

SpringFramework Tutorial

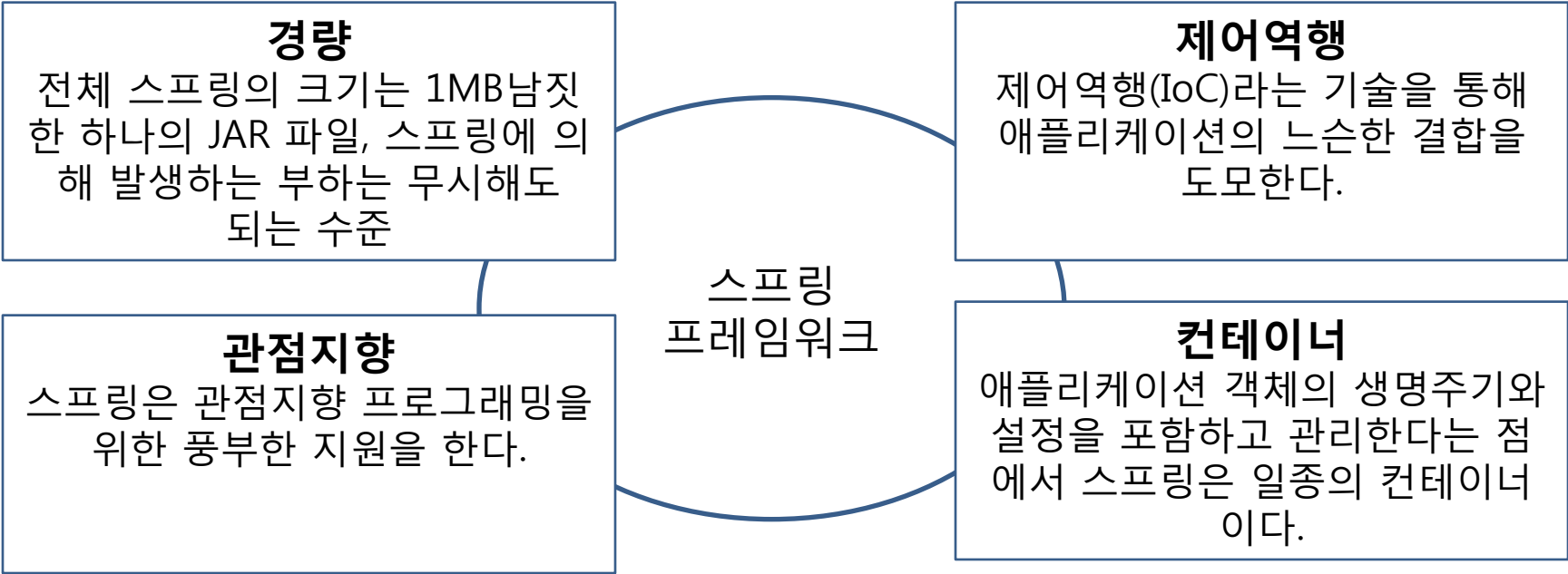
Think about User eXperience for your successful Business

■ Spring Framework 개요

- 경량,관점지향,제어역행,컨테이너에 대한 개념

- Rod Johnson이 만든 오픈소스 프레임워크
- 복잡한 엔터프라이즈 애플리케이션 개발을 겨냥
- 단순성,테스트 용이성, 느슨한 결합성의 측면에서 스프링의 이점을 얻을 수 있음

스프링은 경량의 제어 역행과 관점지향 컨테이너 프레임워크이다.



Spring Framework 개요

- 스프링 모듈

Core

프레임워크의 가장 기본적인 부분이고 IoC와 의존성 삽입(Dependency Injection-DI)기능을 제공한다

DAO

끔찍한 JDBC코딩과 데이터베이스 업체 특정 에러코드의 파싱을 할 필요를 제거하는 JDBC추상화 레이어를 제공한다. 또한 [JDBC](#)패키지는 특정 인터페이스를 구현하는 클래스를 위해서 뿐 아니라 *당신의 모든 POJOs*를 위해서도 선언적인 트랜잭션 관리만큼 프로그램에 따른 방식으로 할수 있는 방법을 제공한다.

ORM

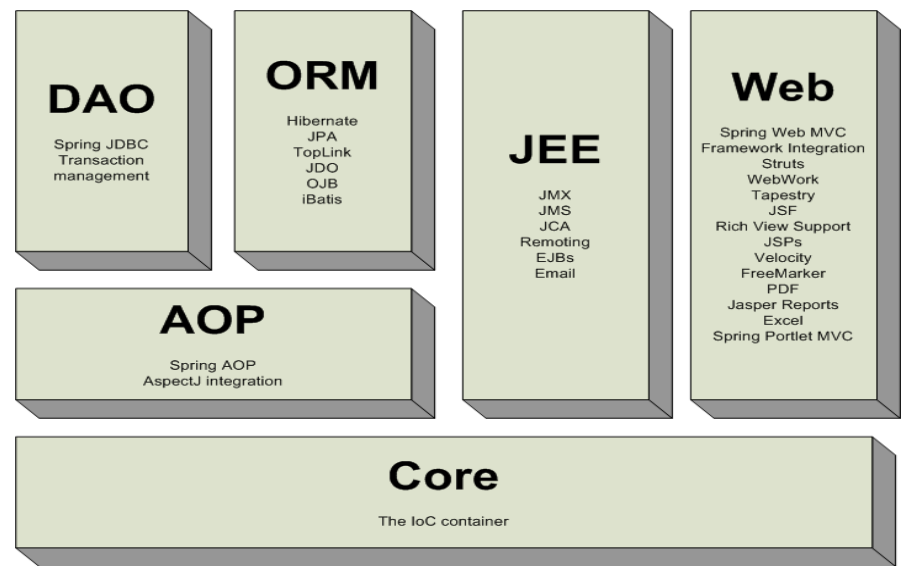
[JDO](#), [Hibernate](#), 그리고 [iBatis](#)를 포함하는 인기있는 객체-관계 맵핑 API를 위한 통합 레이어를 제공한다. ORM패키지는 사용하여 당신은 앞에서 언급된 간단한 선언적인 트랜잭션 관리와 같은 Spring이 제공하는 다른 모든 기능을 사용해서 혼합하여 모든 O/R매퍼를 사용할수 있다

AOP

당신이 정의하는것을 허용하는 *AOP 제휴* 호환 aspect-지향 프로그래밍 구현물을 제공한다. 예를 들어 코드를 명백하게 분리하기 위한 메소드-인터셉터와 pointcut은 논리적으로 구별되어야 할 기능을 구현한다. 소스레벨 메타데이터 기능을 사용하여 당신은 .NET속성과 다소 비슷한 모든 종류의 행위적 정보를 당신의코드로 결합한다.

Web

멀티파트 파일업로드기능, 서블릿 리스너를 사용한 IoC컨테이너의 초기화 그리고 웹-기반 애플리케이션 컨텍스트와같은 기본적인 웹-기반 통합 기능들을 제공한다. WebWork나 Struts와 함께 Spring을 사용할때 이것은 그것들과 통합할 패키지이다.



Spring 프레임워크 개요

MVC

웹 애플리케이션을 위한 Model-View-Controller(MVC)구현물을 제공한다. Spring의 MVC구현물은 어떤 오래된 구현물이 아니다. 이것은 도메인 모델 코드와 웹폼(Web forms)사이의 *분명한* 구분을 제공하고 유효성체크와 같은 Spring프레임워크의 다른 모든 기능을 사용하도록 당신에게 허용한다

■ **HelloService – com/education/edu01**

– 스프링을 한번 경험해 보는 시간으로 HelloService를 작성

- HelloService
- HelloServiceImpl
- edu01.xml
- HelloServiceTest

■ 이전개발과의 차이점

- HelloService를 통하여 기존개발 방식과의 차이점을 알아 본다.

기존의 방식

```
CustomerManager customerManager = new CustomerManager();  
CustomerDao dao = new CustomerDao();  
customerManagere.setDao(dao);  
Customer customer = customerManager.getCustomer(suninet@naver.com);
```

Spring의 방식

```
CustomerManager customerManager = new CustomerManager();  
CustomerDao dao = new CustomerDao();  
customerManagere.setDao(dao);  
Customer customer = customerManager.getCustomer(suninet@naver.com);
```

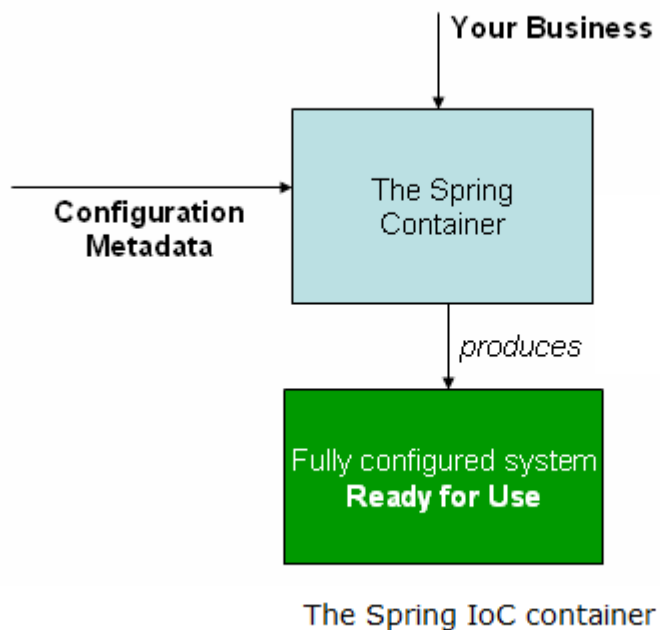


Spring 컨테이너가 처리 ,컨테이너가 처리할 수 있는 메커니즘이 필요, XML 파일에 정의함

■ The container

- Spring IoC container 에 대한 이해

애플리케이션 오브젝트들을 소싱하거나 인스턴스화 하는 책임, BeanFactory 인터페이스는 IoC container의 중심에 있음. XmlBeanFactory 은 XML 설정 메타데이터를 취하여 그것을 이용하여 완전히 설정된 시스템 또는 어플리케이션을 생성.



■ BeanFactory

- BeanFactory의 객체 생성

② 내부적으로 객체를 생성

ServiceObject



BeanFactory

① Bean definitions을 입력으로 사용



Bean
Definitions
File

③ 객체가 외부에서 주입(Injection)



```
public class CustomerManager {  
    private CustomerDao customerDao;  
    public void setCustomerDao(CustomerDao dao) {  
        this.customerDao = dao;  
    }  
}
```

■ configuration metadata

- Bean 설정 방법

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions go here -->

</beans>
```


■ 컨테이너 초기화(instantiating a container)

- Spring IoC container를 직접 인스턴스화 하기

인스턴스화 하기

```
ApplicationContext context = new ClassPathXmlApplicationContext(  
    new String[] {"services.xml", "daos.xml"});
```

```
// an ApplicationContext is also a BeanFactory (via inheritance)  
BeanFactory factory = context;
```

빈을 얻어오기

```
MyBean myBean = (MyBean)factory.getBean("myBean");
```

■ Bean Definitions File 읽기

- 설정파일 읽어오기 방법

FileSystemResource

절대경로 /상대 경로 둘다 가능 함, \\\, / 둘다 가능함

```
BeanFactory factory =  
new XmlBeanFactory(new FileSystemResource("WebContent/WEB-INF/spring-config/hello.xml"));
```

ClassPathResource

classpath 내에 있는 리소스 검색시 사용

```
BeanFactory factory =  
new XmlBeanFactory(new  
ClassPathResource("edu/biz/helloworld/hello.xml"));
```

■ IoC(Inversion of Control)

- 마틴파울러 가 말하기를...

마틴 파울러는 2004년의 글에서 제어의 어떤 측면이 역행되는 것인지에 대한 의문을 제기했다. 그는 **의존하는 객체를 역행적으로 취득**하는 것이라는 결론을 내렸다. 그는 그와 같은 정의에 기초하여 제어 역행이라는 용어에 좀더 참신한 ‘의존성 주입(dependency injection)’이라는 이름을 지어줬다.

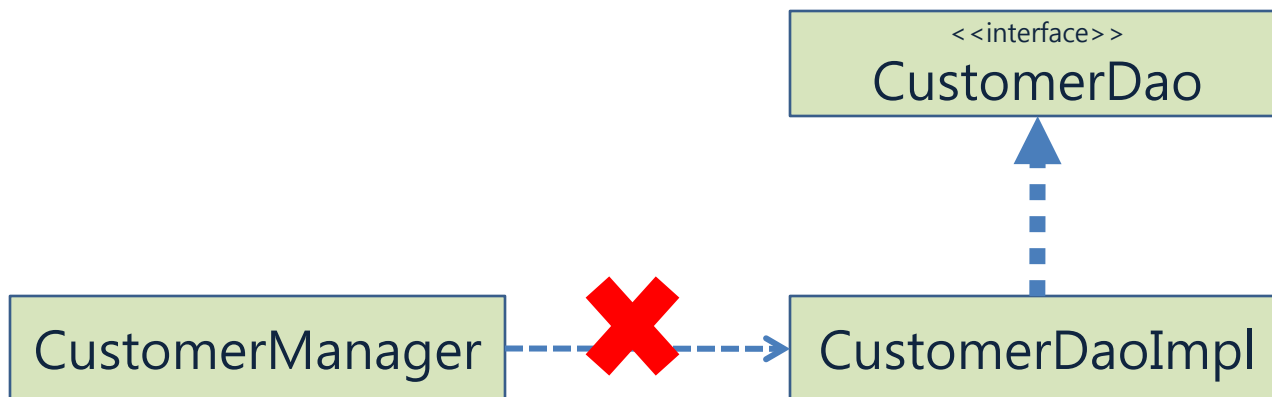
모든 어플리케이션은 비즈니스 로직을 수행하기 위해 서로 협업하는 둘 또는 그 이상의 클래스들로 이뤄진다. 전통적으로 각 객체는 **협업할 객체의 참조를 취득**해야 하는 책임이 있다. 이것이 의존성이다. 이는 결합도가 높으며 테스트하기 어려운 코드를 만들어 낸다.

IoC를 적용함으로써 객체들은 시스템 내의 각 객체를 조정하는 어떤 외부의 존재에 의해 생성 시점에서 의존성을 부여 받는다. 즉 의존성이 객체로 주입(inject)된다는 말이다. 따라서 IoC는 한 객체가 협업해야 하는 다른 객체의 참조를 취득하는 방법에 대한 책의의 역행이라는 의미를 갖는다.

■ Spring에서의 IoC(Inversion of Control)

- 스프링에서 컴포넌트간의 의존성에 대한 고찰

스프링에서는 어떤 컴포넌트가 다른 컴포넌트와의 연관관계를 관리할 책임이 없다. 그대신, 컨테이너에 의해 컴포넌트 간의 협업을 위한 참조가 주어진다.



