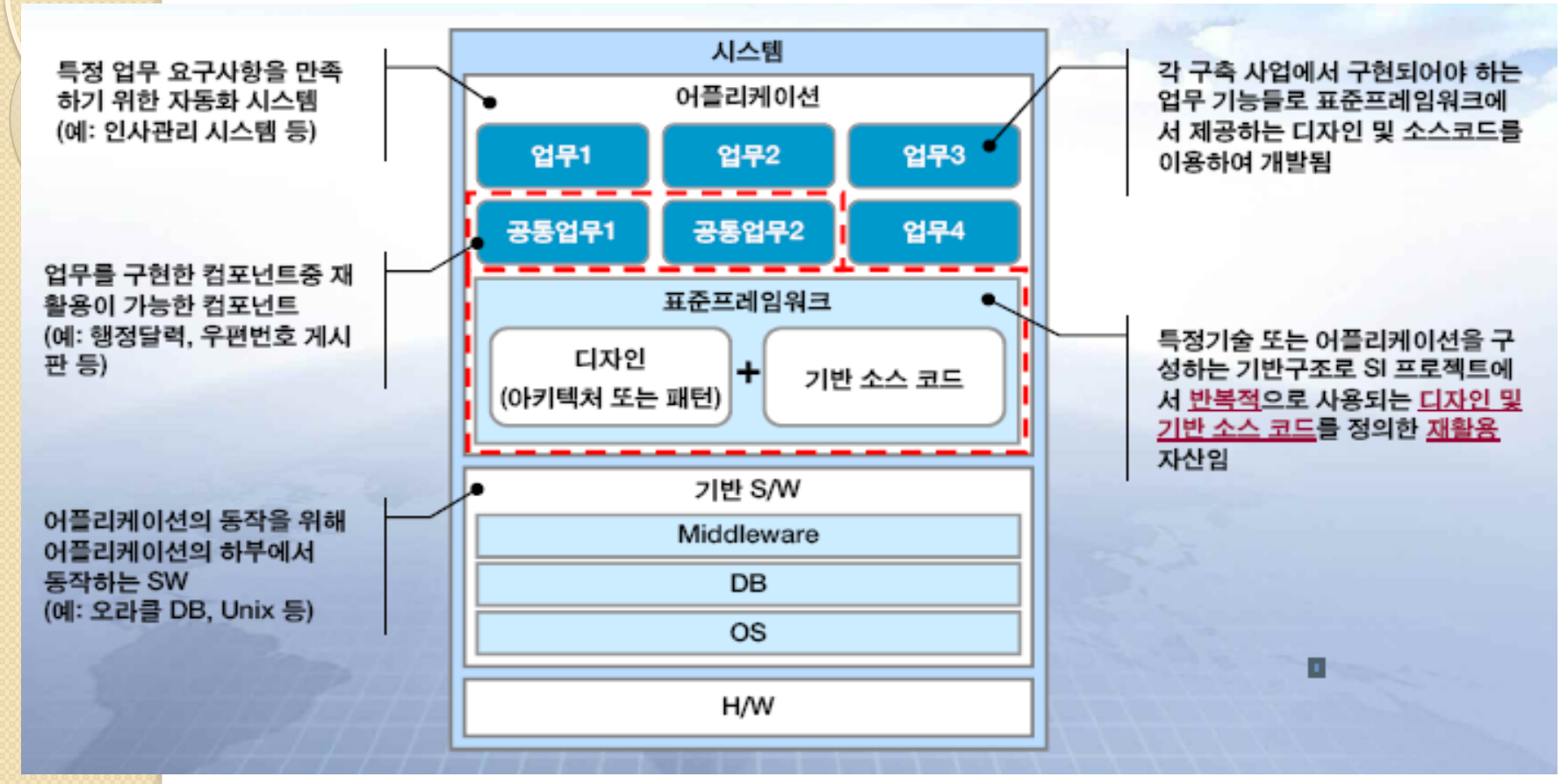
Spring

강사 강태광

어플리케이션 프레임워크 개념

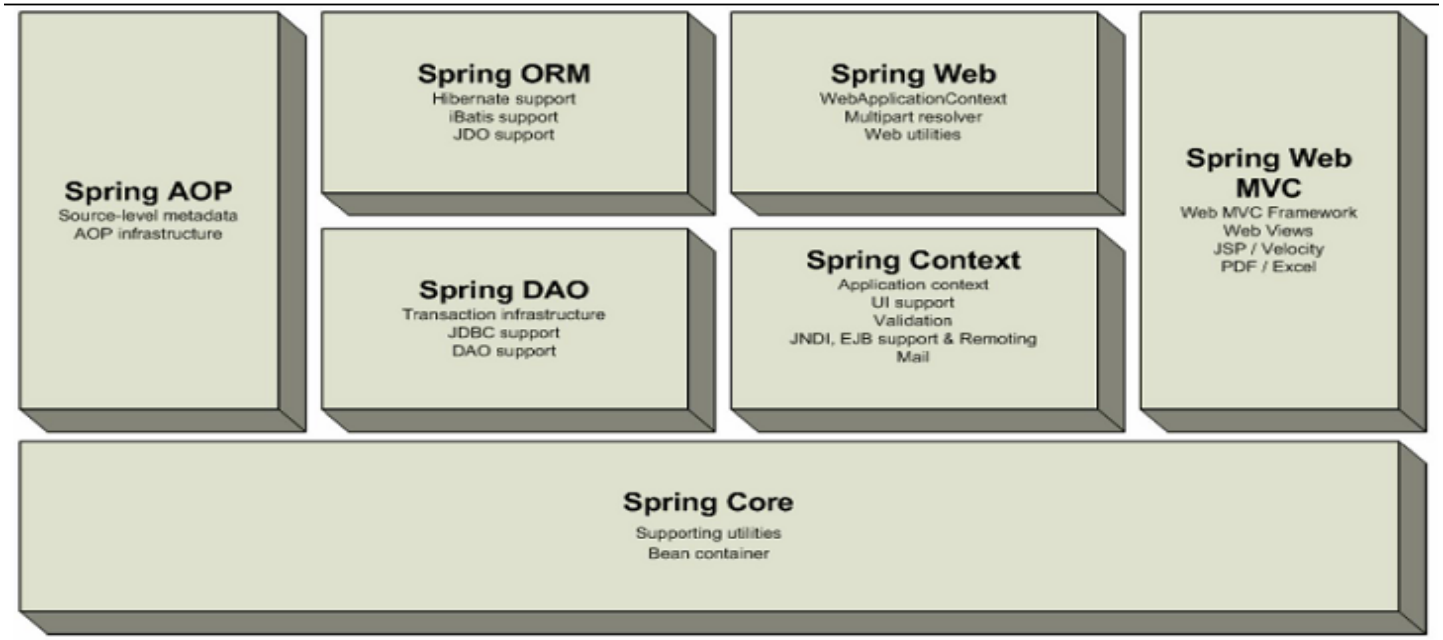
1. 프로그래밍에서 특정 운영체제를 위한 응용프로그램 표준구조를 구현하는클래스와 라이브러리 모임



