

# 데이터 베이스 액세스

강사 : 강병준

# 데이터 액세스 층의 역할

1. 데이터베이스 접속과 sql을 비즈니스로직과 분리하여 소스코드를 간결하게 하고 유지관리를 쉽게 함
2. DAO패턴은 데이터베이스 접속과 SQL발행과 같은 데이터엑세스 처리를DAO라고 불리는 오브젝트로 분리하는 패턴
3. 자바의 데이터베이스엑세스와 스프링의 기능
  - Spring JDBC, Hibernate연결, JPA연결, MyBatis(iBatis), JDO연결방법등이 있다
4. 장점 : 데이터 엑세스처리의 간결, 스프링이 제공하는 범용적이고 체계적인 데이터 엑세스 예외처리, 스프링의 트랜잭션 기능 이용

**Spring JDBC를 사용하면 연결해제 처리를 해결한다.**

- JdbcTemplate
- NamedParameterJdbcTemplate
- SimpleJdbcTemplate

# Spring JDBC역할

1. Connection, PreparedStatement, ResultSet 등의 Open / Close 를 일일이 작성하지 않아도 됨.
2. Close 를 하지 않아 메모리 누수가 발생할 일이 없음
3. JDBC의 PreparedStatement 를 사용함으로써 SQL Injection 의 위험이 없음
4. 높은 추상화 수준으로 쿼리가 간결해짐. -> 유지보수의 이점이 발생함.
5. 높은 추상화 수준으로 수정/확장이 편리해짐.
6. Connection-Pool을 지원하기 때문에, 쿼리의 속도가 빠름.
7. 동시성 문제 발생할 확률이 줄어듦.
8. 간헐적 병목현상이 발생할 수 있음.
9. 업무단위의 Transaction 이 지원됨.
10. Connection 을 분리함으로써 업무단위의 Transaction 이 가능함.
11. spring-jdbc 가 필요함.

# Spring JDBC역할

1. 일반 JDBC 프로그래밍의 예외는 SQLException 하나로 처리되지만 스프링은 데이터베이스와 관련된 예외를 처리하기 위해서 SQLException이 발생하면 스프링이 제공하는 예외 클래스 중에서 알맞은 예외 클래스로 변환해서 발생시킵니다.
2. 스프링이 제공하는 Exception은 DataAccessException의 하위 클래스 입니다.
3. 스프링이 발생시키는 Exception은 전부 RuntimeException이어서 예외 처리하는 코드를 직접 입력하지 않아도 됩니다.
4. 스프링 데이터 베이스의 예외 클래스
  - 1) DataAccessException의 하위 클래스: NonTransientDataAccessException, TransientDataAccessException, RecoverableDataAccessException
  - 2) NonTransientDataAccessException의 하위 클래스: DataIntegrityViolationException, DuplicateKeyException, DataRetrievalFailureException, PermissionDeniedDataAccessException, InvalidDataAccessResourceUsageException, BadSqlGrammarException, TypeMismatchDataAccessException
  - 3) TransientDataAccessException의 하위 클래스: TransientDataAccessResourceException, ConcurrencyFailureException, OptimisticLockingFailureException, PessimisticLockingFailureException

# 오라클 연동

❖오라클을 사용하기 위해서 오라클의 의존성을 pom.xml 파일에 추가

```
<repositories>
  <repository>
    <id>codeIds</id>
    <url>https://code.lds.org/nexus/content/groups/main-repo</url>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc6</artifactId>
    <version>11.2.0.3</version>
  </dependency>
</dependencies>
```

**<org.springframework-version> 태그에 주의 해야 합니다.**  
이 태그가 이후에 사용되는 스프링 모듈의 버전을 의미하게 됩니다.  
프로젝트 종류에 따라 다르게 이름이 다르게 설정되어 있습니다.

# Test클래스 사용하여 DB연동 확인

pom.xml 파일에 테스트 담당하는 junit 테플릿 추가

```
<!-- Test -->
```

```
<dependency>
```

```
  <groupId>junit</groupId>
```

```
  <artifactId>junit</artifactId>
```

```
  <version>4.11</version>
```

```
  <scope>test</scope>
```

```
</dependency>
```

```
<dependency>
```

```
  <groupId>org.springframework</groupId>
```

```
  <artifactId>spring-test</artifactId>
```

```
  <version>${spring-framework.version}</version>
```

```
</dependency>
```

@Inject를 사용할 때

```
<!-- @Inject -->
```

```
  <dependency>
```

```
    <groupId>javax.inject</groupId>
```

```
    <artifactId>javax.inject</artifactId>
```

```
    <version>1</version>
```

```
</dependency>
```

# Test클래스 사용하여 DB연동 확인

```
public class OracleConnectionTest {  
    private static final String DRIVER = "oracle.jdbc.driver.OracleDriver";  
    private static final String URL = "jdbc:oracle:thin:@127.0.0.1:1521:xe";  
    private static final String USER = "scott";  
    private static final String PW = "tiger";  
    @Test  
    public void testConnection() throws Exception{  
        Class.forName(DRIVER);  
        try(  
            Connection con = DriverManager.getConnection(URL, USER, PW)){  
                System.out.println(con);  
            }catch(Exception e){  
                e.printStackTrace();  
            }  
        }  
    }  
}
```



# DataSource

1. jdbc를 이용해 데이터베이스를 사용하려면 Connection 타입의 DB 연결 오브젝트가 필요합니다.
2. Spring 에서는 Connection을 만들때 DataSource를 이용하도록 강제합니다.
3. 스프링에서는 Connection을 만들기 위한 DataSource를 하나의 독립된 bean으로 생성하는 것을 권장합니다.
4. 종류는 SimpleDriverDataSource와 SingleConnectionDataSource가 있는데 첫번째는 getConnection을 호출할 때마다 매번 DBConnection을 새로 만들고 풀을 관리하지 않으며 두번째는 하나의 물리적인 DBConnction만 만들고 이를 계속 사용하는 형태입니다.
5. 위 2개의 클래스보다는 자동으로 해제를 수행해주는 DriverManagerDataSource나 BasicDataSource를 많이 이용
6. DataSource를 생성하는 방법
  - 1) Connection Pool을 이용하는 방법
  - 2) JNDI를 이용하는 방법
  - 3) DriverManager를 이용하는 방법