```
public class CustomerManager {  
    private CustomerDao customerDao;  
  
    public CustomerManager() {  
        this.customerDao = new CustomerDaoImpl();  
    }  
}///~
```

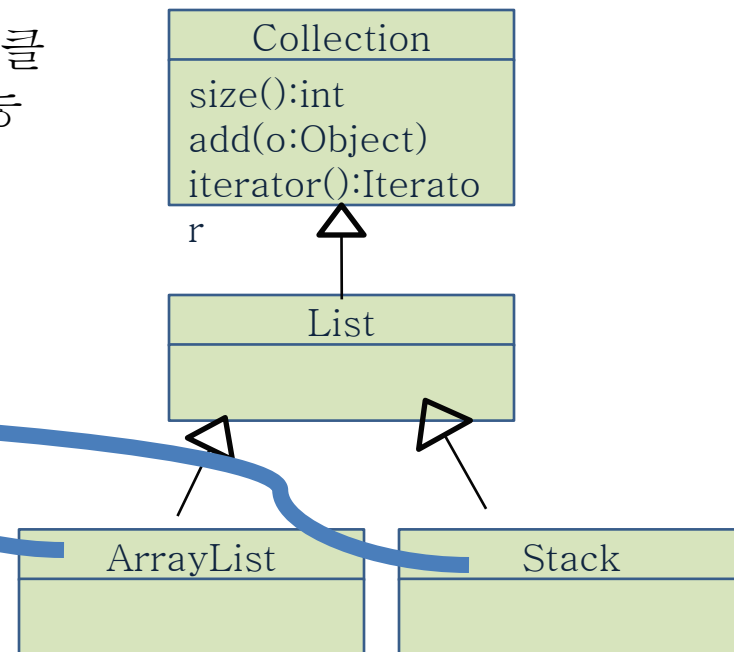
■ IoC는 왜 필요한가?

- interface 살펴보기 (1)

- 인터페이스를 사용하여 실제 구현된 객체를 알지 못함.
- 구현내용을 인터페이스의 뒤로 숨김으로써 클라이언트 클래스에 영향을 주지 않고도 실제 구현 클래스가 교체가능

Collection interface를 사용한 코드예

```
public void setOrder(Collection collection) {  
    ...  
    Iterator iter = collection.iterator();  
    while(iter.hasNext()) {  
        Object o = iter.next();  
        .....  
    }//while  
}//:
```



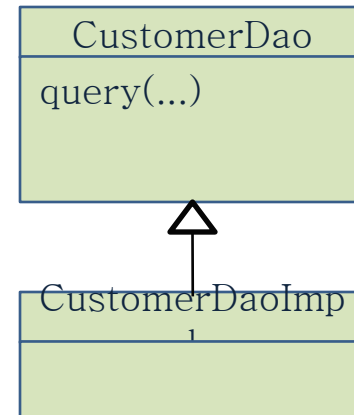
■ IoC는 왜 필요한가?

- interface 살펴보기 (2)

- 인터페이스를 작성하였으나 구상클래스를 직접 인스턴스화함.
- 구현내용을 인터페이스의 뒤로 숨기지 못하고 강하게 결합 (coupling) 되어 있음.

어플리케이션에서의 구현예

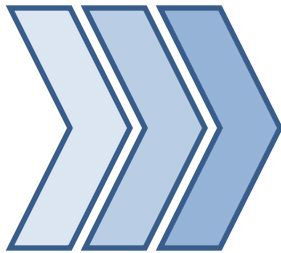
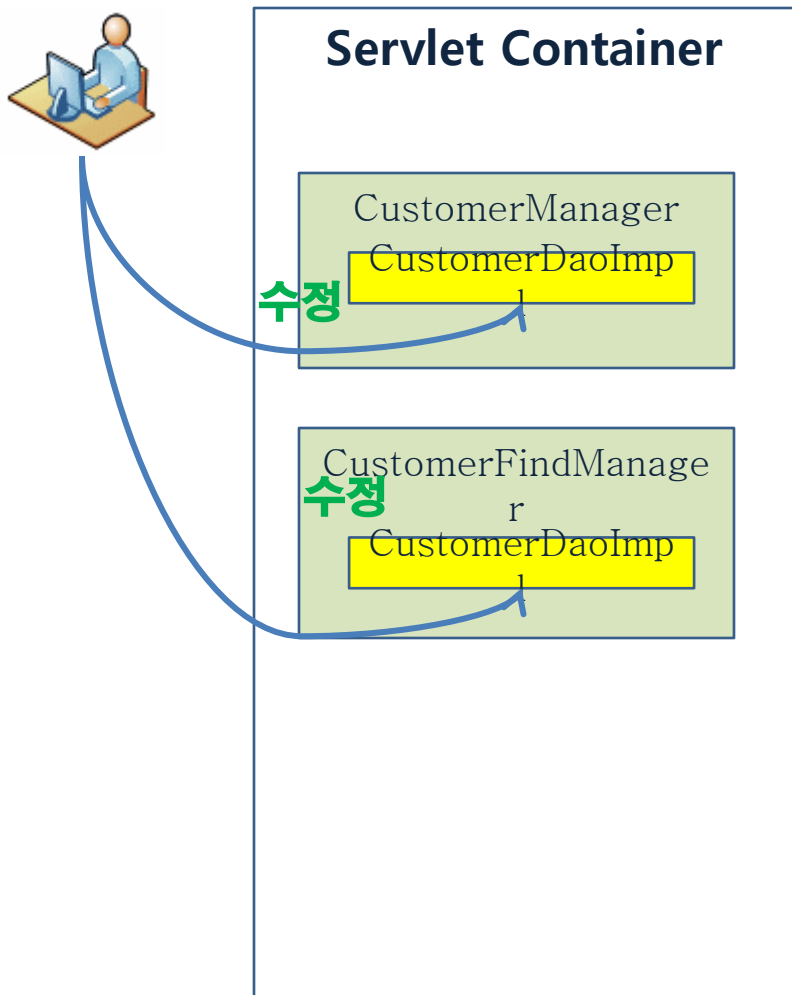
```
public class CustomerManager {  
    private CustomerDao customerDao;  
    public void CustomerManager() {  
        this.customerDao = new CustomerDaoImpl();  
    }  
    public List getCustomers(SearchOptions params) {  
        return this.customerDao.query(params);  
    }  
}
```



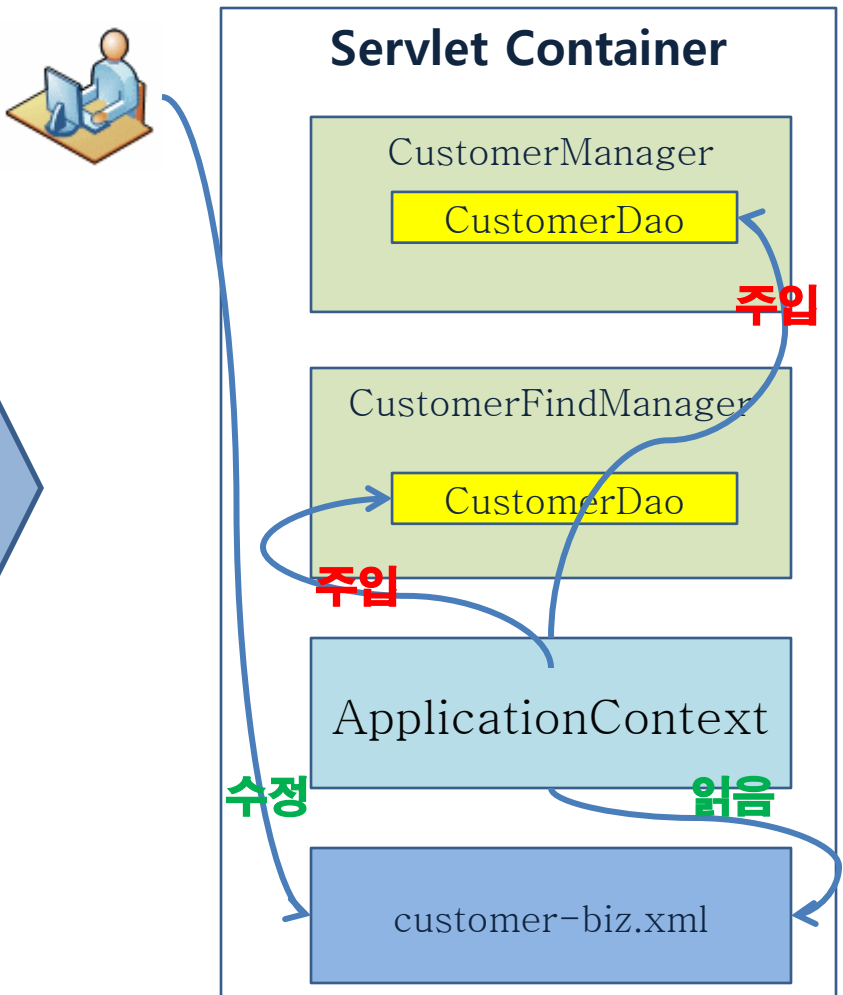
IoC는 왜 필요한가?

- IoC를 사용하지 않는 경우와 그렇지 않은 경우의 비교

IoC를 사용하지 않는 경우



IoC를 사용한 경우



■ 객체지향 설계원칙

- 유연한 객체지향적인 프로그램이 되기 위해서

설계원칙

- 자주 변경되는 구상클래스(Concrete class)에 의존하지 마라
- 어떤 클래스를 상속받아야 한다면, 기반 클래스를 추상 클래스로 만들어라
- 인터페이스를 만들어서 이 인터페이스에 의존하라
→ DIP(Dependency Inversion Principle)

나쁜 품질의 코드

- 경직성: 무엇이든 하나를 바꿀 때 마다 다른 것도 바꿔야 한다.
- 부서지기쉬움: 시스템에서 한 부분을 변경하면 그것과 전혀 상관없는 다른 부분이 오류 발생
- 부동성: 시스템을 여러 컴포넌트로 분해해서 재사용하기 힘들다
- 끈끈함: 개발환경이 편집-컴파일-테스트 순환을 한번도는데 시간이 엄청나게 길다.
- 쓸데없이 복잡함: 괜히 머리를 굴려서 짠 코드 구조가 굉장히 많다.
- 필요없는 반복: copy & paste
- 불투명함: 코드를 만든 의도에 대한 설명을 볼때 그 설명에 '표현이 고인다'라는 말이 잘어울림

■ **ApplicationContext**

- 더 많은 기능을 제공하는 ApplicationContext를 사용하자

기능

- 국제화 지원을 포함해 텍스트 메시지를 해석하는 수단 제공
- 이미지 등과 같은 자원을 로딩하는 범용적인 방법을 제공

ClassPathXmlApplicationContext

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(  
    "conf/appContext.xml");
```

FileSystemClassPathXmlApplicationContext

```
ApplicationContext ctx = new FileSystemClassPathXmlApplicationCont  
ext("conf/appContext.xml");
```