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

Spring

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

I. 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

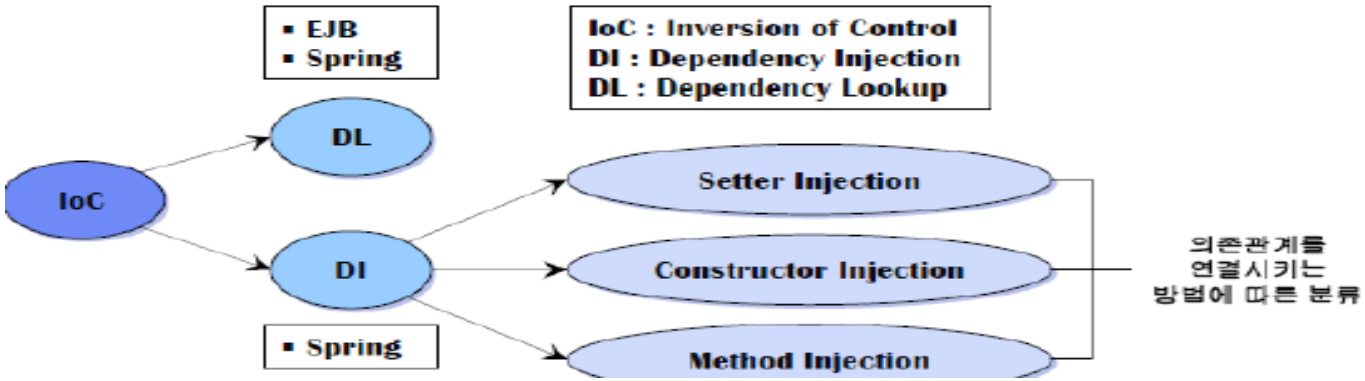
1. IOC

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

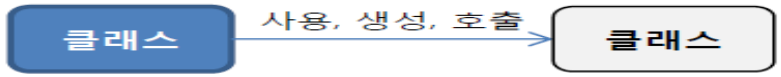
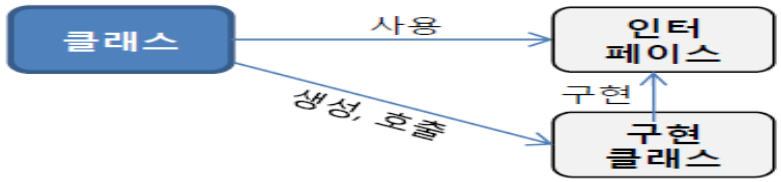
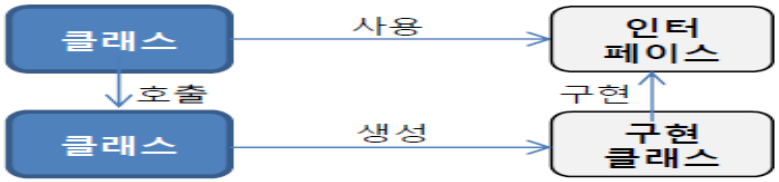
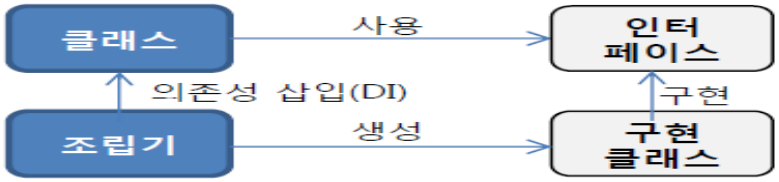
[2] DI(Dependency Injection)
- 의존성 주입. 각 계층 사이, 각 클래스 사이에 필요로 하는 의존 관계를 컨테이너가 자동으로 연결해 주는 것
- 각 클래스 사이의 의존 관계를 빈 설정(Bean Definition) 정보를 바탕으로 컨테이너가 자동으로 연결해 주는 것

- DL 사용 시 컨테이너 종속성이 증가하여, 이를 줄이기 위하여 DI를 사용
- 종류 : Setter Injection, Constructor Injection, Method Injection



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 구현클래스로 분리 구현클래스 교체가 용이하여 다양한 형태로 변화가 가능하지만 구현클래스 교체시 호출 클래스의 소스를 수정
팩토리패턴을 이용한 클래스 호출		팩토리방식은 팩토리가 구현클래스를 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