## DriverManager를 이용한 DataSource 설정

- ❖ `org.springframework.jdbc.datasource.DriverManagerDataSource`
  - `driverClassName`
  - `url`
  - `username`
  - `Password`

# DataSource

## DBCP(Jakarta Commons Database Connection Pool) API 커넥션을 이용한 설정

1. org.apache.commons.dbcp.BasicDataSource 클래스 이용
2. 주요 프로퍼티
  - 1) initialSize: 초기 풀에 생성되는 커넥션 개수
  - 2) maxActive: 커넥션 풀이 제공하는 최대 커넥션 개수
  - 3) maxIdle: 사용되지 않고 풀에 저장될 수 있는 최대 커넥션 개수
  - 4) minIdle: 사용되지 않고 풀에 저장될 수 있는 최소 커넥션 개수
  - 5) maxWait: 풀에 커넥션이 존재하지 않을 때 대기하는 시간으로 1/1000초 단위로 설정하면 -1이면 무한 대기
  - 6) minEvictableIdleTimeMillis: 이 시간 동안 비 활성화 상태이면 커넥션 추출
  - 7) timeBetweenEvictionRunsMillis: 커넥션 추출 주기
3. <http://commons.apache.org/proper/commons-dbcp/configuration.html> 에서 확인

# DataSource

## c3p0 API를 이용한 설정

### 1. **com.mchange.v2.c3p0.ComboPooledDataSource** 클래스 이용

```
<bean destroy-method="close"
class="com.mchange.v2.c3p0.ComboPooledDataSource" id="dataSource">
<property value="데이터베이스 드라이버 이름" name="driverClass"/>
<property value="데이터베이스 url" name="jdbcUrl"/>
<property value="접속할 아이디 " name=" user " />
<property value=" 접속할 비번" name="password"/>
</bean>
```

### 1. <http://www.mchange.com/projects/c3p0/> 에서 프로퍼티 확인

### 2. 메이븐 종속성

```
<dependency>
    <groupId>com.mchange</groupId>
    <artifactId>c3p0</artifactId>
    <version>0.9.5.2</version>
</dependency>
```

# 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
```

## spring-db.xml      템플릿 클래스의 bean등록

```
<?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:aop="http://www.springframework.org/schema/aop"  
    xmlns:tx="http://www.springframework.org/schema/tx"  
    xmlns:jee="http://www.springframework.org/schema/jee"  
    xmlns:context="http://www.springframework.org/schema/context"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd  
http://www.springframework.org/schema/context  
http://www.springframework.org/schema/context/spring-context-3.0.xsd  
http://www.springframework.org/schema/jee  
http://www.springframework.org/schema/jee/spring-jee-3.0.xsd
```

<http://www.springframework.org/schema/tx>

<http://www.springframework.org/schema/tx/spring-tx-3.0.xsd>

<http://www.springframework.org/schema/aop>

<http://www.springframework.org/schema/aop/spring-aop-3.0.xsd>>

```
<context:property-placeholder location="jdbc.properties"/>
```

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
```

```
    <constructor-arg ref="dataSource" />
```

```
</bean>
```

```
<bean
```

```
    class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
```

```
    <constructor-arg ref="dataSource" />
```

```
</bean>
```

```
<!-- <bean id="dataSource" -->
```

```
<!--     class="org.springframework.jdbc.datasource.DriverManagerDataSource" -->
```

```
<!--     <property name="driverClassName" value="${jdbc.driverClassName}" /> -->
```

```
<!--     <property name="url" value="${jdbc.url}" /> -->
```

```
<!--     <property name="username" value="${jdbc.username}" /> -->
```

```
<!--     <property name="password" value="${jdbc.password}" /> -->
```

```
<!-- </bean> -->
```

```
<!-- <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" -->
```

```
<!--     destroy-method="close" -->
```

```
<!--     <property name="driverClassName" value="${jdbc.driverClassName}" /> -->
```

```
<!--     <property name="url" value="${jdbc.url}" /> -->
```

```
<!--     <property name="username" value="${jdbc.username}" /> -->
```

```
<!--     <property name="password" value="${jdbc.password}" /> -->
```

```
<!--     <property name="maxActive" value="${jdbc.maxPoolSize}" /> -->
```

```
<!-- </bean> -->
```

```
<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>
</beans>
```

## log4j.properties

```
log4j.rootLogger=INFO, CA
log4j.appender.CA=org.apache.log4j.ConsoleAppender
log4j.appender.CA.layout=org.apache.log4j.PatternLayout
log4j.appender.CA.layout.ConversionPattern=[%p:%c{1}] %m%n
```

## C3p0모듈의 ComboPooledDataSource클래스

: 커넥션 풀 기능을 제공하는 DataSource구현 클래스

*initialPoolSize*

- 초기의 커넥션 풀의 크기, 기본값은 3

*maxPoolSize*

- 커넥션풀의 최대크기, 기본값은 15

*minPollSize*

- 커넥션풀의 최소크기, 기본값은 3

*maxIdleTime*

- 지정한 시간동안 사용되지 않은 커넥션 제거한다. 단위는 초 단위이며 값이 0일 때는 제거하지 않는다. 기본값은 0

*checkoutTimeout*

- 풀에서 커넥션을 가져올 때 대기 시간, 1/1000초 0은 무한대기  
지정시간 동안 풀에서 커넥션을 가져오지 못할 때에는 SQLException 발생, 기본값은 0

*idle ConnectionTestPeriod*

- 풀속에 있는 커넥션의 테스트 주기, 단위는 초이며  
0인 경우검사하지 않는다. 기본 값은 0



# Test클래스 사용하여 DB연동 확인

## pom.xml 파일에 테스트 담당하는 test 및 inject 테플릿 추가

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${org.springframework.version}</version>
</dependency>
<dependency>
  <groupId>c3p0</groupId>
  <artifactId>c3p0</artifactId>
  <version>0.9.1.2</version>
</dependency>
<!-- @Inject -->
<dependency>
  <groupId>javax.inject</groupId>
  <artifactId>javax.inject</artifactId>
  <version>1</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>${org.springframework.version}</version>
</dependency>
```

# Test클래스 사용하여 DB연동 확인