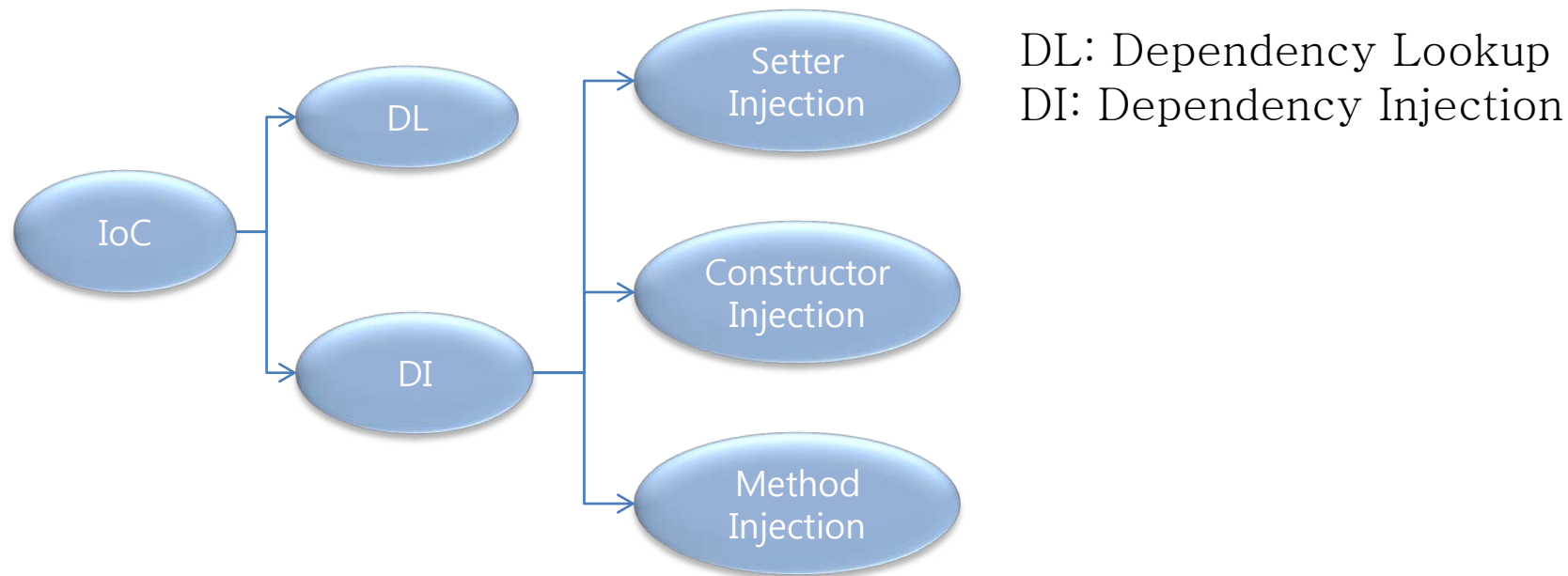
FileSystemXmlApplicationContext

```
ApplicationContext ctx = new FileSystemXmlApplicationContext(  
    "classpath:conf/appContext.xml");
```

■ IoC 분류체계

- IoC 컨테이너를 이용하여 빈을 주입하는 방법과 검색하는 방법에 대한 정의

분류체계



컨테이너 관리객체 저장소

서블릿 컨테이너는 서블릿 관리를 위해서 web.xml을 사용

EJB 컨테이너는 ejb-jar.xml에 설정되어 있는 정보들이 JNDI에 저장되어 관리된다.

Spring에서는 POJO를 관리하기 위해서 XML이나, properties 파일을 이용

■ Lookup Sample

- 저장소의 bean을 검색하여 사용

Sample>

```
String salesManagerName = "salesManager";  
ServletContext ctx= getServletContext();  
WebApplicationContext wc =  
WebApplicationContextUtils.getRequireWebApplicationContext(ctx);  
SalesManager sm = (SalesManager)wc.getBean(salesManagerName);
```

**저장소에 관리되고 있는 Bean을 개발자들이 직접 Lookup하여 사용하는 것을
Dependency Lookup**

**Dependency Injection은 이 같은 의존관계를 컨테이너가 자동적으로 연결시켜
주는 것을 말함.**

■ DI(Dependency Injection)

– Setter Injection

Java

```
public class SalesManager {  
    private SalesMan salesMan;  
    public void setSalesMan(SalesMan sm) {  
        this.salesMan = sm;  
    }  
}////~
```

XML

```
<beans>  
<bean id="salesMan" class="edu.biz.ioc.CarSalesMan"/>  
<bean id="greetingService"  
    class="edu.biz.ioc.SalesManager">  
    <property name="salesMan">  
        <ref bean="salesMan"/>  
    </property>  
</bean>  
</beans>
```

■ DI(Dependency Injection)

– Constructor Injection

Java

```
public class SalesManager {  
    private SalesMan salesMan;  
    public SalesManager(SalelsMan sm) {  
        this.salesMan = sm;  
    }  
}////~
```

XML

```
<beans>  
<bean id="salesMan" class="edu.biz.ioc.CarSalesMan"/>  
<bean id="salesMan" class="edu.biz.ioc.CarSalesMan"/>  
<bean id="salesManager"  
    class="edu.biz.ioc.SalesManager">  
    <constructor-arg>  
        <ref bean="salesMan"/>  
    </constructor-arg>  
</bean> </beans>
```

■ DI(Dependency Injection)

- 리스트나 배열 묶기

Java

```
public class ListManager {  
    public List<Object> list;  
    public void setList(List<String> list) {  
        this.list = list;  
    }  
    public void print() {  
    }  
}
```

XML

```
<property name="barList">  
    <list>  
        <value>bar1</value>  
        <ref bean="bar2" />  
    </list>  
</property>
```

■ DI(Dependency Injection)

– Set (java.util.Set)묶기

Java

```
public class SalesManager {  
    private Set items;  
    public setItems(Set set) {  
        this.items = set;  
    }  
}////~
```

XML

```
<property name="barList">  
    <set>  
        <value>bar1</value>  
        <ref bean="bar2" />  
    </set>  
</property>
```

■ DI(Dependency Injection)

– Map(java.util.Map) 묶기

Java

```
public class SalesManager {  
    private Map items;  
    public setItems(Map map) {  
        this.items = map;  
    }  
}////~
```

XML

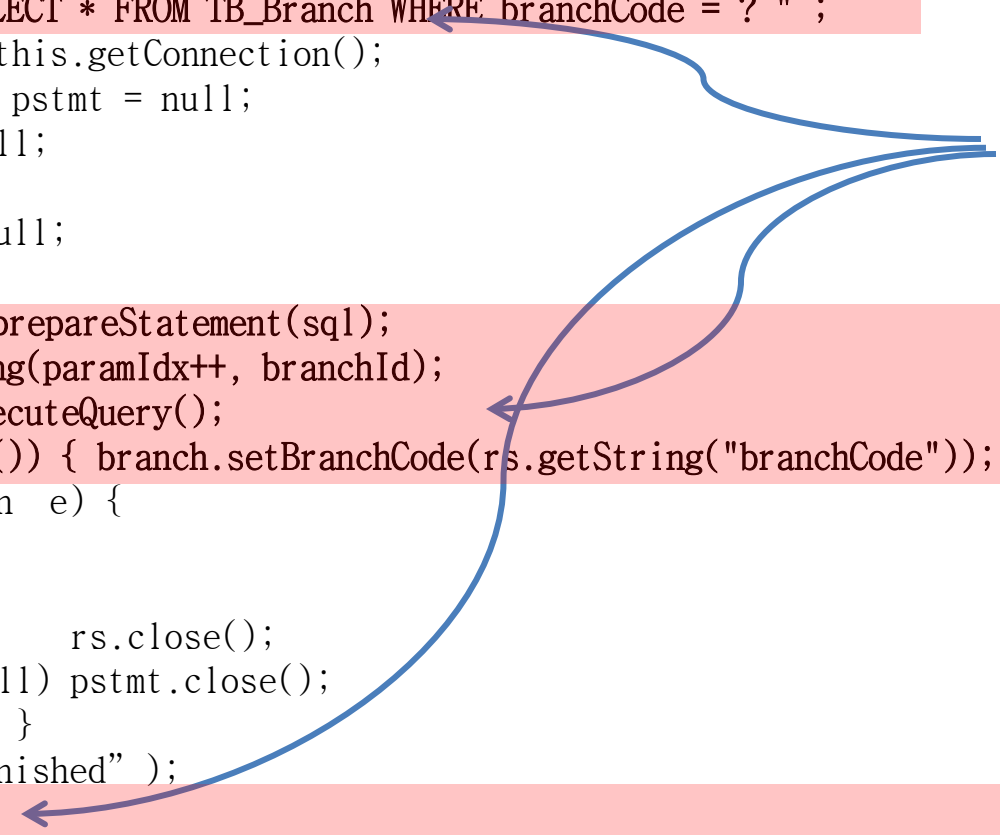
```
<property name="barList">  
    <map>  
        <entry key="key1">  
            <value>bar1</value>  
        </entry>  
        <entry key="key2">  
            <value>bar1</value>  
        </entry>  
    </map>  
</property>
```


■ 전통적인 DAO 단의 코드 분석

- 전통적인 코드에서 문제점 → 반복,부차적인 것들이 뒤섞여 있다.

```
public Branch select(String branchId) throws SQLException {
    Logger logger = LogFactory.getLog(getClass());
    logger.debug( "start" );
    String sql = " SELECT * FROM TB_Branch WHERE branchCode = ? " ;
    Connection con = this.getConnection();
    PreparedStatement pstmt = null;
    ResultSet rs = null;
    int paramIdx = 1;
    Branch branch = null;
    try {
        pstmt = con.prepareStatement(sql);
        pstmt.setString(paramIdx++, branchId);
        rs = pstmt.executeQuery();
        while(rs.next()) { branch.setBranchCode(rs.getString("branchCode")); }// while
    } catch (SQLException e) {
        throw e;
    } finally {
        if(rs != null) rs.close();
        if (pstmt != null) pstmt.close();
        this.close();
    }
    logger.debug( "finished" );
    return branch;
}//:
```

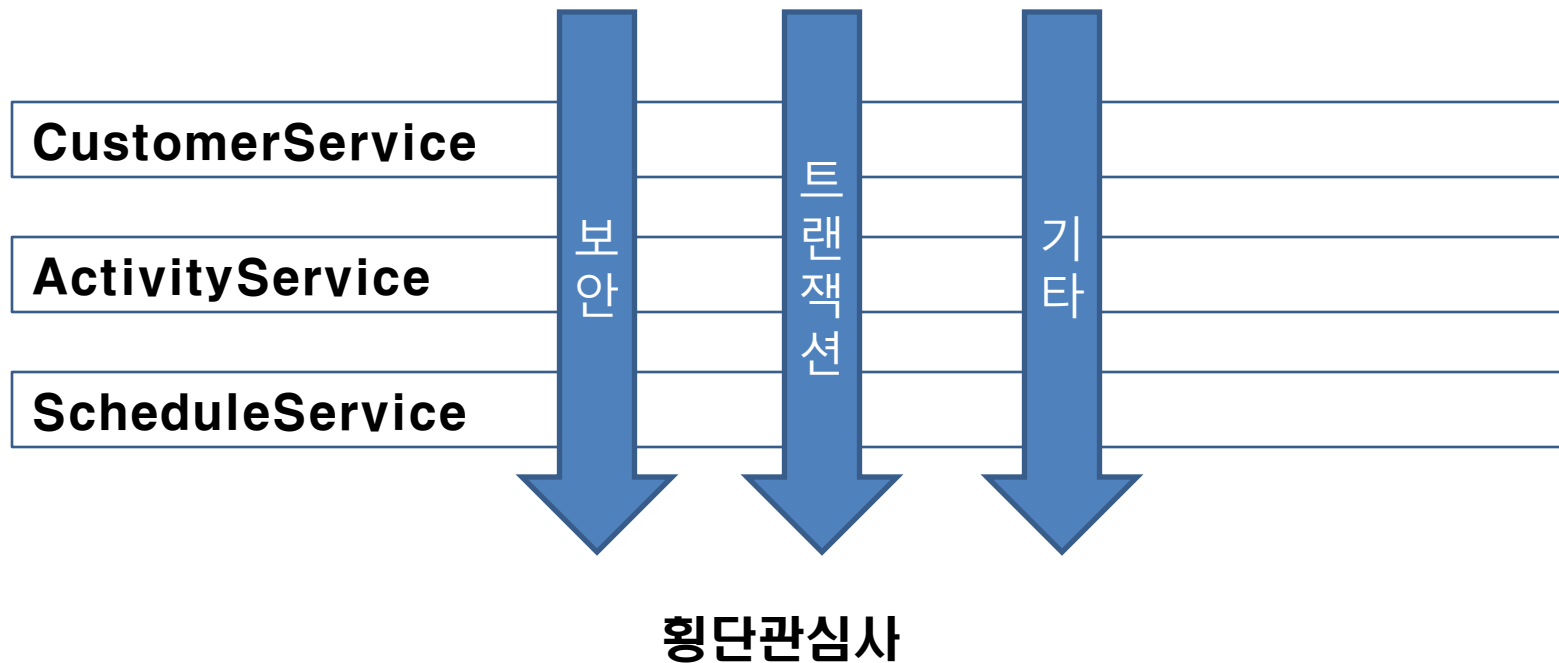
핵심 관심사



■ 횡단관심사

- 복잡한 문제를 다루는 최선의 방법은 문제를 단순하게 만드는 것

객체지향을 사용한다고 하더라도 부차적인 관심사를 코드에서 제거하지 못한다. 따라서 생산성이 저하되고 품질이 저하된다. 또한 개선의 어려움이 있다. 복잡한 문제를 다루는 최선의 방법은 문제를 단순하게 만드는 것이다. 그런데 그에 대한 해결책은?



■ AOP 용어

- 실행하기 전에 먼저 말하는 법을 배우자. AOP가 뭐지????