1. IOC

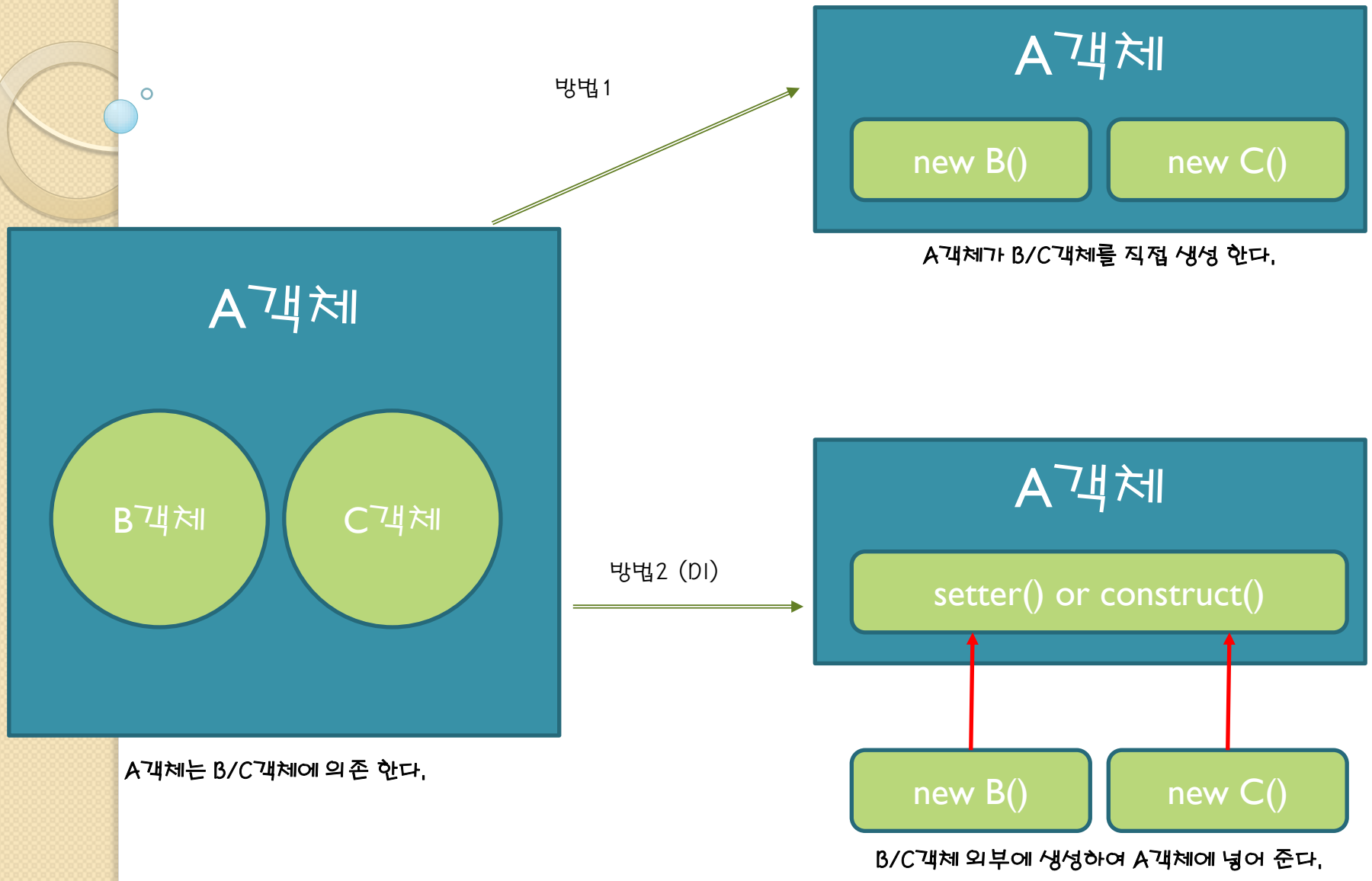
5) DI(Dependency Injection)의 종류

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

DI(Dependency Injection)의 종류



부품을 생성하고 조립하는 라이브러리 집합체.