```
package ch03;
import java.sql.Connection;
import javax.inject.Inject;
import javax.sql.DataSource;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"file:src/main/java/spring-db.xml"})
public class OracleConnectionTest {
    @Inject
    private DataSource ds;
    @Test
    public void testConnection() throws Exception{
        try{
            Connection con = ds.getConnection();
            System.out.println(con);
        }catch(Exception e){ System.out.println(e.getMessage()); }
    }
}
```

# Pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>sample</groupId>
  <artifactId>springjdbc</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <properties>
    <org.springframework.version>4.1.1.RELEASE</org.springframework.version>
  </properties>
  <repositories>
    <repository>
      <id>codeids</id>
      <url>https://code.ids.org/nexus/content/groups/main-repo</url>
    </repository>
  </repositories>
```

```
<dependencies>    <!-- oracle -->
  <dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc6</artifactId> <version>11.2.0.3</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${org.springframework.version}</version>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.16</version>
  </dependency>
  <dependency>
    <groupId>c3p0</groupId>
    <artifactId>c3p0</artifactId>
    <version>0.9.1.2</version>
  </dependency>
</dependencies>
</project>
```

# 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, 예외처리 등을 맡는다
  - 콜백은 질의문 생성, 파라미터 바인딩, 결과집합 마샬링 등 처리
- ==> 개발자는 로직에만 집중!!

## marshal 이란

프로그래밍에서 마샬링은 RPC, RMI 등에서 클라이언트가 원격지(서로 다른 프로세스)의 메서드를 호출할 때 서버에 넘겨지는 인자, 원격지 함수의 리턴 값들을 프로그래밍 인터페이스에 맞도록 그 데이터를 조직화하고, 미리 정해진 다른 형식으로 변환하는 과정을 말한다.

XML 로 마샬링, Byte 스트림으로 마샬링 등 데이터 교환할 때 어떠한 정해진 표준에 맞게 해당 데이터를 가공하는 것을 마샬링, 언마샬링 이라고 한다.

클라이언트에서 마샬링된 데이터를 서버에 전달하게 되면,  
서버에서는 그 데이터를 언마샬링하여 사용함으로써  
원격지(다른 프로세스)간의 데이터 사용이 가능하게 된다

# 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 setJdbcTemplate(JdbcTemplate jdbcTemplate) {  
        this.jdbc = jdbcTemplate;  
    }  
}  
//~
```

# SimpleJdbcTemplate

- JdbcTemplate을 기능을 향상한 것
- 생성 : DataSource를 DI로 받아야 함, 싱글톤

```
SimpleJdbcTemplate temp = new SimpleJdbcTemplate(dataSource);
```

- 일반적인 사용법

```
public class MemberDao {  
    SimpleJdbcTemplate template;  
    public void setDataSource(DataSource dataSource) {  
        this.template = new SimpleJdbcTemplate(dataSource);  
    }  
}
```

@Autowired

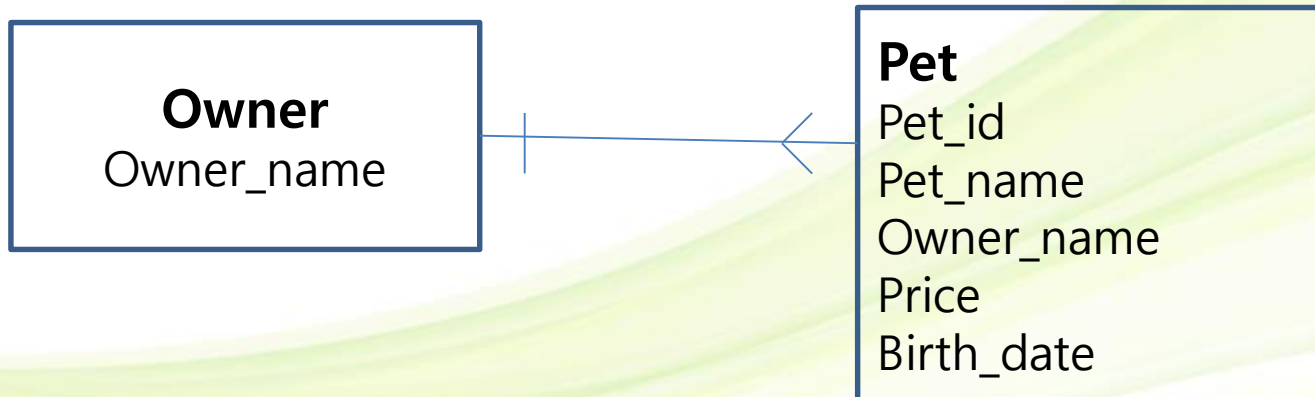
```
public void init(DataSource dataSource) {  
    this.template = new SimpleJdbcTemplate(dataSource);  
}
```



# Table 생성 및 데이터 입력

```
create table owner (  
    ownerName varchar(20) primary key  
);  
create table pet (    petId number(10) primary key,  
    petName varchar2(20),  
    ownerName varchar2(20) references owner(ownerName),  
    price number(10), birthDate date );
```

```
insert into owner values ('aa');  
insert into owner values ('kk');  
insert into owner values ('부산동생');  
insert into pet values (1, 'tom' , 'aa' , 10000 , sysdate);  
insert into pet values (2, 'jerry' , 'kk' , 10000 , sysdate);  
insert into pet values (3, 'mary' , 'kk' , 10000 , sysdate);
```



# 도메인 클래스 Owner.java

```
package sample.biz.domain;
import java.util.ArrayList;
import java.util.List;
public class Owner {
    private String ownerName;
    private List<Pet> petList = new ArrayList<Pet>();
    public String getOwnerName() {
        return ownerName;
    }
    public void setOwnerName(String ownerName) {
        this.ownerName = ownerName;
    }
    public List<Pet> getPetList() {
        return petList;
    }
    public void setPetList(List<Pet> petList) {
        this.petList = petList;
    }
}
```

# Pet.java