결합점(Join point)	인스턴스의 생성시점,메소드를 호출하는 시점, Exception이 발생하는 시점과 같이 애플리케이션이 실행될 때 특정작업이 실행되는 시점 을 의미한다.
교차점(pointcut)	교차점은 결합점들을 선택하고 결합점의 환경정보(context)를 수집하는 프로그램의 구조물이다. Target 클래스와 Advice가 결합(Weaving)될때 둘 사이의 결합규칙을 정의 하는 것이다.
충고(Advice)	충고는 교차점에서 지정한 결합점에서 실행되어야하는 코드 이다.
에스팩트(Aspect)	에스팩트는 AOP의 중심단위. Advice와 pointcut을 합친 것 이다.(Advisor)
대상(target)	충고를 받는 클래스 를 대상(target)라고 한다. 대상은 여러분이 작성한 클래스는 물론, 별도의 기능을 추가하고자 하는 써드파티 클래스가 될 수 있다.
엮기(Weaving)	에스팩트를 대상 객체에 제공하여 새로운 프록시 객체를 생성하는 과정 을 말한다.
프록시(Proxy)	대상객체에 충고가 적용된후 생성된 객체

■ Spring의 AOP

- Spring에서 제공하는 메소드 결합점 이해

객체지향을 사용한다고 하더라도 부차적인 관심사를 코드에서 제거하지 못한다. 따라서 생산성이 저하되고 품질이 저하된다. 또한 개선의 어려움이 있다. 복잡한 문제를 다루는 최선의 방법은 문제를 단순하게 만드는 것이다. 그런데 그에 대한 해결책은?

```

•public class SmallMart implements SmallMartInterface {
    ← 사전충고
    •public String[] getProducts(String productName) throws Exception{
        String[] arr = { "Good morning", "Good evening" };
        System.out.println(arr[0]);
        System.out.println(arr[1]);
        return null;
        ← 예외충고
    }//:
    ← 주변충고
}///~
    ← 사후충고

```

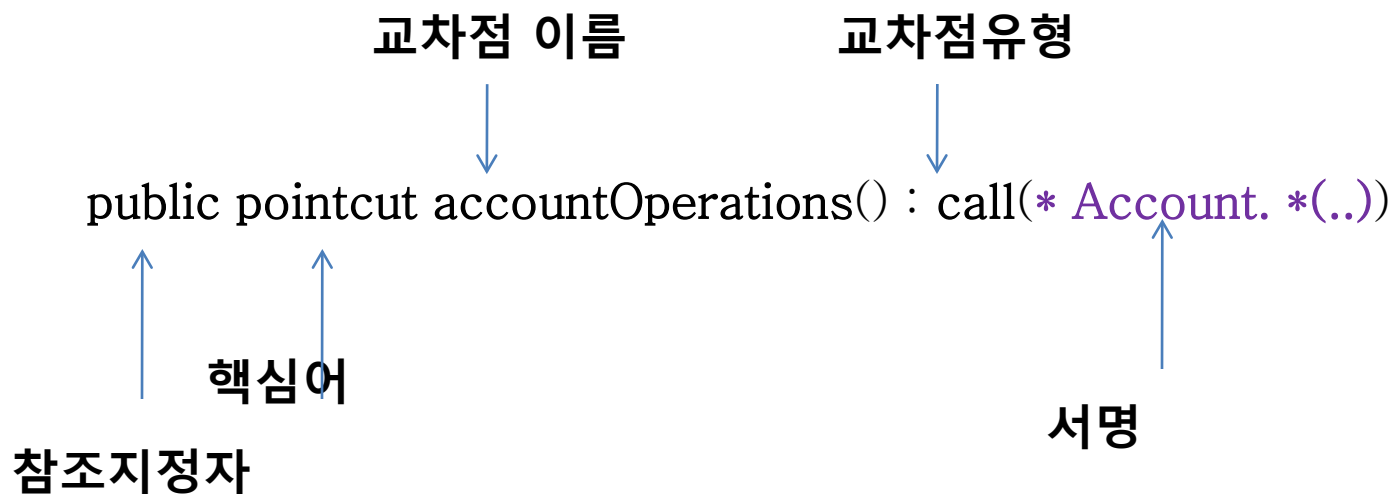
호출부분:

```
smallMart.getProducts("aaa");
```

■ AspectJ 교차점

- AspectJ 교차점 문법

교차점은 프로그램 실행중에 결합점(join point)을 포착하거나 알려줌. 교차점은 에스펙트,클래스,인터페이스 내에 선언됨. AspectJ에서 교차점은 무명이거나 이름을 가질 수 있음.



문법: [참조지정자] pointcut 교차점-이름(([매개변수]):교차점-정의

■ AspectJ 교차점

- AspectJ 와일드 카드

와일드 카드

*	마침표 제외, 문자가 0개이상
..	마침표 포함, 문자가 0개이상
+	주어진 타입의 임의의 서브클래스, 서브인터페이스

단항연산자

!	부정연산자, 교차점에 표시된 결합점을 제외한 모든 결합점을 선택
---	-------------------------------------

이항연산자

	두개의 교차점을 결합하면 양쪽 교차점에 있는 모든 결합점이 선택
&&	두개의 교차점을 모두 만족하는 결합점만 선택

괄호를 사용하여 기본연산자 우선순위를 변경하고코드를 읽기 쉽게 만들 수 있다.

■ AspectJ 교차점

- 서명(signature) 패턴

Java에서 클래스, 인터페이스, 메소드와 필드들은 모두 서명을 가지고 있다. 결합점을 지정할 때 이 서명을 사용한다.

타입서명패턴

*	클래스, 인터페이스, 패키지이름의 일부 표현
..	직접 또는 간접 서브패키지를 표현
+	서브타입(서브클래스 또는 서브인터페이스)를 표현

예시

```
javax.swing.JComponent+
```

javax.swing.JComponent의 모든 서브클래스를 의미

■ AspectJ 교차점

- 타입서명 샘플

타입서명패턴 예

서명패턴	대응되는 타입
Account	Account
*Account	이름이 Account로 끝나는 타입
Java.*.Date	Java 패키지의 직접 서브패키지에 있는 타입 Date Ex) java.util.Date, java.sql.Date
Java..*	Java 패키지 또는 java 패키지의 직접 서브패키지에 있는 모든 타입 Ex) java.awt, java.util 또는 java 패키지의 간접 서브패키지에 있는 모든 타입 Ex) java.awt.event, java.util.logging
Javax.*Model+	Java 패키지 또는 간접 , 직접 서브패키지 이름이 Model로 끝나거나 이들에서 상속된 서브타입 Ex) TableModel, TreeModel 과 이것에서 상속된 서브타입

■ AspectJ 교차점

- 타입서명 샘플

연산자를 사용하여 결합한 타입서명 예

서명패턴	대응되는 타입
<code>!Vector</code>	Vector이외의 모든 타입
<code>Vector Hashtable</code>	Vector 또는 Hashtable
<code>Javax..*.Model javax.swing.text.Document</code>	Javax 패키지나 이것의 직접 또는 간접 서브패키지에 있는 이름이 Model로 끝나는 모든 타입 또는 javax.swing.text.Document
<code>Java.util.RandomAccess+ && java.util.List+</code>	지정된 2개 인터페이스 모두를 구현하는 타입, 예를 들어 java.util.ArrayList는 2개의 인터페이스를 모두 구현하므로 이 서명에 의해 선택된다.

■ AspectJ 교차점

- 메서드 서명 샘플

메서드 서명의 예

서명패턴	대응되는 타입
<code>public void Account.set*(*)</code>	타입에 관계없는 하나의 파라미터 , set으로 시작하는 Account의 public method
<code>public void Account.*()</code>	All public methods in the Account class that return void and take no arguments.
<code>public * Account.*()</code>	All public methods in the Account class that take no arguments and return any type.
<code>public * Account.*(..)</code>	All public methods in the Account class taking any number and type of arguments.

■ AspectJ 교차점

- 메서드 서명 샘플

메서드 서명의 예

서명패턴	대응되는 타입
<code>* Account.*(..)</code>	All methods in the Account class. This will even match methods with private access.
<code>!public * Account.*(..)</code>	All methods with nonpublic access in the Account class. This will match the methods with private, default, and protected access.
<code>public static void Test.main(String[] args)</code>	The static main() method of a Test class with public access.
<code>* Account+.*(..)</code>	All methods in the Account class or its subclasses. This will match any new method introduced in Account's subclasses.

■ AspectJ 교차점

- 메서드 서명 샘플

메서드 서명의 예

서명패턴	대응되는 타입
* java.io.Reader.read(..)	Any read() method in the Reader class irrespective of type and number of arguments to the method. In this case, it will match read(), read(char[]), and read(char[], int, int).
* java.io.Reader.read(char[], ..)	Any read() method in the Reader class irrespective of type and number of arguments to the method as long as the first argument type is char[]. In this case, it will match read(char[]) and read(char[], int, int), but not read().

■ AspectJ 교차점

- 메서드 서명 샘플

메서드 서명의 예

서명패턴	대응되는 타입
* javax..*.add*Listener(Event - Listener+)	Any method whose name starts with add and ends in Listener in the javax package or any of the direct and indirect subpackages that take one argument of type EventListener or its subtype. For example, it will match TableModel.addTableModelListener(Table- ModelListener).
* *.*(..) throws Remote- Exception	Any method that declares it can throw RemoteException.
public void Collection.clear()	The method clear() in the Collection class that has public access, returns void, and takes no arguments.
public void Account.debit(float) throws InsufficientBalanceExcepti	The public method debit() in the Account class that returns void, takes a single float argument, and

■ AspectJ 교차점

- 생성자 서명 샘플

생성자 서명의 예

서명패턴	대응되는 타입
<code>public Account.new()</code>	A public constructor of the Account class taking no arguments.
<code>public Account.new(int)</code>	A public constructor of the Account class taking a single integer argument.
<code>public Account.new(..)</code>	All public constructors of the Account class taking any number and type of arguments
<code>public Account+ .new(..)</code>	Any public constructor of the Account class or its subclasses.
<code>public *Account.new(..)</code>	Any public constructor of classes with names ending with Account. This will match all the public constructors of the SavingsAccount and CheckingAccount classes.
<code>public Account.new(..) throws</code>	

InvalidAccountNumberExce

■ AspectJ 교차점

- 유형별 결합점

유형별 결합점 문법

결합점 유형	유형별 교차점 문법
메소드 실행	execution(MethodSignature)
메소드 호출	call(MethodSignature)
생성자 실행	execution(ConstructorSignature)
생성자 호출	call(ConstructorSignature)
클래스 초기화	staticinitialization(TypeSignature)
필드 읽기 참조	get(FieldSignature)
필드 쓰기 참조	set(FieldSignature)
예외 처리 실행	handler(TypeSignature)
객체 초기화	initialization(ConstructorSignature)
객체 초기화 이전	preinitialization(ConstructorSignature)
충고실행	adviceexecution()

■ AspectJ 교차점 - 어휘 구조에 기초한 교차점

- 어휘 유효범위는 작성된 코드의 정적인 부분을 칭하는 용어

어휘 구조에 기초한 교차점 예

교차점	설명
<code>within(Account)</code>	Account 클래스의 어휘 유효범위 내에 있는 모든 결합점
<code>within(Account+)</code>	Account 클래스와 서브 클래스의 어휘 유효범위 내에 있는 모든 결합점
<code>withincode(* Account.debit(..))</code>	Account 클래스와 서브 클래스의 어휘 유효범위 내에 있는 모든 결합점
<code>withincode(* *Account.getBalance(..))</code>	Account로 이름이 끝나는 클래스의 모든 <code>getBalance()</code> 의 어휘 유효범위 내에 있는 모든 결합점

■ AspectJ 교차점 - 실행객체 교차점

- 실행시점의 객체의 종류에 기초하여 결합점을 선택한다.

실행객체 교차점 예

교차점	설명
this(Account)	현재 객체가 Account인 모든 결합점. 현재 실행객체가 Account이거나 SavingsAccount와 같이 서브클래스이면 그 객체의 모든 메서드 호출(정적메소드제외)과 멤버 변수 참조등이 결합점으로 선택
target(Account)	메서드의 실행 대상 객체가 Account의 인스턴스인 경우 그 객체의 모든 결합점. 대상 객체가 Account이거나 SavingsAccount와 같이 서브클래스이면 그 객체의 모든 메서드 호출(정적메소드제외)과 멤버변수 참조등이 결합점으로 선택

이 교차점들은 타입을 매개변수로 사용. 타입을 지정할 때 *나 ..와 같은 와일드 카드를 사용할 수 없다.

■ AspectJ 교차점 - 매개변수 교차점 및 조건검사 교차점

- 매개변수 및 조건검사에 예 기초하여 결합점을 선택

매개변수 교차점 예

교차점	설명
args(String, ..., int	첫번째 매개변수가 String 타입이고, 마지막 매개변수가 int 타입
args(RemoteException)	RemoteException 타입을 단 한개의 매개변수로 가진 모든 결합점

조건검사 교차점

교차점	설명
if(System.currentTimeMillis(> triggerTime)	현재 시간이 triggerTime을 지난 후에 발생하는 모든 결합점
if(circle.getRadius() < 5)	circle의 radius가 5보다 작은 모든 결합점

■ 충고(advice)

- 충고는 실제로 대상 객체에 무엇인가를 서술한다.