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>낙시</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;
}
```

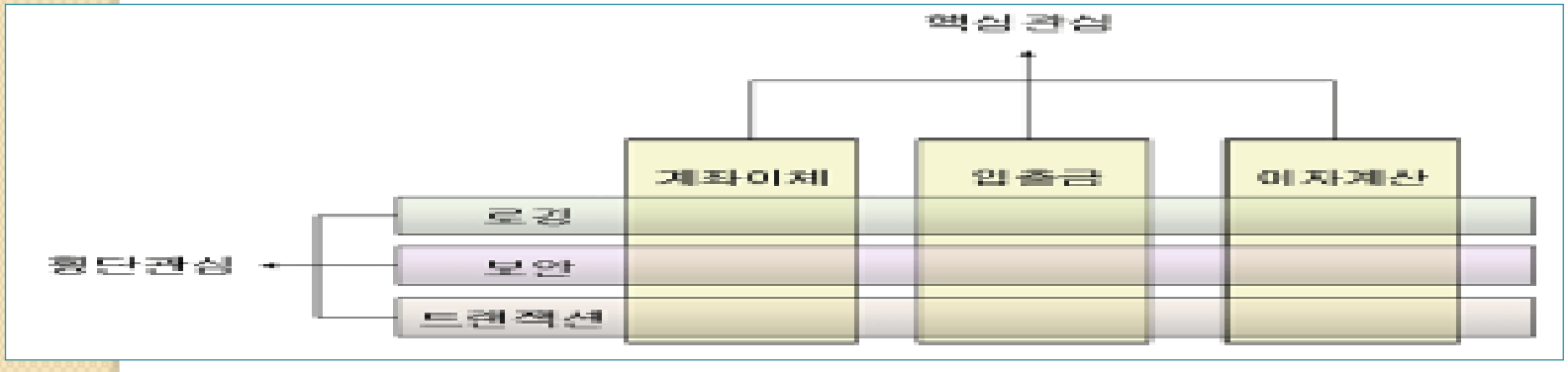
2. AOP(Aspect Oriented Programming)

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. 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. AOP(Aspect Oriented Programming)

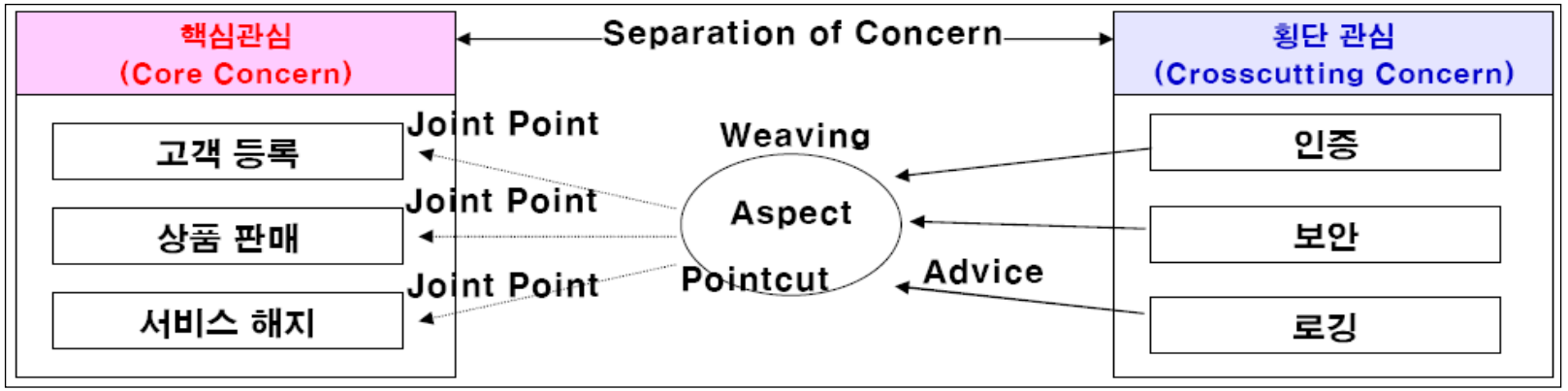
3. AOP의 특징

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. 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. 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
횡단 관심 (Cross-cutting)	- 핵심 관심에 공통적으로 적용되는 부가적인 요구사항 - 보안, 인증, 로그작성, 정책 적용 등 - Cross-cutting Concern
Joint Point	- **횡단 관심의 기능이 삽입되어 실행될 수 있는 프로그램 내의 실행될 위치 혹은 호출 Event** - 관심사가 주 프로그램의 어디에 횡단할 것인지를 나타내는 위치 - 생성자의 호출, 메소드의 호출, 오브젝트 필드에 대한 접근 등이 대표적인 joint point. - before: call (public void update*(. . .))
Point-Cut	- **관심사를 구현한 코드에 끼워 넣을수 있는 프로그램의 Event** - Execution Event , Initialization Events - 어느 Joint Point를 사용할 것인지를 결정하는 선택 기능 - AOP언어에게 언제 조인 포인트가 매치될 것인지 알려주는 구문
Advice	- **관심사를 구현하는 코드** - Point-cut에 의해 매칭된 joint point에 실행할 작업 - BEFORE, AROUND, AFTER의 실행 위치 지정
Aspect	- 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. 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. 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. 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. 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. AOP(Aspect Oriented Programming)

5. Spring AOP 구현 예시

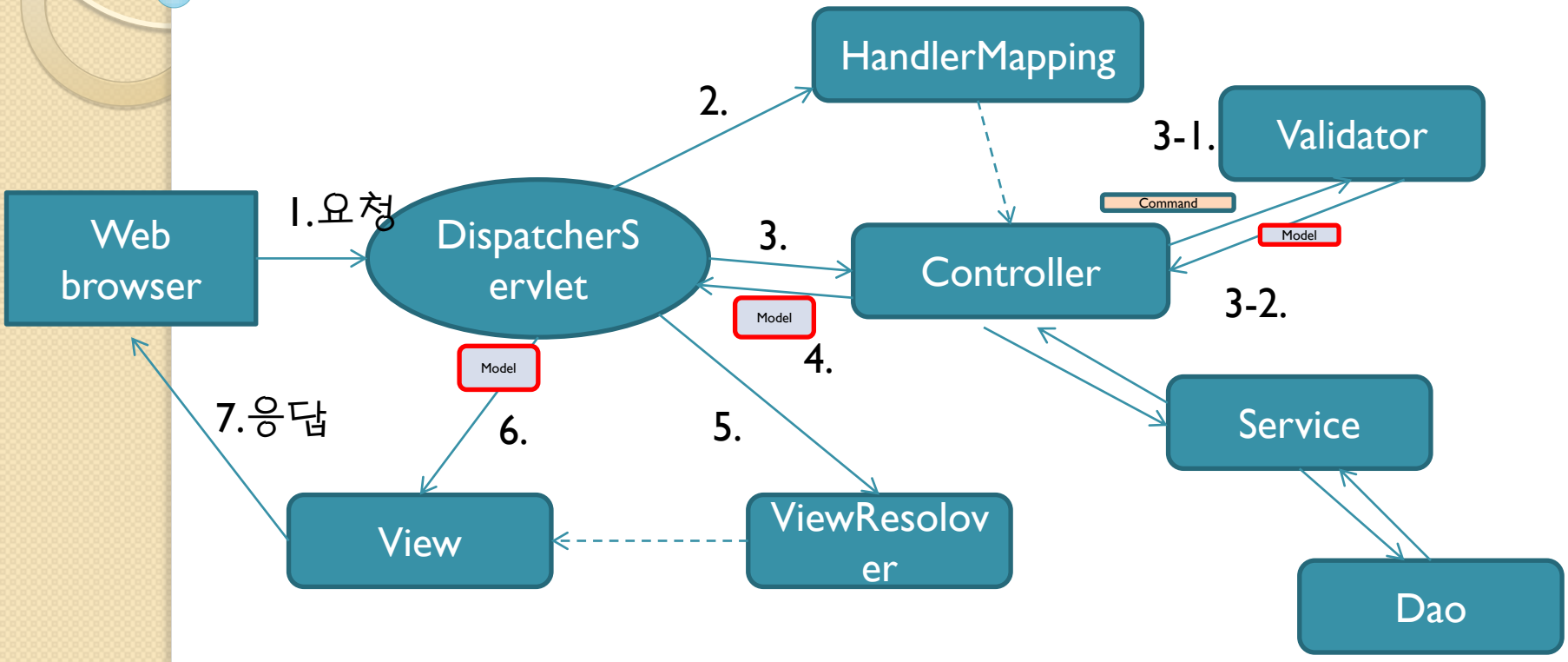
3) Annotation 적용 구현