```
package sample.biz.domain;
import java.util.Date;
public class Pet {
    private int petId;    private String petName;    private String ownerName;
    private int price;    private Date birthDate;
    public Date getBirthDate() {    return birthDate;    }
    public void setBirthDate(Date birthDate) {    this.birthDate = birthDate;    }
    public int getPrice() { return price; }
    public void setPrice(int price) { this.price = price; }
    public Pet() {}
    public Pet(int petId, String petName) {
        this.petId = petId;
        this.petName = petName;
    }
    public int getPetId() { return petId; }
    public void setPetId(int petId) { this.petId = petId; }
    public String getPetName() { return petName; }
    public void setPetName(String petName) { this.petName = petName; }
    public String getOwnerName() {    return ownerName;    }
    public void setOwnerName(String ownerName) {
        this.ownerName = ownerName;
    }
}
```

# JdbcTemplate

## org.springframework.jdbc.core.JdbcTemplate 클래스

1. 생성자에 DataSource를 매개변수로 대입해서 생성
2. Select 수행
  - 1) List query(String sql, RowMapper rowMapper)
  - 2) List query(String sql, Object[]args, RowMapper rowMapper)
  - 3) Object queryForObject(String sql, RowMapper rowMapper)
  - 4) Object queryForObject(String sql, Object[]args, RowMapper rowMapper)
  - 5) public int queryForInt(String sql, Object[]args)
  - 6) public int queryForInt(String sql, Object[]args)
  - 7) Object [] args는 sql에 ?를 사용했을 때의 실제 바인딩되는 데이터 배열
3. RowMapper는 읽어온 데이터를 반환하기 위한 템플릿으로서 Object mapRow(ResultSet rs, int rowNum)throws SQLException 메서드를 재정의해서 읽어온 데이터를 어떻게 리턴한 것인지를 결정합니다.

## mapRow(ResultSet rs, int rownum) 재정의

```
public 리턴타입 mapRow(ResultSet rs, int rownum) throws SQLException {  
    rs는 java.sql 패키지의 ResultSet 객체  
    rownum은 행번호  
    리턴 할 클래스 타입의 객체를 생성 – HashMap 또는 Dto 클래스 타입  
    rs를 이용해서 데이터를 읽어서 저장한 후 객체를 리턴  
}
```

# JdbcTemplate

## Select문 – 도메인으로 변환하지 않을 때

도메인을 변환하지 않을 때란 단순한 값이나 특정한 컬럼값을 가져올 때

### 1) 수치값 : queryForInt, queryForLong

```
int count = jdbcTemplate.queryForObject("select count(*) from pet", Integer.class);
```

```
int count = jdbcTemplate.queryForObject("select count(*) from pet where  
owner_name = ?", Integer.class, ownerName);
```

### 2) 문자열이나 날짜형 : queryForObject

```
String petName = jdbcTemplate.queryForObject("select pet_name from pet  
where pet_id = ?", String.class, id);
```

두번째 인수는 취득결과 오브젝트의 Class오브젝트

```
Date birthDate = jdbcTemplate.queryForObject("select birth_date from pet  
where pet_id = ?", Date.class, id);
```

### 3) 한 레코드 값을 가져올 때 : queryForMap

```
Map<String, Object> pet = jdbcTemplate.queryForMap("select * from pet  
where pet_id = ?", id);
```

### 4) 여러 개의 데이터를 가져올 때 : queryForList

```
List<Map<String, Object>> petList = jdbcTemplate.queryForList("select  
* from pet where owner_name = ?", ownerName);
```

```
List<Integer> priceList = jdbcTemplate.queryForList("select price  
from pet where owner_name = ?", Integer.class, ownerName);
```

두번째 인수는 List에 포함될 오브젝트의 Class오브젝트

# JdbcTemplate

## Select문 – 도메인으로 변환할 때

```
Pet pet = jdbcTemplate.queryForObject(
    "SELECT * FROM PET WHERE PET_ID=?",
    , new RowMapper<Pet>() {
        public Pet mapRow(ResultSet rs, int rowNum) throws
SQLException { Pet p = new Pet();
            p.setPetId(rs.getInt("PET_ID"));
            p.setPetName(rs.getString("PET_NAME"));
            p.setOwnerName(rs.getString("OWNER_NAME"));
            p.setPrice(rs.getInt("PRICE"));
            p.setBirthDate(rs.getDate("BIRTH_DATE"));
            return p;
        }}
    , id);
```

### 1) 인수

- . 첫 번째 인수 : select문
- . 두 번째 인수 : 도메인으로 처리하는 클래스의 오브젝트
- . 세 번째 인수 : select문의 파라메타

2) RowMapper ; 스프링이 제공하는 인터페이스로 mapRow라는 추상메소드 정의  
mapRow를 구현한 클래스 작성과 그 오브젝트를 queryForObject의 인수로 건넨  
두 번째 부분처럼 구현한 것을 익명클래스라고 함



# JdbcTemplate

## 3) 익명클래스 사용하지 않을 때

```
class MyRowMapper implements RowMapper<Pet> {  
    public Pet mapRow(ResultSet rs, int rowNum) throws SQLException {  
        Pet p = new Pet();  
        p.setPetId(rs.getInt("PET_ID"));  
        p.setPetName(rs.getString("PET_NAME"));  
        p.setOwnerName(rs.getString("OWNER_NAME"));  
        p.setPrice(rs.getInt("PRICE"));  
        p.setBirthDate(rs.getDate("BIRTH_DATE"));  
        return p;  
    }  
}  
  
Pet pet = jdbcTemplate.queryForObject( " SELECT * FROM PET WHERE  
    PET_ID=?", new MyRowMapper(), id);
```

# JdbcTemplate

## Query 메소드

```
List<Pet> petList = jdbcTemplate.query(
    " SELECT * FROM PET WHERE OWNER_NAME=?"
    , new RowMapper<Pet>() {
        public Pet mapRow(ResultSet rs, int rowNum) throws SQLException {
            Pet p = new Pet();      p.setPetId(rs.getInt("PET_ID"));
            p.setPetName(rs.getString("PET_NAME"));
            p.setOwnerName(rs.getString("OWNER_NAME"));
            p.setPrice(rs.getInt("PRICE"));
            p.setBirthDate(rs.getDate("BIRTH_DATE"));
            return p;
        }
    }
    , ownerName);
```

RowMapper에서 하는 도메인 변환처리도 자동으로 하기 위해서는 스프링에서 제공하는 BeanPropertyRowMapper를 사용

```
Pet pet = jdbcTemplate.queryForObject("select * from Pet where pet_id = ?"
    , new BeanPropertyRowMapper<Pet>(Pet.class)
    , id);
```

# JdbcTemplate

테이블을 조인한 여러 레코드를 가져올 때 ; ResultSetExtractor를 사용