이전충고(before advice)

```
before():call( * Account.*(..)) {  
    .... 여기서 무엇을 한다.  
}
```

이후충고(after advice)

```
after():call( * Account.*(..)) {  
    .... 메서드의 실행이 끝남을 로깅  
}
```

이후충고 - 예외발생시

```
after() throwing : call(* Account.*(..)) {  
    ...메서드의 비정상적 종료를 로깅  
}
```

이후충고 - 반환값 포착

```
after() returning (Amount amt) : call(* Account.*(..) {  
    ...메서드의 정상적인 종료를 로깅  
}
```

■ 충고(advice)

- 충고는 실제로 대상 객체에 무엇인가를 서술한다.

대체충고(arround advice)

void around(Account account, float amount) throws

InsufficientBalanceException:

```
    call(* Account.debit(float) throws InsufficientBalanceException)
    && target(account)
    && args(amount) {
        try {
            proceed(account , amount);
        } catch (InsufficientBalanceException ex) {
            .. 로직
        }
    }
```

■ 결합점에서 충고로 환경정보 전달

- 충고를 구현하기 위해 결합점의 자료가 필요한 경우 처리

교차점을 사용하여 실행시점의 환경정보를 얻어서 충고에게 전달, `this()`, `target()`, `args()`
 교차점은 환경정보를 제공. 아래에서 이전충고의 무명교차점은 메서드와 관련된 모든
 매개 변수를 수집. 현재 실행객체를 얻으려면 `target()` 대신 `this()`를 사용

환경정보 전달하기 - 무명 교차점 사용

```
before (Account account, float amount):
  call (void Account, credit(float))
  && target ( account)
  && args (amount) {
    System.out.println("Crediting" + amount + "to " + account);
  }
```

매개변수전달

대상객체전달

■ 결합점에서 충고로 환경정보 전달

- 충고를 구현하기 위해 결합점의 자료가 필요한 경우 처리

유명교차점이 포착한 실행 객체와 매개변수 환경정보를 결합점에서 충고로 전달하기. 아래에서는 유명 교차점을 사용. 유명교차점을 사용하는 경우는 유명 교차점에서 환경 정보를 수집 해야 함.

환경정보 전달하기 - 유명 교차점 사용

```
public creditOperation(Account account, float amount) :  
    call(void Account.credit(float))  
    && target( account )  
    && args( amount);
```

```
before(Account account, float amount) :  
    creditOperation(account, amount) {  
        System.out.println("Crediting " + amount + " to " + account);  
    }
```

■ 결합점에서 충고로 환경정보 전달

- 충고를 구현하기 위해 결합점의 자료가 필요한 경우 처리

대체충고의 예

```
public aspect FailureHandlingAspect {
    final int MAX_TRIES = 3;
```

```
Object around() throws RemoteException    [1]충고의 메서드 부분
```

```
: call( * RemoteService.get*(..) throws RemoteException [2]충고의 무명교차점 부분
```

```
int retry = 0;
```

```
while(true) {
```

```
    try {
```

```
        return proced(); [3]포착된 결합점의 실행
```

```
    }catch(RemoteException ex) {
```

```
        System.out.println("Encountered " + ex);
```

```
        if(++ retry > MAX_RETRIES) {
```

```
            throw ex;
```

```
        }
```

```
        System.out.println("WtRetrying...");
```

```
    }
```

```
}
```

■ 반사기능으로 결합점 정보 참조하기

- reflection을이용하여 결합점과 관련된 정적/동적 정보를 사용하기

3가지 특별한 종류의 객체

thisJoinPoint	결합점과 관련된 동적정보
thisJoinPointStaticPart	결합점의 구조적인 정보
thisEnclosingJoinPointStaticPart	결합점을 둘러싼 환경정보

JoinPoint

getArgs():Object[]
getTarget():Object
getThis():Object
getStaticPart():JoinPoint.StaticPart

JoinPoint.StaticPart

getKind():String
getSignauture():Signature
getSourceLocation():SourceLocation

Signature

getDeclaringType():Class
getModifiers():int
getName():String

■ Aspect

- 관점지향 횡단 관심사를 구현하는 기본 단위로 클래스와 유사

Aspect 선언문법

```
[access specification] aspect <AspectName>
    [extends class-or-aspect-name]
    [implements interface-list]
    [<association-specifier>(Pointcut)] {
    ...aspect body
}
```

- 에스펙트는 데이터 멤버와 메서드를 포함할 수 있다.
- 에스펙트는 참조 지정자를 가질 수 있다.
- 에스펙트를 추상(abstract) 에스펙트로 정의할 수 있다.

```
public abstract aspect AbstractLogging {
    public abstract pointcut logPoints(); // 추상교차점
    public abstract Logger getLogger(); // 추상메소드
    before(): logPoints() {           // 추상교차점에 대한 충고
        getLogger().log(thisJoinPoint); // 추상 메서드 사용
    }
}
```

■ Spring AOP – @AspectJ support

– Enabling @AspectJ Support

Spring Configuration file

AspectJ 지원을 활성화 하기 위해서 스프링 설정 파일에 다음을 추가

```
<aop:aspectj-autoproxy/>
```

@Aspect annotation을 가진 bean class를 다른 빈과 마찬가지로 선언

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect"> <!-- configure properties of aspect here as normal --> </bean>
```

class 작성

@Aspect annotation을 가진 bean class를 작성

```
package org.xyz;  
import org.aspectj.lang.annotation.Aspect;
```

```
@Aspect  
public class NotVeryUsefulAspect {  
}
```

■ Spring AOP - Pointcut

- Supported Pointcut

스프링에서 지원하는 포인트 컷

구분	설명
execution	메소드 실행에 대응하는 조인 포인트
within	특정 타입(에 포함되는)의 메소드 실행으로 제한
this	참조 빈. 즉, Spring AOP 프록시가 주어진 타입인 경우로 제한. 타입과 일치하는 모든 JoinPoint
target	프록시의 대상이 되는 객체가 주어진 타입인 경우로 제한. 타입과 일치하는 target 객체의 모든 메소드 Joinpoint
args	인자가 주어진 타입인 경우로 제한
@target	실행하는 객체의 클래스가 주어진 타입의 애노테이션을 갖는 경우로 제한
@args	실행시점의 인자가 특정 타입의 애노테이션을 갖는 경우로 제한
@within	주어진 애노테이션을 갖는 타입으로 제한
@annotation	조인 포인트의 주체가 주어진 애노테이션을 갖는 경우로 제한

'&&', '||', '!' 를 이용하여 논리연산을 할 수 있고, id로써 연산도 가능.

■ Spring AOP

- Combining Pointcut expression

스프링에서 포인트컷 표현을 '&&', '||', '!' 를 이용하여 논리연산을 할 수 있고, id로써 연산도 가능 .

```
@Pointcut("execution(public * *(..))")
private void anyPublicOperation() {}
```

```
@Pointcut("within(com.xyz.someapp.trading..*)")
private void inTrading() {}
```

```
@Pointcut("anyPublicOperation() && inTrading()")
private void tradingOperation() {}
```

Pointcut 참조

외부클래스에 정의된 Pointcut을 참조하기 위해 아래와 같이 함

```
@Before("x.y.z.ExamplePointcut.beforePointcut()")
public void beforeAdvice() {
    ...
}
```

Spring AOP는 런타임 Proxy방식이기 때문에 this와 target 지시자는 같은 객체를 참조 하며

@target과 @within 지시자 역시 같은 객체를 참조 합니다.

■ Spring AOP

– Pointcut Examples

```
execution(public * *(..))
```

```
execution(* set*(..))
```

```
execution(* com.xyz.service.AccountService.*(..))
```

```
execution(* com.xyz.service.*.*(..))
```

```
execution(* com.xyz.service..*.*(..))
```

```
within(com.xyz.service.*)
```

```
within(com.xyz.service..*)
```

```
this(com.xyz.service.AccountService)
```

```
target(com.xyz.service.AccountService)
```

```
args(java.io.Serializable)
```

```
@target(org.springframework.transaction.annotation.Transactional)
```

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

```
@args(com.xyz.security.Classified)
```

■ Spring AOP – Advice

– Before advice

@Before annotation

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

    @Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
    public void doAccessCheck() {
        // ...
    }
}
```

Using an in-place pointcut expression

```
@Aspect
public class BeforeExample {

    @Before("execution(* com.xyz.myapp.dao.*.*(..))")
    public void doAccessCheck() {
        // ...
    }

}
```

■ Spring AOP – Advice

- After returning advice

@AfterReturning annotation

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

    @AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
    public void doAccessCheck() {
        // ...
    }
}
```

Access the actual value that was returned

```
@Aspect
public class AfterReturningExample {

    @AfterReturning(
        pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
        returning="retVal")
    public void doAccessCheck(Object retVal) {
        // ...
    }
}
```

■ Spring AOP – Advice

– After throwing advice

@AfterThrowing annotation

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

    @AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
    public void doRecoveryActions() {
        // ...
    }
}
```

예외를 제한하기(지정된 타입의 예외가 발생했을 때만 적용)

```
@Aspect
public class AfterThrowingExample {

    @AfterThrowing(
        pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
        throwing="ex")
    public void doRecoveryActions(DataAccessException ex) {
        // ...
    }
}
```


■ Spring AOP – Advice

– After (finally) advice

@After annotation

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;

@Aspect
public class AfterFinallyExample {

    @After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
    public void doReleaseLock() {
        // ...
    }
}
```

■ Spring AOP - Advice

- Around advice

@Around annotation

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class AroundExample {

    @Around("doAroundPointcut()")
    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
        // start stopwatch
        Object retVal = pjp.proceed();
        // stop stopwatch
        return retVal;
    }

}
```

첫번째 매개변수로 반드시 ProceedingJoinPoint 타입의 변수를 선언.

■ Spring AOP - Advice

- 어드바이스에 파라미터 넘기기(Passing parameters to advice)

어드바이스에서 넘기기

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +  
        "args(account,..)")  
public void validateAccount(Account account) {  
    // ...  
}
```

포인트컷을 사용하여 넘기기

```
@Pointcut("execution( * register*(..))")  
private void loggingOperation() {}// the pointcut signature  
  
@Pointcut("loggingOperation() && args(bean)")  
private void loggingOperation2(AopBean bean) {}// the pointcut signature  
  
@Before("loggingOperation2(bean)")  
public void validParameter2(AopBean bean) {  
    //.....  
}//:
```

■ Spring AOP - JoinPoint

- JoinPoint 사용방법

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() ")
public void validateAccount(JoinPoint joinPoint) {
    // ...
}

@AfterReturning( pointcut="execution(public * com.education.*(..))", returning = "ret")
public void returningLogging(JoinPoint joinPoint , Object ret) {
    // ...
}

@Pointcut("execution( * com.education..*.addCourse*(..))")
private void loggingOperation() {}// the pointcut signature

@Pointcut("loggingOperation() && args(..,userId)")
private void loggingOperationArgs(String userId) {}// the pointcut signature

@Before("loggingOperationArgs(userId)")
public void beforeLogging( JoinPoint jp, String userId) {
    System.out.println(jp.getTarget().toString());
    System.out.println("검색조건 사용자 아이디 = " + userId);
}//:
```

■ Spring AOP - XML Schema 이용

- XML 설정방법 살펴보기

XML 설정

```
<aop:config>
  <aop:aspect id="loggingAspect" ref="loggingAspect2">
    <aop:pointcut id="publicMethod" expression="execution(public *
com.education..*(..))" />
    <aop:before pointcut-ref="publicMethod" method="beforeLogging" />
    <aop:after-returning pointcut-ref="publicMethod" method="afterReturningLoggin" />
  </aop:aspect>
</aop:config>
```

Tag 설명

구분	설명
<aop:config>	AOP 설정 정보 표현
<aop:aspect>	Aspect 설정
<aop:pointcut>	포인트컷 설정
<aop:around>	Around Advice 설정

■ Spring AOP - XML Schema 이용

- XML 설정방법 살펴보기

Advice Tag 설명

구분	설명
<aop:before>	
<aop:after-returning>	
<aop:after-throwing>	
<aop:after>	
<aop:around>	

■ Spring AOP - XML Schema 이용

- XML 설정방법 살펴보기

After Returning Advice

```
<aop:config>
  <aop:aspect id="loggingAspect" ref="loggingAspect2">
    <aop:pointcut id="publicMethod" expression="execution(public *
com.education..*(..))" />
    <aop:before pointcut-ref="publicMethod" method="beforeLogging" />
    <aop:after-returning pointcut-ref="publicMethod" method="afterReturningLogin"
returning="ret"/>
  </aop:aspect>
</aop:config>
```

(Java)

```
public void afterReturningLogin(JoinPoint joinPoint, Object ret) {
    System.out.println("실행이 완료된 메소드 :" + joinPoint.getSignature().getName());
}//:
```

■ Spring AOP - XML Schema 이용

- XML 설정방법 살펴보기

After Throwing Advice

```
<aop:config>
  <aop:aspect id="loggingAspect" ref="loggingAspect2">
    <aop:pointcut id="publicMethod" expression="execution(public *
com.education..*(..))" />
    <aop:before pointcut-ref="publicMethod" method="beforeLogging" />
    <aop:after-throwing pointcut-ref="publicMethod" method="afterReturningLoggin"
throwing="ex"/>
  </aop:aspect>
</aop:config>
```

(Java)

```
public void afterThrowingLoggin(SomeException ex) {
    ...
}//:
```


■ Spring JDBC

- 예외처리

DataSourceException

- Spring의 DAO 프레임워크가 던지는 모든 예외는 DataSourceException
- DataSourceException은 반드시 직접처리할 필요는 없다.
 - RuntimeException이기때문에 unchecked exception에 속한다.
- checked exception 이 과도한 catch나 throws 절을 야기시켜 코드가 난잡하게 만
들
수 있다.
- Unchecked-Exception이 발생하는 경우는 대부분 복구가 불가능한 것이므로 직접처
리
할 필요는 없다.
- 만약 복구가 가능한 경우라면 예외를 잡아 호출 스택으로 전달되도록 할 수 있다.