```
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

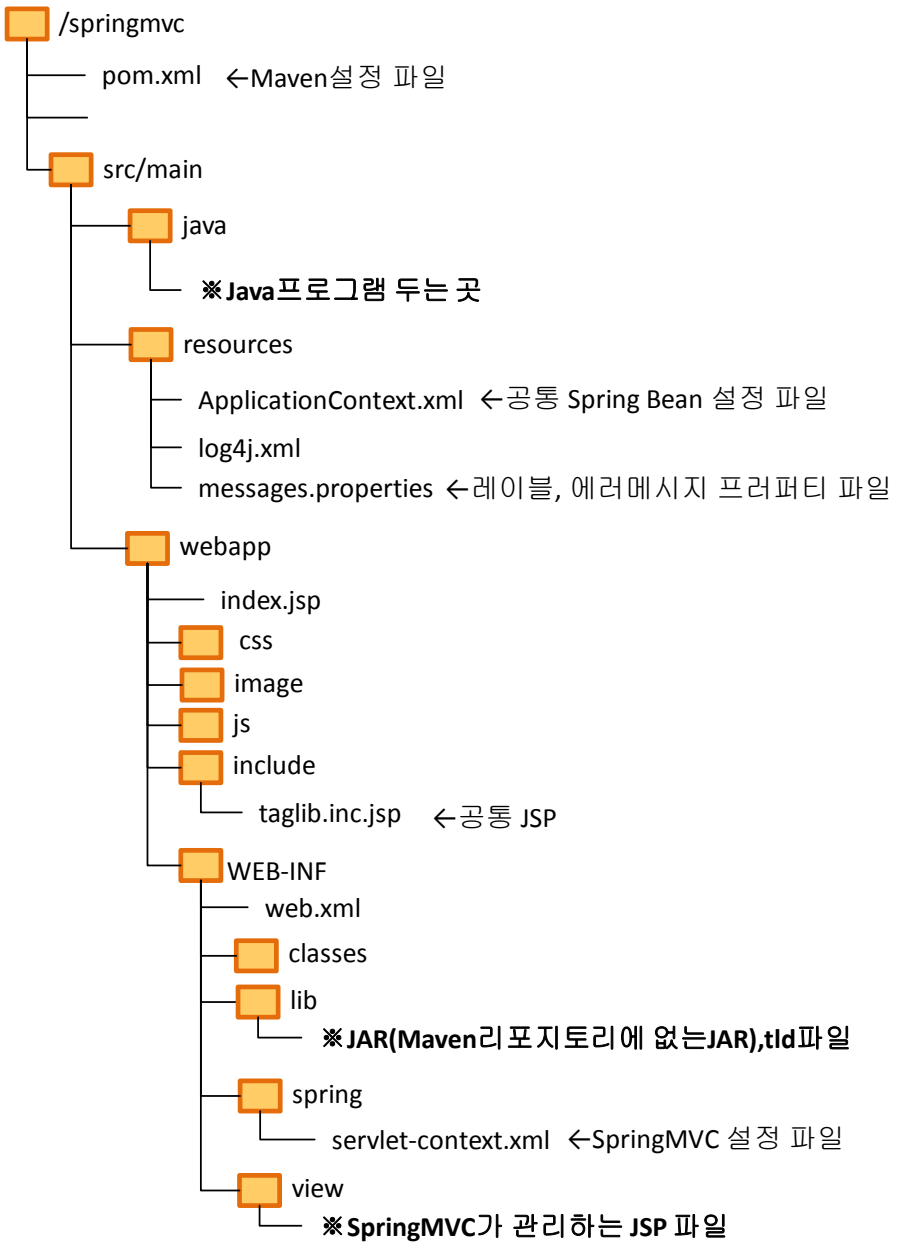
3. Spring MVC

1) MVC 흐름



3. Spring MVC

2) 파일구성



3. Spring MVC

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. spring ORM

1. OR-Mapping의 개요

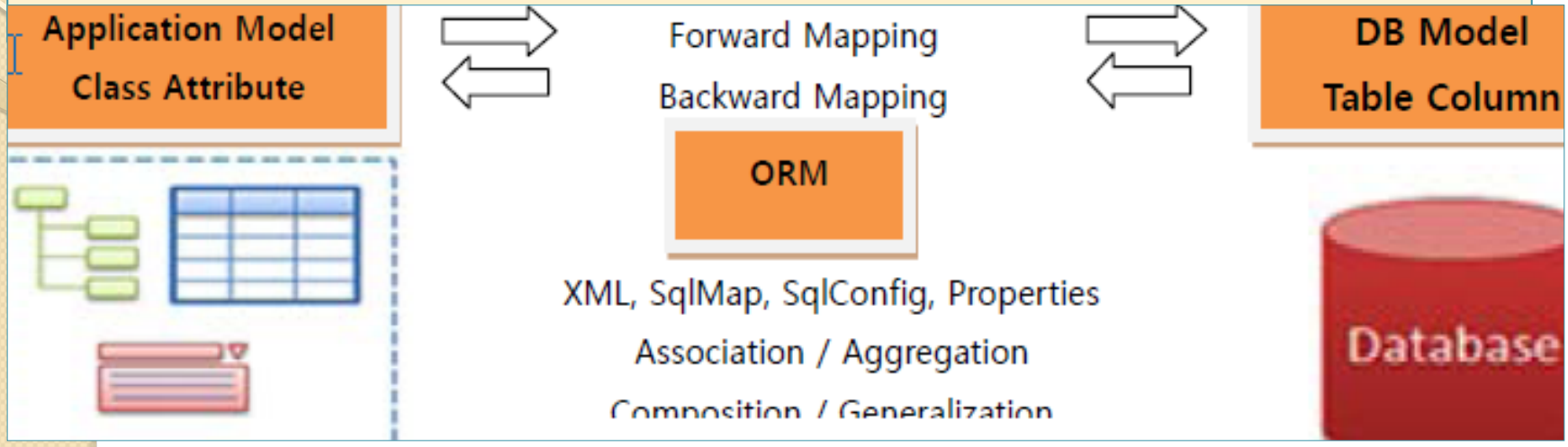
가. OR-Mapping의 정의
- **OOP 프로그래밍시 설계할 클래스들과 데이터저장소로 이용될 RDBMS의 Table 간의 Mapping**
- 데이터베이스 연계처리를 위하여 기존 SQL에 의존하는 것이 아니라 직접 테이블의 컬럼을 Java Class에 매핑하거나 XML형태의 SQL을 실행하여 처리를 수행하는 Persistence Layer를 담당하는 Framework 개발 모델

나. OR-Mapping의 등장배경
- OOP에 있어서 관계형 데이터베이스와의 연계성을 분명히 하자는 의도

다. OR-Mapping의필요성
- 초기 OO전문가들은 Object와 관계형 DB간의 심각한 구조적 불일치를 깨고, OODB를 창안
- 그러나 대부분의 프로젝트에서 OODB가 RDB만큼 안정성이 보장되지않아, Risk가 존재하는 OODB가 확산 되지 못함

4. spring ORM

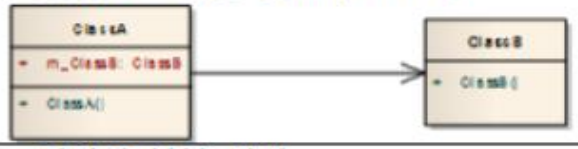
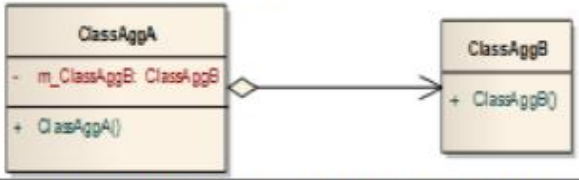
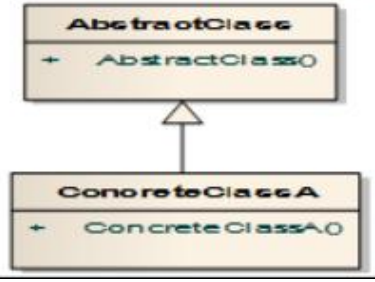
2. 객체지향 클래스와 데이터베이스 테이블간 매핑 관계
가. 객체지향 설계와 그에 맞는 DB 설계를 요구하는 OR Mapping