```
Owner owner = jdbcTemplate.query( " SELECT * FROM OWNER O, PET P
where O.OWNER_NAME=P.OWNER_NAME and O.OWNER_NAME=?"
    , new ResultSetExtractor<Owner>() {
        public Owner extractData(ResultSet rs) throws SQLException,
        DataAccessException {
            if (!rs.next()) { return null; }
            Owner owner = new Owner();
            owner.setOwnerName(rs.getString("OWNER_NAME"));
            do { Pet pet = new Pet();
pet.setPetId(rs.getInt("PET_ID"));
            pet.setPetName(rs.getString("PET_NAME"));
            pet.setOwnerName(rs.getString("OWNER_NAME"));
            pet.setPrice(rs.getInt("PRICE"));
            pet.setBirthDate(rs.getDate("BIRTH_DATE"));
            owner.getPetList().add(pet);
            } while(rs.next());
            return owner;
        }
    }
    , ownerName);
```

# JdbcTemplate

## Insert/update/delete 메소드

- 1) jdbcTemplate.update( "INSERT INTO PET (PET\_ID, PET\_NAME, OWNER\_NAME, PRICE, BIRTH\_DATE) VALUES (?, ?, ?, ?, ?)"  
 , pet.getPetId(), pet.getPetName(), pet.getOwnerName(), pet.getPrice(),  
 pet.getBirthDate());
- 2) jdbcTemplate.update( "UPDATE PET SET PET\_NAME=?, OWNER\_NAME=?,  
 PRICE=?, BIRTH\_DATE=? WHERE PET\_ID=?"  
 , pet.getPetName(), pet.getOwnerName(), pet.getPrice(), pet.getBirthDate(),  
 pet.getPetId());
- 3) jdbcTemplate.update("DELETE FROM PET WHERE PET\_ID=?", pet.getPetId());

# NamedParameterJdbcTemplate

1. `org.springframework.jdbc.namedparam.NamedParameterJdbcTemplate` 클래스
2. `JdbcTemplate` 클래스와 거의 동일한 메서드를 소유하고 있으며 바인딩되는 매개변수에 `Object` 배열 대신에 `Map` 또는 `MapSqlParameterSource` 객체를 이용해서 파라미터를 설정하는 것이 다릅니다.
3. 또한 ? 대신에 실제 키 이름을 기재하는데 이름을 기재할 때는 **:키이름**의 형태로 기재해야 하며 이름이 `Map`에서의 키가 되어야 합니다.

# NamedParameterJdbcTemplate

1) 메소드체인으로 기술

```
" INSERT INTO PET (PET_ID, PET_NAME, OWNER_NAME, PRICE, BIRTH_DATE)" +  
    " VALUES  
(:PET_ID, :PET_NAME, :OWNER_NAME, :PRICE, :BIRTH_DATE)"  
    , new MapSqlParameterSource()  
    .addValue("PET_ID", pet.getPetId())  
    .addValue("PET_NAME", pet.getPetName())  
    .addValue("OWNER_NAME", pet.getOwnerName())  
    .addValue("PRICE", pet.getPrice())  
    .addValue("BIRTH_DATE", pet.getBirthDate())  
);
```

# NamedParameterJdbcTemplate

2) 메소드체인을 사용하지 않을 때

```
MapSqlParameterSource map2 = new MapSqlParameterSource();
map2.addValue("PET_ID", pet.getPetId());
map2.addValue("PET_NAME", pet.getPetName());
map2.addValue("OWNER_NAME", pet.getOwnerName());
map2.addValue("PRICE", pet.getPrice());
map2.addValue("BIRTH_DATE", pet.getBirthDate());
npJdbcTemplate.update(
    " INSERT INTO PET (PET_ID, PET_NAME, OWNER_NAME, PRICE, " +
    " BIRTH_DATE) values (:PET_ID, :PET_NAME, :OWNER_NAME, " +
    " :PRICE, :BIRTH_DATE)"
    ,map2
);
```

3) 도메인으로부터 파라미터로의 변환을 자동화할 때

```
BeanPropertySqlParameterSource beanProps =
new BeanPropertySqlParameterSource(pet);
npJdbcTemplate.update(
    " INSERT INTO PET (PET_ID, PET_NAME, OWNER_NAME, PRICE,
    BIRTH_DATE)" +
    " VALUES (:petId, :petName, :ownerName, :price, :birthDate)" ,
    beanProps
);
```



# NamedParameterJdbcTemplate

## 4) 메소드 체인으로 기술하지 않을 때

```
MapSqlParameterSource map2 = new MapSqlParameterSource();
map2.addValue("PET_ID", pet.getPetId());
map2.addValue("PET_NAME", pet.getPetName());
map2.addValue("OWNER_NAME", pet.getOwnerName());
map2.addValue("PRICE", pet.getPrice());
map2.addValue("BIRTH_DATE", pet.getBirthDate());
npJdbcTemplate.update(
    " INSERT INTO PET (PET_ID, PET_NAME, OWNER_NAME, PRICE,
BIRTH_DATE)" +
    " VALUES (:PET_ID, :PET_NAME, :OWNER_NAME, :PRICE, :BIRTH_DATE)"
    ,map2
);
```

## 스프링에서 사용하는 데이터 소스

데이터소스는 접속 Object인 Connection factory로서 커넥션 풀을 사용

- 1) SingleConnectionDataSource : 커넥션이 하나임을 보장, 닫지않고 다시 사용
- 2) DriverManagerDataSource : 커넥션을 요청할 때마다 오브젝트 생성
- 3) 서드파트 제공
  - . DBCP ; 아파치 제공
  - . C3p0 ; Machinery For Oracle사 제공

# ExecuteSqlMain.java

```
package sample;
import java.sql.*;                import java.util.*;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;
import sample.biz.domain.Owner;
import sample.biz.domain.Pet;
public class ExecuteSqlMain {
    public static void main(String[] args) {
        // Spring의 컨테이너를 생성
        ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-db.xml");
        // JdbcTemplate과 NamedParameterJdbcTemplate의 오브젝트를 취득
        JdbcTemplate jdbcTemplate = ctx.getBean(JdbcTemplate.class);
        NamedParameterJdbcTemplate npJdbcTemplate =
        ctx.getBean(NamedParameterJdbcTemplate.class);
    }
}
```

// SELECT문 ~ 도메인으로 변환하지 않는 경우

// queryForInt 메소드