예외처리의 규칙작성

- 비지니스 로직을 수행하는 중 발생하는 비지니스 오류는 Checked Exception으로
처리하고 그렇지 않으면 Unchecked Exception으로 처리
- Checked Exception중 사용자가 인지해야되는 Exception은 해당 메시지 출력한
다.

■ Spring JDBC

- Tomcat 설정

- Spring의 DAO 프레임워크에서는 Connection 객체를 DataSource를 통해서 취득.
- Spring에서는 Connection 객체를 직접 프로그래머가 다루지 않는다.

<context>.xml

```
<Resource auth="Container" description="MDB DB Connection" name="jdbc/DS_EDU"
type="javax.sql.DataSource"/>
  <ResourceParams name="jdbc/DS_EDU">
    <parameter>      <name>password</name>    <value></value>    </parameter>
    <parameter>
      <name>url</name>
      <value>jdbc:odbc:Driver={Microsoft Access Driver
(*.mdb)};DBQ=Z:\Weducation\Wsource\Wspring.ibatis\Wdata\WEDU.mdb</value>
    </parameter>
    <parameter>
      <name>driverClassName</name>
      <value>sun.jdbc.odbc.JdbcOdbcDriver</value>
    </parameter>
    <parameter>
      <name>username</name>
      <value>admin</value>
    </parameter>
  </ResourceParams>
```

■ Spring JDBC

- web.xml에 설정하고 JNDI로 부터 DataSource 취득하기

web.xml

```
<resource-ref>
  <description>MDB DB Connection</description>
  <res-ref-name>jdbc/DS_EDU</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

JNDI로 부터 DataSource 취득

- Spring에서는 DataSource를 애플리케이션의 다른 서비스 객체들과 마찬가지로 스프링 빈으로 취급하는데, 이경우 JndiObjectFactoryBean을 사용한다.

```
<bean id="dataSource"
  class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/jdbc/myDataSource</value>
  </property>
</bean>
```

■ Spring JDBC

- web.xml에 설정하고 JNDI로 부터 DataSource 취득하기

web.xml

```
<resource-ref>
  <description>MDB DB Connection</description>
  <res-ref-name>jdbc/DS_EDU</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

JNDI로 부터 DataSource 취득

- Spring에서는 DataSource를 애플리케이션의 다른 서비스 객체들과 마찬가지로 스프링 빈으로 취급하는데, 이경우 JndiObjectFactoryBean을 사용한다.

```
<bean id="dataSource"
  class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/jdbc/myDataSource</value>
  </property>
</bean>
```

■ Spring JDBC

- Spring에서 지원하는 DAO 특징 알아보기

일관된 DAO 지원

- 스프링은 데이터 접근 프로세스에 있어서 고정된 부분과 가변적인 부분을 명확히 분류.
- 고정적인 부분은 템플릿(template)이며, 가변적인 부분은 콜백(callback)이다.
- 템플릿은 프로세스의 고정적인 부분을 관리.
- 콜백은 구체적인 구현을 넣어야 하는 장소.
- 고정된 부분은 트랜잭션 관리, 자원관리,예외처리등을 맡는다.
- 콜백은 질의문 생성, 파라미터 바인딩, 결과집합 마샬링 등 처리.

로직에만 집중

■ Spring JDBC

- 전통적인 방식의 DAO 개발코드 살펴보기

JDBC 코드의 문제점

```
public insertPerson(Person person) throws SQLException {  
    Connection con = null;  
    PreparedStatement stmt = null; 】 자원선언  
  
    try {  
        con = dataSource.getConnection(); 연결개시  
        .....  
        stmt.executeUpdate();  
    } catch (SQLException e) { 】 예외처리  
    }  
    finally {  
        try { if((stmt != null) stmt.close(); }  
        catch (SQLException e) {LOGGER.warn(e);} 】 자원반환  
        try { if((con != null) con.close(); }  
        catch (SQLException e) {LOGGER.warn(e);}   
    }  
}
```

■ Spring JDBC

- 스프링의 DAO 핵심인 JdbcTemplate

JdbcTemplate

- 스프링의 모든 데이터 접근 프레임워크는 템플릿 클래스를 포함한다. 이 경우 템플릿 클래스는 JdbcTemplate 클래스이다.
- JdbcTemplate 클래스가 작업하기 위해 필요한 것은 DataSource 뿐이다.
- 스프링의 모든 DAO 템플리 클래스는 스레드에 안전하기 때문에, 애플리케이션 내의 각각의 DataSource에 대해서 하나의 JdbcTemplate 인스턴스만을 필요로 한다.

```
public class MyDaoImpl implement MyDao {  
  
    private JdbcTemplate jdbc;  
  
    public void setJdbcTeampate(JdbcTemplate jdbcTemplate) {  
        this.jdbc = jdbcTemplate;  
    }//:  
}///~
```

■ Spring JDBC

– DataSource, JdbcTemplate, DAO 정의

Bean 정의

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/jdbc/education</value>
  </property>
  <property name="resourceRef" value="true" />
</bean>

<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource">
    <ref bean="dataSource" />
  </property>
</bean>

<bean id="basicDao" class="edu.jdbc.basic.BasicDao">
  <property name="jdbcTemplate">
    <ref bean="jdbcTemplate" />
  </property>
</bean>
```


■ Spring JDBC

- int, long, double, Date 값을 반환하는 기본 쿼리 연습

기본쿼리

```
public int queryForInt() {
    String sql = " SELECT col_int FROM TBA_Basic";
    return this.jdbc.queryForInt(sql);
}//:

public long queryForLong() {
    String sql = " SELECT col_long FROM TBA_Basic";
    return this.jdbc.queryForLong(sql);
}//:

public double queryForDouble() {
    String sql = " SELECT col_double FROM TBA_Basic";
    Double d = (Double)this.jdbc.queryForObject(sql, Double.class);
    return d.doubleValue();
}//:

public Date queryForDate() {
    String sql = " SELECT col_date FROM TBA_Basic";
    Date d = (Date)this.jdbc.queryForObject(sql, Date.class);
    return d;
}//:
```

■ Spring JDBC

- 질의 결과를 Map에 담거나, RowCallbackHandler 사용하기

기본쿼리

```
/** * 쿼리 결과를 Map에 담는다. */
public List queryForList() {
    String sql = "SELECT top 100 * FROM TBA_Sample";
    return this.jdbc.queryForList(sql);
}

/** * 쿼리 결과를 Map에 담는다. */
public List queryForList2(String city) {
    String sql = "SELECT top 100 * FROM TBA_Sample WHERE city like ? ";
    Object[] args = { city + "%" };
    return this.jdbc.queryForList(sql, args);
}

public SampleBean queryWithProcessRow() {
    String sql = " SELECT top 1 * FROM TBA_Sample ";
    final SampleBean bean = new SampleBean();
    this.jdbc.query(sql, new RowCallbackHandler() {
        public void processRow(ResultSet rs) throws SQLException {
            bean.setAddr(rs.getString("addr"));
            bean.setCity(rs.getString("city"));
            bean.setZipCode(rs.getString("zipcode"));
        }
    });
    return bean;
}
```

■ Spring JDBC

– insert, update, delete 처리하기

데이터 쓰기

```
public int insertWithObject(Person person) {

    StringBuffer sb = new StringBuffer();
    sb.append("INSERT INTO TBA_Person (id, name, age,sex, profile, createDate, updateDate )");
    sb.append("VALUES ");
    sb.append("( ?,?,?,?,?,,?)");
    Object[] args = { person.getId()
        ,person.getName(),person.getAge(),person.getSex(),person.getProfile()
        ,person.getCreateDate(),person.getUpdateDate()
        };
    return this.jdbc.update(sb.toString(), args);
}//:

public int update(Person person) {
    String sql = ""
        + "UPDATE TBA_Person "
        + " SET "
        + "     age = ? "
        + "     ,name = ? "
        + " WHERE id = ? ";
    Object[] args = { person.getAge(),person.getName(),person.getId() };
    return this.jdbc.update(sql, args);
}//:
```

■ Spring JDBC

- 여러개의 데이터를 한번에 쓰기

Batch Update

```

public int[] insert(final List persons) {
    String sql = ""
    + " update tba_person set                                Wn"
    + "     name = ? "
    + "     ,age  = ? "
    + "     ,sex  = ? where id = ? ";

    BatchPreparedStatementSetter setter = null;
    setter = new BatchPreparedStatementSetter() {
        public int getBatchSize() {
            return persons.size();
        }
    };
    public void setValues(PreparedStatement ps, int index)
    throws SQLException {
        Person person = (Person)persons.get(index);
        int parameterIndex = 1;
        ps.setString(parameterIndex++, person.getName());
        ps.setInt(parameterIndex++, person.getAge());
        ps.setString(parameterIndex++, person.getSex());
        ps.setString(parameterIndex++, person.getId());
    };
    return this.jdbc.batchUpdate(sql, setter);
}

```

■ Spring JDBC

- 객체를 목록에 담기(1)

RowMapper

```
public class SampleBeanRowMapper implements RowMapper {
    public Object mapRow(ResultSet rs, int arg1) throws SQLException {
        SampleBean bean = new SampleBean();
        bean.setAddr(rs.getString("addr"));
        bean.setCity(rs.getString("city"));
        bean.setZipCode(rs.getString("zipcode"));
        return bean;
    }
}

/**
 * 외부에서 구현된 RowMapper를 사용하여 리스트 반환
 */
public List queryForListWithExternalRowMapper() {
    String sql = " SELECT top 100  A.* FROM TBA_Sample A";
    return this.jdbc.query(sql, new SampleBeanRowMapper());
}
```

■ Spring JDBC

- 객체를 목록에 담기(2)

RowMapper

```
/**
 * 내부에서 구현된 RowMapper 이용하여 리스트 반환
 */
public List queryForListWithRowMapper() {
    String sql = " SELECT top 100  A.* FROM TBA_Sample A";
    return this.jdbc.query(sql,
        new RowMapper() {
            public Object mapRow(ResultSet rs, int arg1)
                throws SQLException {
                SampleBean bean = new SampleBean();
                bean.setAddr(rs.getString("addr"));
                bean.setCity(rs.getString("city"));
                bean.setZipCode(rs.getString("zipcode"));
                return bean;
            }
        });
}
```

■ Spring JDBC

– 저장프로시저 사용하기 (1)

CallableStatementCallback

```
public ProcTestBean execProcedure(final String userId) {  
  
    return (ProcTestBean)this.jdbc.execute("{ call simple(?, ?) } ",  
        new CallableStatementCallback() {  
        public Object doInCallableStatement(CallableStatement arg0)  
            throws SQLException {  
            ProcTestBean bean = new ProcTestBean();  
            arg0.setString(1, userId);  
            arg0.registerOutParameter(2, Types.VARCHAR);  
            arg0.execute();  
            bean.setMessage(arg0.getString(2));  
            return bean;  
        } // doInCallable...  
    });  
}
```

■ Spring JDBC

- 저장프로시저 사용하기(2)

```

public List execResultSetProc(final String userId) {
    return (List)this.jdbc.execute("{ call resultset_proc(?, ?) } ",
        new CallableStatementCallback() {
            public Object doInCallableStatement(CallableStatement arg0)
                throws SQLException {
                List list = new ArrayList();
                arg0.registerOutParameter(1, Types.INTEGER);
                arg0.registerOutParameter(2, Types.VARCHAR);
                boolean hasResults = arg0.execute();
                int errorCode = arg0.getInt(1);
                String message = arg0.getString(2);
                if(errorCode != 0)
                    throw new SQLException(message, "", errorCode);
                if(hasResults) {
                    ResultSet rs = null;
                    try {
                        rs = arg0.getResultSet();
                        while(rs.next()) {
                            CategoryTestBean bean = new CategoryTestBean();
                            bean.setCategoryId(rs.getString("categoryid"));
                            bean.setCateogryName(rs.getString("categoryName"));
                            list.add(bean);
                        }
                    } catch (Exception e) {
                    } finally {
                        if(rs != null) rs.close(); rs = null;
                    }
                }
            }
        });
}

return list;

```


■ Spring JDBC – Transaction

– Spring에서 지원하는 transaction

TransactionDefinition