- 객체지향방법론으로 설계된 클래스를 데이터 저장소로 이용할 수 있는 Database Table로 매핑하는 절차나 방법론
- Database 연계 위해 기존 SQL에 의존하지 않고, 직접 테이블 컬럼을 자바 클래스에 매핑하거나 XML 선언을 통한 SQL 처리를 지원하는 방법론

4. spring ORM

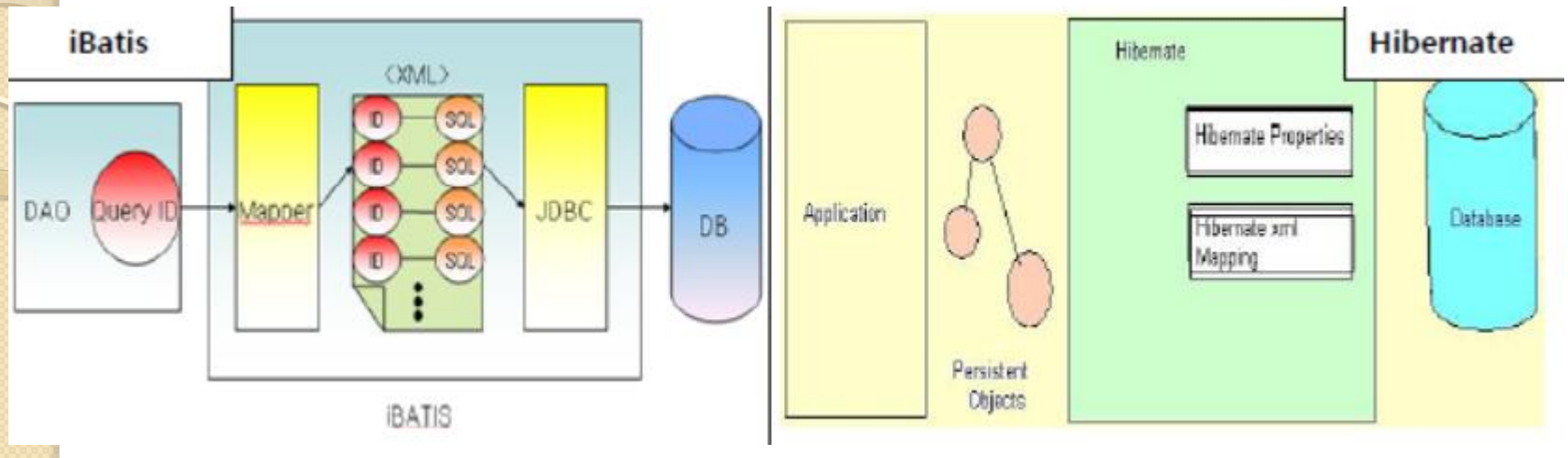
3. 클래스 다이어그램과 테이블간 매핑 관계

관계	설명	매핑 방안
Association (연관)	– 클래스간 가장 일반적인 관계 	– 1:1 관계(접근 빈도수가 많은 쪽으로 상대방 PK가 FK로 등록) – 1:N 관계(N쪽으로 1쪽의 PK를 FK로 매핑) – N:M 관계(새로운 관계 테이블을 추가)
Aggregation (집합)	– 전체와 부분 관계 	– 참고하는 테이블에서 단순 FK로 참조 – 비식별관계(약한 의존성)
Composition (복합연관)	– 동일 생명주기를 가진 전체와 부분 	– Delete Cascade 제약 추가 – 식별관계(강한 의존성)
Generalization (상속)	– 상속을 나타내는 일반화 관계 	– 1안 : 슈퍼 클래스와 서브 클래스를 별도의 테이블로 매핑 – 2안 : 슈퍼 클래스가 서브 클래스의 모든 속성을 포함한 단일 테이블로 매핑 – 3안 : 서브 클래스들이 슈퍼 클래스 속성을 상속받아 서브 클래스들을 테이블로 매핑

- OR Mapper에서 클래스는 테이블에, 클래스의 Attribute는 테이블의 Column에, Class Relationship은 테이블간 Relation에 매핑

4. spring ORM

4. XML에 임베디드된 myBatis와 Full-ORM 프레임워크 Hibernate의 비교



1) myBatis : - 소스 코드 외부에 정의된 SqlMapConfig.xml, SqlMap.xml 파일 정보를 기반으로 생성된 Mapped Statement를 이용하여 SQL과 객체간의 매핑 기능을 제공
- 개발자가 지정한 SQL, 저장프로시저 그리고 몇가지 고급 매핑을 지원하는 퍼시스턴스 프레임워크

2) Hibernate : 자바 클래스 객체와 데이터베이스 테이블 객체간 매핑을 통한 자동화된 질의 수행

4. spring ORM

5. myBatis와 Hibernate의 세부특징 비교

구분	iBatis	Hibernate
공통점	– XML 템플릿 등과 같은 표준 패턴을 이용하여 매핑 – JDBC와 같은 코드 사용시 보다 훨씬 간단한 구현 및 변경사항 적용이 가능 – 사용자에 의한 리소스 관리 및 코드의 중복 개발 감소 – 오픈 소스 기반의 ORM 프레임워크로 자유로운 이용과 배포 허용	
기반 사상	– SQL Mapping(Partial ORM)	– OR Mapping(Full ORM)
매핑 특징	– SQL 문을 활용한 객체 Mapping – 개발자가 직접 객체지향 관점에서 매핑 – 객체 모델과 데이터 모델 사이 매핑에 아무런 제약 사항이 없음	– 자바 클래스 객체와 테이블을 Mapping – 매핑 정보의 수정만으로 변경사항 적용 – SQL Mapper 보다 다양한 작업 수행
활용 특징	– 응답 지연 시간이 짧음 – 사용자 학습 곡선이 작음 – SQL 지식이 높아야 함 – 유연성이 우수	– 응답 지연 시간이 김(쿼리 자동생성) – 사용자 학습 곡선이 큼 – SQL 지식이 별로 필요 없음 – 유연성이 부족함
적용 방법	– 자바 객체를 SQL 문장에 매핑 – 자바 코드에서 SQL 부분을 제거하고 XML에 임베디드된 SQL 활용 – SQL 문장은 개발자에 의해 작성됨	– 자바 객제를 테이블 Row에 동기화 – 모든 SQL 문은 프레임워크에서 생성하고 실행하는 방식 – SQL 작업 필요시 HSQL 통해 이뤄짐
적용/조건	– SQL 문을 통한 튜닝이나 최적화 필요시 – 부적절한 DB 설계 상황 – 3rd Party 데이터베이스에 접근하는 경우 – 여러 개의 테이블과 하나의 자바 클래스 매핑되는 경우	– SQL Mapper 보다 효율적인 매핑 – 새로운 프로젝트가 시작된 상태 – 객체 모델과 데이터베이스 디자인이 미완성인 상태 – 하나의 테이블과 하나의 자바 클래스가 매핑되는 경우
장점/단점	– 복잡한 데이터 전송 환경에 효과적 – SQL의 장점 활용에 비교 우위	– OR Mapper에 의한 자동화 지원 – 간단한 CRUD 어플리케이션 테이블–클래스 매핑 사용시 단순성과 성능 비교 우위
유사 제품	– Oracle SQLJ, Pro*C embedded SQL – 대부분의 임베디드 SQL 시스템	– TopLink, JDO, ADO.NET
환경 설정	– SqlMapConfig.xml : DataSource, Data Mapper 및 Thread 관리 등의 설정 정보 – SqlMap.xml : 많은 캐시 모델, 파라미터 맵, Results Maps, Statements 정보 포함	– hibernate.properties : 전체 구성 지정 – hibernate.cfg.xml : hibernate.properties 파일에 대한 대용 혹은 중복정의에 활용