```
int count = jdbcTemplate.queryForInt("SELECT COUNT(*) FROM PET");
```

```
System.out.println(count);
```

```
String ownerName = "kk";
```

```
count = jdbcTemplate.queryForInt(
```

```
    "SELECT COUNT(*) FROM PET WHERE OWNER_NAME=?", ownerName);
```

```
System.out.println(count);
```

// queryForObject 메소드

```
int id = 1;
```

```
String petName = jdbcTemplate.queryForObject(
```

```
    "SELECT PET_NAME FROM PET WHERE PET_ID=?", String.class, id);
```

```
System.out.println(petName);
```

```
Date birthDate = jdbcTemplate.queryForObject(
```

```
    "SELECT BIRTH_DATE FROM PET WHERE PET_ID=?", Date.class, id);
```

```
System.out.println(birthDate);
```

// queryForMap 메소드

```
Map<String, Object> map = jdbcTemplate.queryForMap(
```

```
    "SELECT * FROM PET WHERE PET_ID=?", id);
```

```
System.out.println(map.get("PET_NAME"));
```

```
System.out.println(map.get("OWNER_NAME"));
```

// queryForList 메소드

```
List<Map<String, Object>> mapList = jdbcTemplate.queryForList(
```

```
    " SELECT * FROM PET WHERE OWNER_NAME=?", ownerName);
```

```
System.out.println(mapList.get(0).get("PET_NAME"));
```

```
System.out.println(mapList.get(0).get("OWNER_NAME"));
```

```
List<Integer> priceList = jdbcTemplate.queryForList(
    "SELECT PRICE FROM PET WHERE OWNER_NAME=?", Integer.class, ownerName);
System.out.println(priceList.get(0));
// SELECT문 ~ 메소드로 변환하는 경우
// queryForObject 메소드
// RowMapper를 익명 클래스로 할 때
Pet pet = jdbcTemplate.queryForObject(
    "SELECT * FROM PET WHERE PET_ID=?",
    , new RowMapper<Pet>() {
        public Pet mapRow(ResultSet rs, int rowNum) throws SQLException {
            Pet p = new Pet();
            p.setPetId(rs.getInt("PET_ID"));
            p.setPetName(rs.getString("PET_NAME"));
            p.setOwnerName(rs.getString("OWNER_NAME"));
            p.setPrice(rs.getInt("PRICE"));
            p.setBirthDate(rs.getDate("BIRTH_DATE"));
            return p;
        }
    }, id);
System.out.println(pet.getPetName());
System.out.println(pet.getOwnerName());
// RowMapper를 익명 클래스로 하지 않을 때
```

```

class MyRowMapper implements RowMapper<Pet> {
    public Pet mapRow(ResultSet rs, int rowNum) throws SQLException {
        Pet p = new Pet();
        p.setPetId(rs.getInt("PET_ID"));
        p.setPetName(rs.getString("PET_NAME"));
        p.setOwnerName(rs.getString("OWNER_NAME"));
        p.setPrice(rs.getInt("PRICE"));
        p.setBirthDate(rs.getDate("BIRTH_DATE"));
        return p;
    }
}

pet = jdbcTemplate.queryForObject( " SELECT * FROM PET WHERE PET_ID=?"
    ,new MyRowMapper() ,id);
System.out.println(pet.getPetName());    System.out.println(pet.getOwnerName());
// query 메소드
List<Pet> petList = jdbcTemplate.query(
    " SELECT * FROM PET WHERE OWNER_NAME=?"
    , new RowMapper<Pet>() {
        public Pet mapRow(ResultSet rs, int rowNum) throws SQLException {
            Pet p = new Pet();    p.setPetId(rs.getInt("PET_ID"));
            p.setPetName(rs.getString("PET_NAME"));
            p.setOwnerName(rs.getString("OWNER_NAME"));
            p.setPrice(rs.getInt("PRICE"));    p.setBirthDate(rs.getDate("BIRTH_DATE"));
            return p;
        }
    }
    , ownerName);

```

```

System.out.println(petList.get(0).getPetName());
System.out.println(petList.get(0).getOwnerName());
// BeanPropertyRowMapper를 사용해서 도메인으로의 변경을 자동화
pet = jdbcTemplate.queryForObject( " SELECT * FROM PET WHERE PET_ID=?"
    , new BeanPropertyRowMapper<Pet>(Pet.class)    , id);
System.out.println(pet.getPetName());
System.out.println(pet.getOwnerName());
// ResultSetExtractor를 사용한 도메인의 변환
// 부모 도메인이 하나일 때
Owner owner = jdbcTemplate.query( " SELECT * FROM OWNER O, PET P where
O.OWNER_NAME=P.OWNER_NAME and O.OWNER_NAME=?"
    , new ResultSetExtractor<Owner>() {
        public Owner extractData(ResultSet rs) throws SQLException,
DataAccessException {
            if (!rs.next()) {    return null;                }
            Owner owner = new Owner();
            owner.setOwnerName(rs.getString("OWNER_NAME"));
            do { Pet pet = new Pet();    pet.setPetId(rs.getInt("PET_ID"));
                pet.setPetName(rs.getString("PET_NAME"));
                pet.setOwnerName(rs.getString("OWNER_NAME"));
                pet.setPrice(rs.getInt("PRICE"));
                pet.setBirthDate(rs.getDate("BIRTH_DATE"));
                owner.getPetList().add(pet);
            } while(rs.next());
            return owner;
        }
    }
    , ownerName);

```



```
System.out.println(owner.getOwnerName());
System.out.println(owner.getPetList().get(0).getPetName());
System.out.println(owner.getPetList().get(0).getOwnerName());
// 부모 도메인이 여럿일 때
List<Owner> ownerList = jdbcTemplate.query(
    " SELECT * FROM OWNER O, PET P where O.OWNER_NAME=P.OWNER_NAME
ORDER BY p.OWNER_NAME" , new ResultSetExtractor<List<Owner>>() {
    public List<Owner> extractData(ResultSet rs) throws SQLException,
    DataAccessException {
        List<Owner> result = new ArrayList<Owner>();
        Owner owner = null;    String currentPk = "";
        while (rs.next()) {
            String ownerName = rs.getString("OWNER_NAME");
            if (!ownerName.equals(currentPk)) {
                owner = new Owner();
                owner.setOwnerName(rs.getString("OWNER_NAME"));
                currentPk = ownerName;    result.add(owner);
            }
            Pet pet = new Pet();    pet.setPetId(rs.getInt("PET_ID"));
            pet.setPetName(rs.getString("PET_NAME"));
            pet.setOwnerName(rs.getString("OWNER_NAME"));
            pet.setPrice(rs.getInt("PRICE"));
            pet.setBirthDate(rs.getDate("BIRTH_DATE"));
            owner.getPetList().add(pet);
        }
        return result;
    }
});
```



