url pattern="/welcome.html*" access="ROLE_ADMIN"/>
</security:http>

<security:authentication-manager>
<security:authentication-provider>
<security:user-service>
<security:user name="user" password="123" authorities="ROLE_USER"/>
<security:user name="admin" password="123" authorities="ROLE_ADMIN,ROLE_USER"/>
</security:user-service>
</security:authentication-provider>
</security:authentication-manager>
```