4. MyBatis setting

1. Root.context

```
<!-- //Oracle 접속부분 -->
<context:property-placeholder
location="classpath:mybatis/jdbc.properties"/>
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
destroy-method="close">
<property name="driverClass" value="${jdbc.driverClassName}" />
<property name="jdbcUrl" value="${jdbc.url}" />
<property name="user" value="${jdbc.username}" />
<property name="password" value="${jdbc.password}" />
<property name="maxPoolSize" value="${jdbc.maxPoolSize}" />
</bean>
<!-- 스프링 jdbc 즉 스프링으로 oracle 디비 연결 -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
<property name="dataSource" ref="dataSource" />
<property name="configLocation" value="classpath:mybatis/configuration.xml" />
</bean>
<bean id="session" class="org.mybatis.spring.SqlSessionTemplate">
<constructor-arg index="0" ref="sqlSessionFactory" />
</bean>
<!-- transactionmanager 선언 -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
<property name="dataSource" ref="dataSource" />
</bean>
```

4. MyBatis setting

2. configuration.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config
3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
<typeAliases>
<typeAlias alias="Emp" type="oracle.java.myBatis3.model.Emp" />
<typeAlias alias="Dept" type="oracle.java.myBatis3.model.Dept" />
<typeAlias alias="EmpDept"
type="oracle.java.myBatis3.model.EmpDept" />
</typeAliases>
<mappers>
<mapper resource="mybatis/Emp.xml" />
<mapper resource="mybatis/Dept.xml" />
<mapper resource="mybatis/EmpDept.xml" />
</mappers>
</configuration>
```

4. MyBatis setting

3. EmpDept.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="EmpDept">
  <!-- Use type aliases to avoid typing the full classname every time. -->
  <resultMap id="EmpDeptResult" type="EmpDept">
    <result property="empno" column="empno"/>
    <result property="ename" column="ename"/>
    <result property="job" column="job"/>
    <result property="mgr" column="mgr"/>
    <result property="hiredate" column="hiredate"/>
    <result property="sal" column="sal"/>
    <result property="comm" column="comm"/>
    <result property="deptno" column="deptno"/>
    <result property="dname" column="dname"/>
    <result property="loc" column="loc"/>
  </resultMap>
   <select id="listEmp" parameterType="EmpDept"
   resultMap="EmpDeptResult">
        select e.empno, e.ename, e.job, d.dname, d.loc
    from emp e, dept d where e.deptno=d.deptno order by empno
  </select>
</mapper>
```

4. MyBatis setting

4. jdbc.properties

```
jdbc.driverClassName=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@127.0.0.1:1521:xe
jdbc.username=scott
jdbc.password=tiger
jdbc.maxPoolSize=20
```

4. MyBatis setting

6. SQL Map XML 파일
   1) MyBatis 의 가장 큰 장점은 매핑된 구문
   2) SQL Map XML 파일은 상대적으로 간단하다.
   3) JDBC코드와 비교하면 아마도 95% 이상 코드수가 감소
   4) MyBatis 는 SQL 을 작성하는데 집중하도록 만들어 짐

cache – 해당 명명공간을 위한 캐시 설정
cache-ref – 다른 명명공간의 캐시 설정에 대한 참조
resultMap – 데이터베이스 결과데이터를 객체에 로드하는 방법을 정의하는 요소
parameterMap – 비권장됨! 예전에 파라미터를 매핑하기 위해 사용되었으나 현재는 사용하지 않음
sql – 다른 구문에서 재사용하기 위한 SQL 조각
insert – 매핑된 INSERT 구문
update – 매핑된 UPDATE 구문.
delete – 매핑된 DELEETE 구문.
select – 매핑된 SELECT 구문.

4. MyBatis setting

6. SQL Map XML 파일
   1) MyBatis 의 가장 큰 장점은 매핑된 구문
   2) SQL Map XML 파일은 상대적으로 간단하다.
   3) JDBC코드와 비교하면 아마도 95% 이상 코드수가 감소
   4) MyBatis 는 SQL 을 작성하는데 집중하도록 만들어 짐

cache – 해당 명명공간을 위한 캐시 설정
cache-ref – 다른 명명공간의 캐시 설정에 대한 참조
resultMap – 데이터베이스 결과데이터를 객체에 로드하는 방법을 정의하는 요소
parameterMap – 비권장됨! 예전에 파라미터를 매핑하기 위해 사용되었으나 현재는 사용하지 않음
sql – 다른 구문에서 재사용하기 위한 SQL 조각
insert – 매핑된 INSERT 구문
update – 매핑된 UPDATE 구문.
delete – 매핑된 DELEETE 구문.
select – 매핑된 SELECT 구문.