```

System.out.println(ownerList.get(0).getOwnerName());
System.out.println(ownerList.get(0).getPetList().get(0).getPetName());
System.out.println(ownerList.get(0).getPetList().get(0).getOwnerName());
// INSERT/UPDATE/DELETE문
pet = new Pet();                pet.setPetId(03);
pet.setPetName("나비");          pet.setOwnerName("kk");
pet.setPrice(10000);             pet.setBirthDate(new Date());
jdbcTemplate.update( "INSERT INTO PET (PET_ID, PET_NAME, OWNER_NAME, PRICE,
BIRTH_DATE) VALUES (?, ?, ?, ?, ?)"
    , pet.getPetId(), pet.getPetName(), pet.getOwnerName(), pet.getPrice(),
pet.getBirthDate());
jdbcTemplate.update( "UPDATE PET SET PET_NAME=?, OWNER_NAME=?, PRICE=?,
BIRTH_DATE=? WHERE PET_ID=?"
    , pet.getPetName(), pet.getOwnerName(), pet.getPrice(), pet.getBirthDate(),
pet.getPetId());
jdbcTemplate.update("DELETE FROM PET WHERE PET_ID=?", pet.getPetId());
// NamedParameterJdbcTemplate를 사용      // 메소드 체인으로 기술할 때
npJdbcTemplate.update(
    " INSERT INTO PET (PET_ID, PET_NAME, OWNER_NAME, PRICE, BIRTH_DATE)" +
    " VALUES (:PET_ID, :PET_NAME, :OWNER_NAME, :PRICE, :BIRTH_DATE)"
    , new MapSqlParameterSource()
    .addValue("PET_ID", pet.getPetId())
    .addValue("PET_NAME", pet.getPetName())
    .addValue("OWNER_NAME", pet.getOwnerName())
    .addValue("PRICE", pet.getPrice())
    .addValue("BIRTH_DATE", pet.getBirthDate())
);

```

```
jdbcTemplate.update("DELETE FROM PET WHERE PET_ID=?", pet.getPetId());
// 메소드 체인으로 기술하지 않을 때
MapSqlParameterSource map2 = new MapSqlParameterSource();
map2.addValue("PET_ID", pet.getPetId());
map2.addValue("PET_NAME", pet.getPetName());
map2.addValue("OWNER_NAME", pet.getOwnerName());
map2.addValue("PRICE", pet.getPrice());
map2.addValue("BIRTH_DATE", pet.getBirthDate());
npJdbcTemplate.update(
    " INSERT INTO PET (PET_ID, PET_NAME, OWNER_NAME, PRICE, BIRTH_DATE)" +
    " VALUES (:PET_ID, :PET_NAME, :OWNER_NAME, :PRICE, :BIRTH_DATE)"
    ,map2
);
jdbcTemplate.update("DELETE FROM PET WHERE PET_ID=?", pet.getPetId());
// BeanPropertySqlParameterSource를 사용 도메인으로부터 파라미터로의 변환을 자동화
BeanPropertySqlParameterSource beanProps =
    new BeanPropertySqlParameterSource(pet);
npJdbcTemplate.update(
    " INSERT INTO PET (PET_ID, PET_NAME, OWNER_NAME, PRICE, BIRTH_DATE)" +
    " VALUES (:petId, :petName, :ownerName, :price, :birthDate)" ,beanProps
);
jdbcTemplate.update("DELETE FROM PET WHERE PET_ID=?", pet.getPetId());
}
}
```

# Spring 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 만들 수 있다
- unchecked-Exception이 발생하는 경우는 대부분 복구가 불가능한 것이므로 직접처리 할 필요는 없다.
- 만약 복구가 가능한 경우라면 예외를 잡아 호출 스택으로 전달되도록 할 수 있다.

## 예외처리의 규칙작성


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

# pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>sample</groupId>
  <artifactId>transaction</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <properties>
    <org.springframework.version>4.1.1.RELEASE</org.springframework.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>${org.springframework.version}</version>
    </dependency>
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>1.6.8</version>
    </dependency>
```



```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${org.springframework.version}</version>
</dependency>
<dependency>
  <groupId>commons-dbcp</groupId>
  <artifactId>commons-dbcp</artifactId>
  <version>1.2.2</version>
</dependency>      <dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.16</version>
</dependency>
</dependencies>
</project>
```



# 트랜잭션

1. 트랜잭션은 단일 작업으로 동작되어야 하는 논리적인 작업의 묶음
2. 물리적으로 여러 개 쿼리문이 마치 한 개 쿼리문처럼 동작되도록 하는 것
3. 트랜잭션 성질
  - 1) 원자성(Atomicity) : **분리 할 수 없는 하나의 단위로 작업**은 모두 완료되거나 모두 취소되어야 합니다.
  - 2) 일관성(Consistency) : 사용되는 **모든 데이터는 일관되어야** 합니다.
  - 3) 격리성(Isolation) : 접근하고 있는 데이터는 **다른 트랜잭션으로 부터 격리** 되어야 합니다. 트랜잭션이 진행되기 전과 완료된 후에 상태를 볼 수 있지만 트랜잭션이 진행되는 중간 데이터는 볼 수 없습니다.
  - 4) 영속성(Durability) : 트랜잭션이 정상 종료되면 그 결과는 **시스템에 영구적으로 적용**되어야 합니다.
  - 5) 순차성(Sequentiality) : 데이터를 다시 로드하고 트랜잭션을 재생하여 원래 트랜잭션이 수행된 후의 상태로 데이터를 되돌리는 것을 말합니다.