```
public interface TransactionDefinition {  
    int getPropagationBehavior();  
    int getIsolationLevel();  
    int getTimeout();  
    boolean isReadOnly();  
}
```

getTimeOut

실행하는 트랜잭션이 시작해서 종료할 때까지의 시간을 초단위 제어

isReadOnly

실행하는 트랜잭션이 read-only 상태여부를 결정가능

getIsolationLevel

트랜잭션의 격리레벨을 결정

getPropagationBehavior

트랜잭션이 실행되어야 하는 범위에 대한 제어 및 여러개의 트랜잭션이 상호작용하는 것에 대한 결정

■ Spring JDBC – Transaction

– 격리레벨 이해하기

격리레벨(Isolation level)

ISOLATION_DEFULAT	개별적인 PlatformTransactionManager를 위한 디폴트 값
ISOLATION_READ_UNCOMMITTED	다른 트랜잭션에서 commit되지 않은 데이터 조회가능함으로 트랜잭션의 기능수행을 하지 않는다.
ISOLATION_READ_COMMITTED	대개의 데이터 베이스의 디폴트이다. 다른 트랜잭션에서 commit하지 않은 데이터는 볼수 없다.
ISOLATION_REPEATABLE_READ	다른 트랜잭션이 새로운 데이터를 입력했다면 볼수 있다
ISOLATION_SERIALIZABLE	하나의 트랜잭션이 실행된 후에 다른 트랜잭션이 실행

■ Spring JDBC - Transaction

- 격리레벨 이해하기

격리레벨(Isolation level)

1. Read Uncommitted(커밋되지 않은 읽기)

한 사용자가 "A"라는 데이터를 "B"라는 데이터로 변경하는 동안 다른 사용자는 "B" 라는 아직 완료되지 않은 Uncommitted 혹은 Dirty 데이터 "B"를 읽을 수 있다. → 다른 트랜잭션에서 커밋하지 않은 데이터를 읽을 수 있음.

2. Read Committed (커밋된 읽기)

SELECT 문장이 수행되는 동안 해당 데이터에 Shared Lock이 걸립니다. 그러므로, 어떠한 사용자가 A라는 데이터를 B라는 데이터로 변경하는 동안 다른 사용자는 해당 데이터에 접근할 수 없습니다. → 다른 트랜잭션에 의해 커밋된 데이터를 읽을 수 있다.

3. Repeatable Read (반복읽기)

트랜잭션이 완료될 때까지 SELECT 문장이 사용하는 모든 데이터에 Shared Lock이 걸리므로 다른 사용자는 그 영역에 해당되는 데이터에 대한 수정이 불가능합니다. 가령, `Select col1 from A where col1 between 1 and 10`을 수행하였고 이 범위에 해당하는 데이터가 2건이 있는 경우(`col1=1`과 `5`) 다른 사용자가 `col1`이 `1`이 나 `5`인 Row에 대한 UPDATE이 불가능합니다. 하지만, `col1`이 `1`과 `5`를 제외한 나머지 이 범위에 해당하는 Row를 INSERT하는 것이 가능합니다. → 처음에 읽어온 데이터와 두 번째 읽어온 데이터가 동일한 값을 갖는다.

4. Serializable (직렬 혹은 순차기능)

트랜잭션이 완료될 때까지 SELECT 문장이 사용하는 모든 데이터에 Shared Lock이 걸리므로 다른 사용자는 그 영역에 해당 되는 데이터에 대한 수정 및 입력이 불가능합니다. 예를 들어, Repeatable Read의 경우 1에서 10 사이에 해당되는 데이터에 대한 UPADTE이 가능하였습니다. 하지만 이 Level에서는 UPDATE 작업도 허용하지 않습니다.

→ 동일한 데이터에 대해서 동시에 두개 이상의 트랜잭션이 수행될 수 없다.

■ Spring JDBC – Transaction

– 전달행위 알아보기

TransactionDefinition

PROPAGATION_REQUIRED	하나의 트랜잭션이 존재하면 그 트랜잭션을 지원, 없다면 새로운 트랜잭션을 시작. 가장 자주 사용되는 옵션
PROPAGATION_SUPPORTS	하나의 트랜잭션이 존재하면 그 트랜잭션을 지원, 없다면 비-트랜잭션 형태로 수행
PROPAGATION_MANDATORY	이미 트랜잭션이 존재하면 그 트랜잭션을 지원하고, 활성화된 트랜잭션이 없으면 예외를 던진다.
PROPAGATION_REQUIRES_NEW	언제나 새로운 트랜잭션을 수행, 이미 활성화된 트랜잭션이 있다면 일시정지한다.
PROPAGATION_NOT_SUPPORTED	트랜잭션을 실행하지 않고 있다가 트랜잭션이 존재하게 되면 이를 일시중지
PROPAGATION_NEVER	트랜잭션을 실행하지 않고 있다가 트랜잭션이 존재하게 되면 예외를 던진다.
PROPAGATION_NESTED	현재의 트랜잭션이 존재하면 중첩된 트랜잭션 내에서 실행, 없으면 REQUIRED 처럼 작동

■ Spring JDBC – Transaction

- 트랜잭션의 상태와, 트랜잭션의 실질적인 관리자에 대해 알아보기

TransactionStatus

트랜잭션의 상태를 관리하는 역할을 담당. PlatformTransactionManager에서 트랜잭션을 Commit할지 Rollback 할지를 결정하기 위해 사용한다.

```
public interface TransactionStatus extends SavepointManager {  
    boolean isNewTransaction();  
    void setRollbackOnly();  
    boolean isRollbackOnly();  
}
```

PlatformTransactionManager

실질적인 트랜잭션을 실행하는 역할을 한다. commit/rollback작업을 실행하도록 지원

```
public interface PlatformTransactionManager {  
    void commit(TransactionStatus status)...  
    void rollback(TransactionStatus status)..  
}
```

■ Spring JDBC – Transaction

– 트랜잭션 매니저 bean정의하기

TransactionManger 정의

```
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

```
public class TransactionService1 {

    /** 트랜잭션 매니저 */
    private PlatformTransactionManager transactionManager;

    public void setTransactionManager(PlatformTransactionManager
transactionManager) {
        this.transactionManager = transactionManager;
    }

    ...
}///~
```

■ Spring JDBC – Transaction

- 프로그래밍적으로 트랜잭션 처리하기(1)

프로그래밍적으로 트랜잭션 처리 샘플

```
public void addPerson(Person person) {
    DefaultTransactionDefinition transDef =
        new
        DefaultTransactionDefinition(TransactionDefinition.PROPGATION_REQUIRED);
    transDef.setReadOnly(false);
    transDef.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
    TransactionStatus status = transactionManager.getTransaction(transDef);
    Assert.notEmpty(person.getFamilies());
    String newId = this.createNewPersonId();
    person.setId(newId);
    ...
    person.setUpdateDate(createDateTime);
    try {
        ....
        transactionManager.commit(status);
    } catch (Exception e) {
        transactionManager.rollback(status);
        throw new RuntimeException(e);
    }
}
} //:
```

■ Spring JDBC - Transaction

- 프로그래밍적으로 트랜잭션 처리하기(2-1)

프로그래밍적으로 트랜잭션 처리 샘플

```
<bean id="transactionTemplate"  
class="org.springframework.transaction.support.TransactionTemplate">  
    <property name="transactionManager">  
        <ref bean="transactionManager" />  
    </property>  
</bean>
```

```
public class TransactionService4 {  
  
    private PersonDao personDao;  
    private FamilyDao familyDao;  
  
    private TransactionTemplate transactionTemplate;  
  
}///~
```


■ Spring JDBC – Transaction

- 프로그래밍적으로 트랜잭션 처리하기(2-2)

프로그래밍적으로 트랜잭션 처리 샘플

```
public void addPerson(final Person person) {
    final String newId = this.createNewPersonId();
    person.setId(newId);
    String createDateTime = DatetimeUtil.getCurrentDateTime().substring(0, 8);
    person.setCreateDate(createDateTime);
    person.setUpdateDate(createDateTime);
    Object retObject = transactionTemplate.execute(
        new TransactionCallback() {
            public Object doInTransaction(TransactionStatus status) {
                int rv = personDao.insertWithObject(person);
                for(int i=0; i < person.getFamilies().length; i+ + ) {
                    Family fbean = person.getFamilies()[i];
                    fbean.setId(person.getId());
                    rv = addFamily(fbean);
                }
                return null;
            }
        });
}
```

■ Spring JDBC – Transaction

- EJB에서 지원되던 선언적으로 트랜잭션 처리하기

선언적으로 트랜잭션 처리 (2-1)

```

<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
<bean id="nameMatch"
      class="org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource">
    <property name="properties">
        <props><prop key="add*">PROPAGATION_REQUIRED,-Exception</prop></props>
    </property>
</bean>
<bean id="transactionInterceptor"
      class="org.springframework.transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager">
        <ref bean="transactionManager" />
    </property>
    <property name="transactionAttributeSource">
        <ref bean="nameMatch" />
    </property>
</bean>

```

→ 트랜잭션 에트리뷰트

■ Spring JDBC – Transaction

- EJB에서 지원되던 선언적으로 트랜잭션 처리하기

선언적으로 트랜잭션 처리 (2-2)

```
<bean id="autoProxyCreator"
class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator
">

    <property name="beanNames">
        <list>
            <idref bean="transactionService" />
        </list>
    </property>
    <property name="proxyTargetClass">
        <value>true</value>
    </property>
</bean>
```

→ 트랜잭션 적용대상

■ Spring JDBC - Transaction

- 격리레벨과 전파행위, 오류시 롤백 규칙 정의

TransactionAttributes

PROPAGATION, ISOLATION, readOnly, -Exception, +Exception

↑ ↑ ↑ ↑
전달행위(필수) 격리레벨(선택) 읽기전용(선택) Rollback 규칙(선택)

Rollback 규칙(선택)

RuntimeException이 발생하는 경우에는 rollback, CheckedException이 발생하는 경우에는 commit하도록 되어 있다. 마이너스(-)로 시작하는 Exception에 대해서는 무조건 rollback 하도록 한다. 플러스(+)로 시작하는 Exception에 대해서는 commit한다.

■ Spring JDBC – Transaction

- JDK1.5 Annotation 지원

@Transactional

```
@Transactional(propagation=Propagation.REQUIRED, rollbackFor=RuntimeException.class)
public void addPerson(Person person) {
    String createTime = DatetimeUtil.getCurrentDateTime().substring(0, 8);
    Assert.notEmpty(person.getFamilies());
    String newId = this.createNewPersonId();
    person.setId(newId);
    person.setCreateDate(createTime);
    person.setUpdateDate(createTime);

    int rv = this.personDao.insertWithObject(person);
    Assert.isTrue(rv == 1);
    for (int i = 0; i < person.getFamilies().length; i++) {
        Family fbean = person.getFamilies()[i];
        fbean.setId(newId);
        rv = this.addFamily(fbean);
        Assert.isTrue(rv == 1);

    }

}

}
```

■ Spring JDBC – Transaction

– JDK1.5 Annotation 지원

@Transactional

propagation	트랜잭션 전파규칙, Default=REQUIRED
isolation	격리레벨
readOnly	읽기전용여부, default=false
rollbackFor	롤백할 예외 타입 설정. 예) rollbackFor={Exception.class}
noRollbackFor	롤백하지 않을 타입설정. 예) noRollbackFor={BusinessException}
timeout	트랜잭션의 타임아웃 시간, 초단위

작성예시)

```
public class TestService {  
  
    @Transactional(readOnly=false,  
                    rollbackFor={DuplicateOrderIdException.class, AaaException.class})  
    void createOrder(Order order) throws DuplicateOrderIdException ;  
  
    List queryByCriteria(Order criteria);  
}
```