4. MyBatis setting

7. **Select**

select 구문은 MyBatis 에서 가장 흔히 사용할 요소이다. 데이터베이스에서 데이터를 가져온다. 아마도 대부분의 애플리케이션은 데이터를 수정하기보다는 조회하는 기능을 많이 가진다. 그래서 MyBatis는 데이터를 조회하고 그 결과를 매핑하는데 집중하고 있다. Select 는 다음 예처럼 단순한 경우에는 단순하게 설정된다.

```
<select id="selectPerson" parameterType="int" resultType="hashmap">
SELECT * FROM PERSON WHERE ID = #{id}
</select>
```

- 이 구문의 이름은 selectPerson 이고 int 타입의 파라미터를 가진다.
  그리고 결과 데이터는 HashMap에 저장된다. 파라미터 표기법을 보자.

#{id}
이 표기법은 MyBatis 에게 PreparedStatement 파라미터를 만들도록 지시

4. MyBatis setting

7. **Select**

2] select 요소는 각각의 구문이 처리하는 방식에 대해 좀더 세부적으로 설정하도록 많은 속성을 설정

```
<select
id="selectPerson"
parameterType="int"
parameterMap="deprecated"
resultType="hashmap"
resultMap="personResultMap"
flushCache="false"
useCache="true"
timeout="10000"
fetchSize="256"
statementType="PREPARED"
resultSetType="FORWARD_ONLY"
>
```

속성	설명
id	구문을 찾기 위해 사용될 수 있는 명명공간내 유일한 구분자
parameterType	구문에 전달될 파라미터의 패키지 경로를 포함한 전체 클래스명이나 별칭
parameterMap	외부 parameterMap 을 찾기 위한 비권장된 접근방법. 인라인 파라미터 매핑과 parameterType 을 대신 사용하라.

4. MyBatis setting

7. Select(계속)

속성	설명
resultType	이 구문에 의해 리턴되는 기대타입의 패키지 경로를 포함한 전체 클래스명이나 별칭. collection 이 경우, collection 타입 자체가 아닌 collection 이 포함된 타입이 될 수 있다. resultType 이나 resultMap 을 사용하라.
resultMap	외부 resultMap 의 참조명. 결과맵은 MyBatis 의 가장 강력한 기능이다. resultType 이나 resultMap 을 사용하라.
flushCache	이 값을 true 로 셋팅하면, 구문이 호출될때마다 캐시가 지원질것이다(flush). 디폴트는 false 이다.
useCache	이 값을 true 로 셋팅하면, 구문의 결과가 캐시될 것이다. 디폴트는 true 이다.
timeout	예외가 던져지기 전에 데이터베이스의 요청 결과를 기다리는 최대시간을 설정한다. 디폴트는 셋팅하지 않는 것이고 드라이버에 따라 다소 지원되지 않을 수 있다.
fetchSize	지정된 수만큼의 결과를 리턴하도록 하는 드라이버 힌트 형태의 값이다. 디폴트는 셋팅하지 않는 것이고 드라이버에 따라 다소 지원되지 않을 수 있다.
statementType	STATEMENT, PREPARED 또는 CALLABLE 중 하나를 선택할 수 있다. MyBatis 에게 Statement, PreparedStatement 또는 CallableStatement 를 사용하게 한다. 디폴트는 PREPARED 이다.
resultSetType	FORWARD_ONLY|SCROLL_SENSITIVE|SCROLL_INSENSITIVE 중 하나를 선택할 수 있다. 디폴트는 셋팅하지 않는 것이고 드라이버에 다라 다소 지원되지 않을 수 있다.

4. MyBatis setting

8. **Insert, Update, Delete**

2] select 요소는 각각의 구문이 처리하는 방식에 대해 좀더 세부적으로 설정하도록 많은 속성을 설정

```
<insert
id="insertAuthor"
parameterType="domain.blog.Author"
flushCache="true"
statementType="PREPARED"
keyProperty=""
keyColumn=""
useGeneratedKeys=""
timeout="20000">

<update
id="insertAuthor"
parameterType="domain.blog.Author"
flushCache="true"
statementType="PREPARED"
timeout="20000">

<delete
id="insertAuthor"
parameterType="domain.blog.Author"
flushCache="true"
statementType="PREPARED"
timeout="20000">
```

4. MyBatis setting

8. Insert, Update, Delete (계속)

속성	설명
Id	구문을 찾기 위해 사용될 수 있는 명명공간내 유일한 구분자
parameterType	구문에 전달될 파라미터의 패키지 경로를 포함한 전체 클래스명이나 별칭
parameterMap	외부 parameterMap 을 찾기 위한 비권장된 접근방법. 인라인 파라미터 매핑과 parameterType 을 대신 사용하라.
flushCache	이 값을 true 로 셋팅하면, 구문이 호출될때마다 캐시가 지원질것이다(flush). 디폴트는 false 이다.
Timeout	예외가 던져지기 전에 데이터베이스의 요청 결과를 기다리는 최대시간을 설정한다. 디폴트는 셋팅하지 않는 것이고 드라이버에 따라 다소 지원되지 않을 수 있다.
statementType	STATEMENT, PREPARED 또는 CALLABLE 중 하나를 선택할 수 있다. MyBatis 에게 Statement, PreparedStatement 또는 CallableStatement 를 사용하게 한다. 디폴트는 PREPARED 이다.
useGeneratedKeys	(입력(insert)에만 적용) 데이터베이스에서 내부적으로 생성한 키(예를 들어, MySQL 또는 SQL Server 와 같은 RDBMS 의 자동 증가 필드)를 받는 JDBC getGeneratedKeys 메서드를 사용하도록 설정하다. 디폴트는 false 이다.
keyProperty	(입력(insert)에만 적용) getGeneratedKeys 메서드나 insert 구문의 selectKey 하위 요소에 의해 리턴된 키를 셋팅할 프로퍼티를 지정. 디폴트는 셋팅하지 않는 것이다.
keyColumn	(입력(insert)에만 적용) 생성키를 가진 테이블의 칼럼명을 셋팅. 키 칼럼이 테이블이 첫번째 칼럼이 아닌 데이터베이스(PostgreSQL 처럼)에서만 필요하다.

4. MyBatis setting

8. **Insert, Update, Delete 예시문 (계속)**

```
<insert id="insertAuthor“ parameterType="domain.blog.Author">
        insert into Author (id,username,password,email,bio)
        values (#{id},#{username},#{password},#{email},#{bio})
</insert>
<update id="updateAuthor" parameterType="domain.blog.Author">
        update Author set
        username = #{username},
        password = #{password},
        email = #{email},
        bio = #{bio}
        where id = #{id}
</update>
<delete id="deleteAuthor” parameterType="int">
                delete from Author where id = #{id}
</delete>
```

5. 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>
```

5. 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>
```

5. 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>
```