## Member Table 생성

```
create table member1 (  
    id varchar2(20) primary key,  
    email varchar2(20),  
    password varchar2(10),  
    name varchar2(20),  
    regdate date  
);
```



## appCtx.xml

```
<?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:aop="http://www.springframework.org/schema/aop"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.1.xsd">
```

# 회원가입 입력프로젝트

```
<context:component-scan base-package="ch03"/>
<context:property-placeholder location="jdbc.properties"/>
<bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
    <constructor-arg ref="dataSource" />
</bean>
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
destroy-method="close">
    <property name="driverClass" value="${driverClassName}" />
    <property name="jdbcUrl" value="${url}" />
    <property name="user" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
    <property name="maxPoolSize" value="${maxPoolSize}" />
</bean>
```

## JdbcTemplate조회 메소드

`List<T> query(String sql, RowMapper<T> rowMapper)`

`List<T> query(String sql, Object[] args, RowMapper<T> rowMapper)`

`List<T> query(String sql, RowMapper<T> rowMapper, Object... args)`

RowMapper를 이용해서 ResultSet의 결과를 자바객체로 변환

## Member.java

```
public class Member {  
    private String id;                private String email;  
    private String password;         private String name;  
    private Date regdate;  
  
    public String getId() {           return id;}  
    public void setId(String id) {this.id = id;    }  
    public String getEmail() {        return email;    }  
    public void setEmail(String email) { this.email = email; }  
    public String getPassword() {      return password; }  
    public void setPassword(String password) {  
        this.password = password;  
    }  
    public String getName() { return name;    }  
    public void setName(String name) { this.name = name;}  
    public Date getRegdate() { return regdate; }  
    public void setRegdate(Date regdate) {  
        this.regdate = regdate;  
    }  
    public String toString() {  
        return "회원[아이디:"+id+", 이메일:"+email+", 암호:"+password+  
            ", 이름:"+name+", 가입일:"+regdate+"]";  
    }  
}
```

## MemberDao.java

```
package member;
import java.util.List;
public interface MemberDao {
    Member select(String id);
    int insert(Member member);
    List<Member> list();
    int update(Member member);
    int delete(String id);
}
```

# 회원가입 입력프로젝트

```
package member;
@Repository
public class MemberDaoImpl implements MemberDao {
    @Autowired
    JdbcTemplate jt;
    public Member select(String id) {
        Member member = new Member();
        try{ member = jt.queryForObject(
            "select * from member3 where id=?",
            new BeanPropertyRowMapper<Member>(Member.class),id);
        }catch(Exception e) {      return null; }
        return member;
    }
    public int insert(Member member) {
        int result = jt.update(
            "insert into member3 values(?,?,?,sysdate)",
            member.getId(), member.getEmail(),
            member.getPassword(), member.getName());
        return result;
    }
}
```

# 회원가입 입력프로젝트

```
public List<Member> list() {
    List<Member> list = jt.query(
        "select * from member3 order by id",
        new BeanPropertyRowMapper<Member>(Member.class));
    return list;
}
public int update(Member member) {
    int result = jt.update( "update member3 set email=?, "
        + "password=?,name=? where id =?",
        member.getEmail(), member.getPassword(),
        member.getName(), member.getId());
    return result;
}
public int delete(String id) {
    int result = jt.update(
        "delete from member3 where id=?", id);
    return result;
}
}
```



```
package member;  
import java.util.List;  
public interface MemberService {  
    int insert(Member member);  
    Member select(String string);  
    List<Member> list();  
    int update(Member member);  
    int delete(String string);  
}
```

# 회원가입 입력프로젝트

```
package member;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
@Service
public class MemberServiceImpl implements MemberService {
    @Autowired
    MemberDao md;
    public int insert(Member member) {
        int result = 0;
        Member member2 = md.select(member.getId());
        if (member2 == null) {
            result = md.insert(member);
        } else System.out.println("이미 있는 데이터 입니다");
        return result;
    }
    public Member select(String id) {
        return md.select(id);
    }
}
```

# 회원가입 입력프로젝트

```
public List<Member> list() {  
    return md.list();  
}  
public int update(Member member) {  
    int result = 0;  
    Member member2 = md.select(member.getId());  
    if (member2 != null) {  
        result = md.update(member);  
    } else System.out.println("없는데 우짜 수정해유");  
    return result;  
}  
public int delete(String id) {  
    int result = 0;  
    Member member2 = md.select(id);  
    if (member2 != null) {  
        result = md.delete(id);  
    } else System.out.println("데이터가 없어서 삭제할 수 없습니다");  
    return result;  
}  
}
```

# Main.java

```
public class Ex01 {
    private static MemberService ms;
    public static void main(String[] args) {
        AbstractApplicationContext ac=new GenericXmlApplicationContext("spring_db.xml");
        ms = ac.getBean(MemberService.class);
        Scanner sc = new Scanner(System.in);
        while(true) {
            System.out.println("명령어를 입력하세요");
            String command = sc.nextLine();
            if (command.equals("x")) break;
            else if (command.startsWith("insert")) {
                insert(command.split(" "));
            } else if (command.startsWith("select")) {
                select(command.split(" "));
            } else if (command.equals("list")) { list();
            } else if (command.startsWith("update")) {
                update(command.split(" "));
            } else if (command.startsWith("delete")) {
                delete(command.split(" "));
            } else help();
        }
        System.out.println("작업 종료");
        ac.close(); sc.close();
    }
}
```

# Main.java

```
private static void delete(String[] str) {
    if (str.length != 2) {          help(); return;    }
    int result = ms.delete(str[1]);
    if (result > 0 ) System.out.println("삭제 성공 ! 대박");
    else System.out.println("삭제 실패 ! 쪽박");
}
private static void update(String[] str) {
    if (str.length != 5) {          help(); return;    }
    Member member = new Member();
    member.setId(str[1]);              member.setEmail(str[2]);
    member.setPassword(str[3]); member.setName(str[4]);
    int result = ms.update(member);
    if (result > 0) System.out.println("수정 성공 ㅋㅋ");
    else System.out.println("수정 실패 ㅠㅠ");
}
private static void list() {
    List<Member> list = ms.list();
    if (list == null || list.size() == 0)
        System.out.println("데이터가 없습니다");
    else {
        for(Member member : list) {
            System.out.println(member);
        }
    }
}
```

# Main.java

```
private static void insert(String[] str) {
    if (str.length != 5) { help(); return; }
    Member member = new Member();
    member.setId(str[1]);                member.setEmail(str[2]);
    member.setPassword(str[3]); member.setName(str[4]);
    int result = ms.insert(member);
    if (result > 0 ) System.out.println("입력 성공");
}
private static void select(String[] str) {
    if (str.length != 2) { help(); return; }
    Member member = ms.select(str[1]);
    if (member == null) System.out.println("없는 데이터 입니다");
    else System.out.println(member);
}
private static void help() {
    System.out.println("명령어가 잘못 됐습니다");
    System.out.println("insert id email password name");
    System.out.println("update id email password name");
    System.out.println("delete id");
    System.out.println("select id");
    System.out.println("list");
    System.out.println();
}
}
```