■ Spring JDBC – Transaction

– JDK1.5 Annotation 지원

@Transactional

ApplicationContext.xml 설정

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jee="http://www.springframework.org/schema/jee"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <tx:annotation-driven transaction-manager="transactionManager"/>
```

■ Spring MVC

- 웹개발 환경 이해

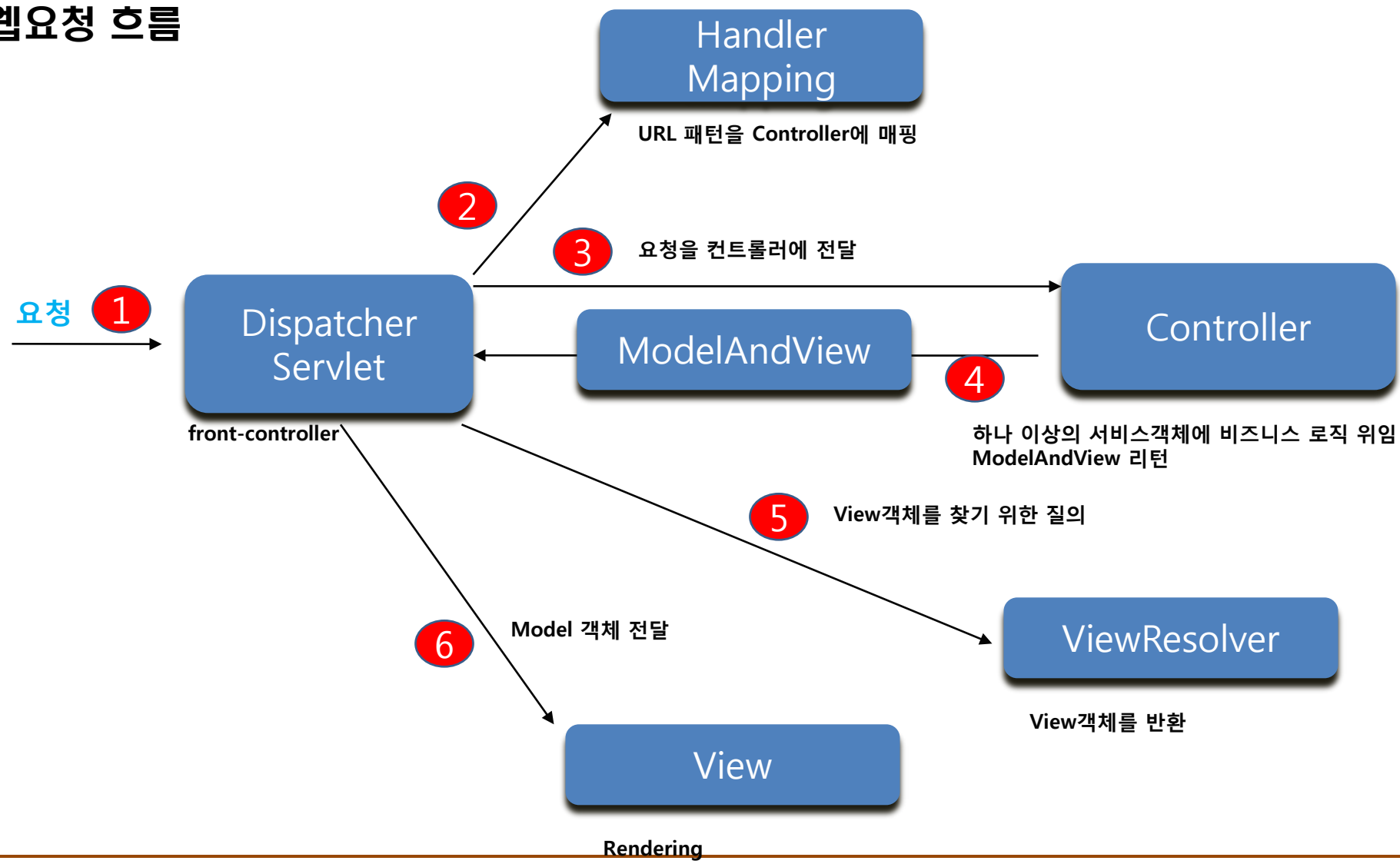
환경설정

1. Eclipse 3.4
2. Tomcat 6.0
3. JDK 1.5 or higher
4. run Tomcat
5. Test Hello World!!

Spring MVC

- Spring Web 요청 흐름 이해하기

웹요청 흐름



■ Spring MVC

- 스프링 MVC 주요 구성 요소 개념 파악

Spring MVC 주요 구성 요소

DispatcherServlet	클라이언트의 요청에 대한 전체 흐름을 제어
HandlerMapping	클라이언트의 요청을 처리할 컨트롤러 결정
Controller	클라이언트의 요청을 처리한다.
ModelAndView	결과정보 및 뷰의 정보
ViewResolver	컨트롤러의 처리결과를 생성할 뷰를 찾아
View	컨트롤러의 처리결과 화면 생성

■ Spring MVC

- web.xml에 MVC 요소 설정

DispatcherServlet 설정

```
<servlet>
  <servlet-name>oraclejava</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/config/edu-02-action.xml
      /WEB-INF/config/tuto/tuto08-message-action.xml
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

■ Spring MVC

- web.xml에 MVC 요소 설정

Character Encoding Filter

```
<filter>
  <filter-name>encodingFilter</filter-name>
  <filter-class>
    org.springframework.web.filter.CharacterEncodingFilter
  </filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>utf-8</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>encodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

■ Spring MVC

- ApplicationContext 이해 (1)

DispatcherServlet의 설정파일에는 스프링 MVC 컴포넌트와 관련된 <bean>정의를 포함

서비스 계층과 데이터 계층에 속하는 빈 역시 DispatcherServlet의 설정파일에 포함할 수 있지만 별도의 설정파일을 두는 것이 좋음. 설정파일들이 모두 로드되도록 하기 위해 컨텍스트 로더를 설정 컨텍스트 로더는 DispatcherServlet이 로드하는 것 이외의 컨텍스트 설정파일을 로드 ContextLoaderListener를 web.xml에 설정

```
<listener>
<listener-class>
    org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
```

설정파일의 위치를 알려주지 않으면 /WEB-INF/applicationContext.xml이라는 스프링 설정 파일을 찾는다. 서블릿 컨텍스트에 contextConfigLocation이라는 파라미터를 설정하면, 컨텍스트 로더가 로드할 수 있는 하나 이상의 스프링 설정 파일을 지정할 수 있다.

■ Spring MVC

- ApplicationContext 이해 (2)

```
<context-param>  
<param-name>contextConfigLocation</param-name>  
<param-value>  
    /WEB-INF/config/base-config.xml  
</param-value>  
</context-param>
```

DispatcherServlet이 bean을 필요로 하는 경우, ContextLoaderListener를 사용하여 사용될 빈을 설정할 수 있다. contextConfigLocation 컨텍스트 파라미터를 이용하여 설정.

■ Spring MVC

- 스프링에서 view를 찾는 방법

ViewResolver(1)

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolve
r">
    <property name="prefix">
        <value></value>
    </property>
    <property name="suffix">
        <value></value>
    </property>
</bean>
```

- Controller의 코드 조각

```
String jspUrl = "/ui/edu02/edu02.jsp";
return new ModelAndView(jspUrl);
```

■ Spring MVC

- 스프링에서 view를 찾는 방법

ViewResolver(2)

JSP파일의 위치를 WEB-INF 아래의 경로로 이동후 설정하기

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolve
r">
    <property name="prefix">
        <value>/WEB-INF/ui/</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>
```

- Controller의 코드 조각

```
String jspUrl = "edu02/edu02";
return new ModelAndView(jspUrl);
```


■ Spring MVC

- 스프링 설정파일 위치를 변경하기

Relative path를 classpath로 변경하기

```
<init-param>  
  <param-name>contextConfigLocation</param-name>  
  <param-value>  
    /WEB-INF/config/edu-02-action.xml  
  </param-value>  
</init-param>
```



```
<init-param>  
  <param-name>contextConfigLocation</param-name>  
  <param-value>  
    classpath:com/education/web/config/edu-02-action.xml  
  </param-value>  
</init-param>
```

→ test 해보기

■ Spring MVC

- 웹요청이 들어오면 해당 요청을 처리할 컨트롤러 구하기

HandlerMapping

DispatcherServlet은 클라이언트 요청이 들어오면 해당 요청을 처리할 컨트롤러를 구하기 위해 HandlerMapping을 이용한다.

1) base-config.xml

```
<bean name="beanNameUrlMapping"  
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">  
</bean>
```

2) edu-03-action.xml

```
<bean name="/beanname.do"  
class="com.education.web.controller.edu03.BeanNameController">  
</bean>
```

3) 웹요청

```
<a href="/beanname.do">lesson 3</a>
```

■ Spring MVC

- 웹요청이 들어오면 해당 요청을 처리할 컨트롤러 구하기

SimpleUrlHandlerMapping

1) base-config.xml

```
<bean id="simpleUrlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <bean class="org.springframework.beans.factory.config.PropertiesFactoryBean">
      <property name="locations">
        <list><value>
          classpath:com/education/web/config/edu-04-urlmap.properties
        </value></list>
      </property>
    </bean>
  </property>
</bean>
```

2) edu-03-action.xml

```
<bean id="simpleurl"
      class="com.education.web.controller.edu04.SimpleUrlController" />
```

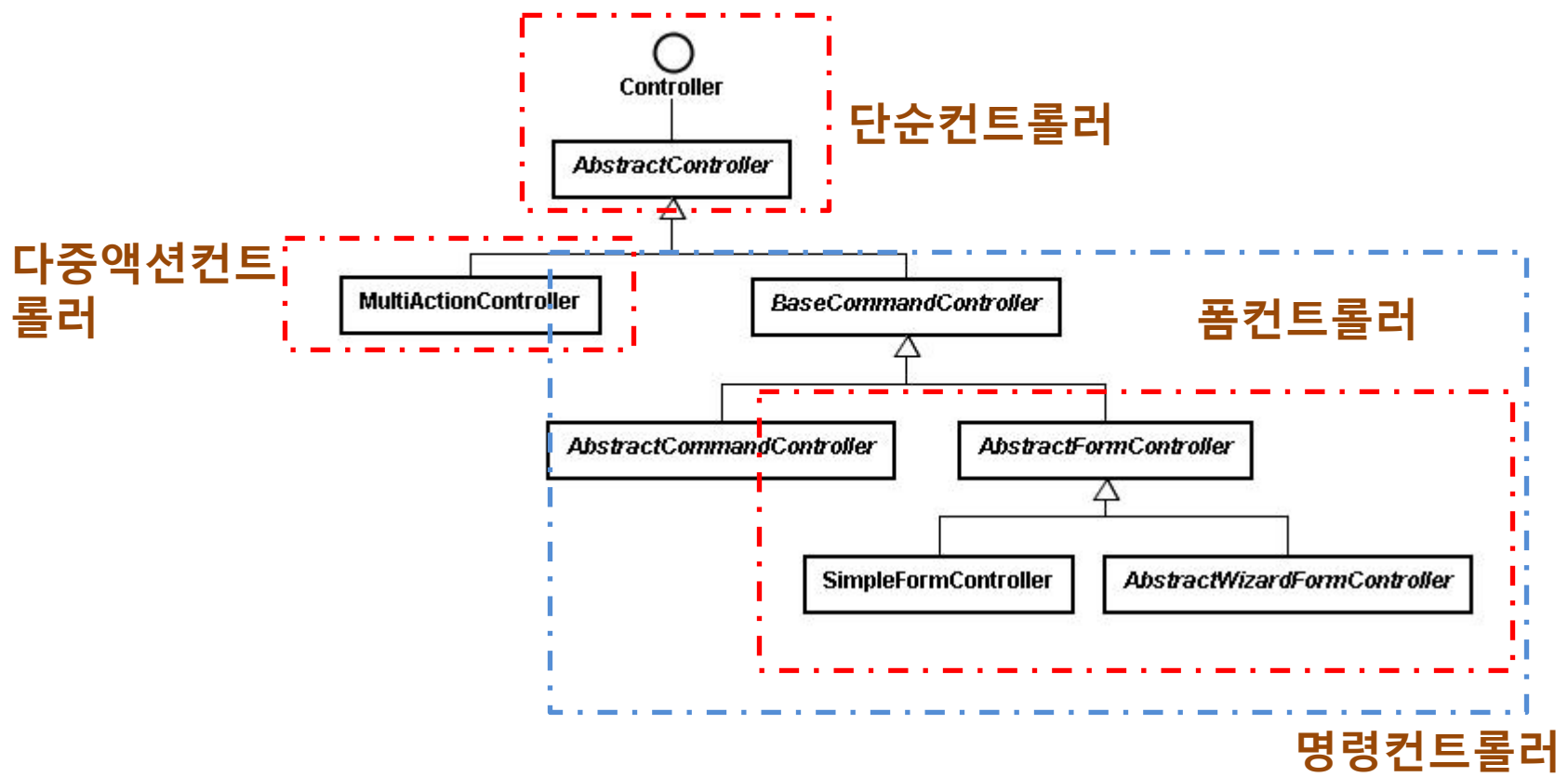
3) 웹요청

```
<a href="/simpleurl.do">lesson 4</a><br/>
```

Spring MVC

- 웹요청이 들어오면 해당 요청을 처리할 컨트롤러 의 종류

컨트롤러 종류



■ Spring MVC

- 다양한 웹요청에 대한 가장 적합한 컨트롤러인 MultiActionController 이해

MultiActionController (1)

- 1) 연관된 기능과 관련된 코드를 하나의 컨트롤러에서 구현
- 2) 컨트롤러가 지나치게 많이 생기지 않도록 함
- 3) 보통 하나의 화면에 하나의 컨트롤러 를 둔다.

Java Code

```
public class FirstController extends MultiActionController{  
    public ModelAndView showPage(HttpServletRequest request,  
        HttpServletResponse response) throws Exception {  
        String jspUrl = "/ui/edu02/edu02.jsp";  
        return new ModelAndView(jspUrl);  
    }///  
  
    public ModelAndView getName(HttpServletRequest request,  
        HttpServletResponse response) throws Exception {  
        String jspUrl = "/ui/edu02/edu02.jsp";  
        request.setAttribute("userName", "김상현");  
        return new ModelAndView(jspUrl);  
    }///  
}
```

}///
Oracle & Java

■ Spring MVC

- 다양한 웹요청에 대한 가장 적합한 컨트롤러인 MultiActionController 이해

MultiActionController (2)

메소드 이름 결정

ParameterMethodNameResolver를 사용하면 편리하게 메소드를 결정할 수 있다.
가장 간편한 방법이고 가장 많이 선호하는 방법

ParameterMethodNameResolver

```
<bean id="controllerMethodNameResolver"  
      class="org.springframework.web.servlet.mvc.multiaction.  
          ParameterMethodNameResolver">  
    <property name="paramName">  
        <value>request</value>  
    </property>  
    <property name="defaultMethodName">  
        <value>doDefault</value>  
    </property>  
</bean>
```

■ Spring MVC

- 컨트롤러의 처리결과 → 데이터와 ViewResolver에게 view를 결정하도록 하는 view 정보

ModelAndView

```
ModelAndView mav = new ModelAndView();  
mav.setViewName("/jsp/list.jsp");  
mav.addObject("userName", "김상현");  
mav.addObject("customerList", customerList);  
mav.addObject("customer", customerObj);  
return mav;
```

```
return new ModelAndView("/jsp/list.jsp");
```

```
request.setAttribute("userName", "김상현");  
return new ModelAndView("/jsp/list.jsp");
```

■ Spring MVC

- 샘플을 통해서 배우기

Login Sample

Interceptor

FileUpload

FileDownload

Ajax 처리