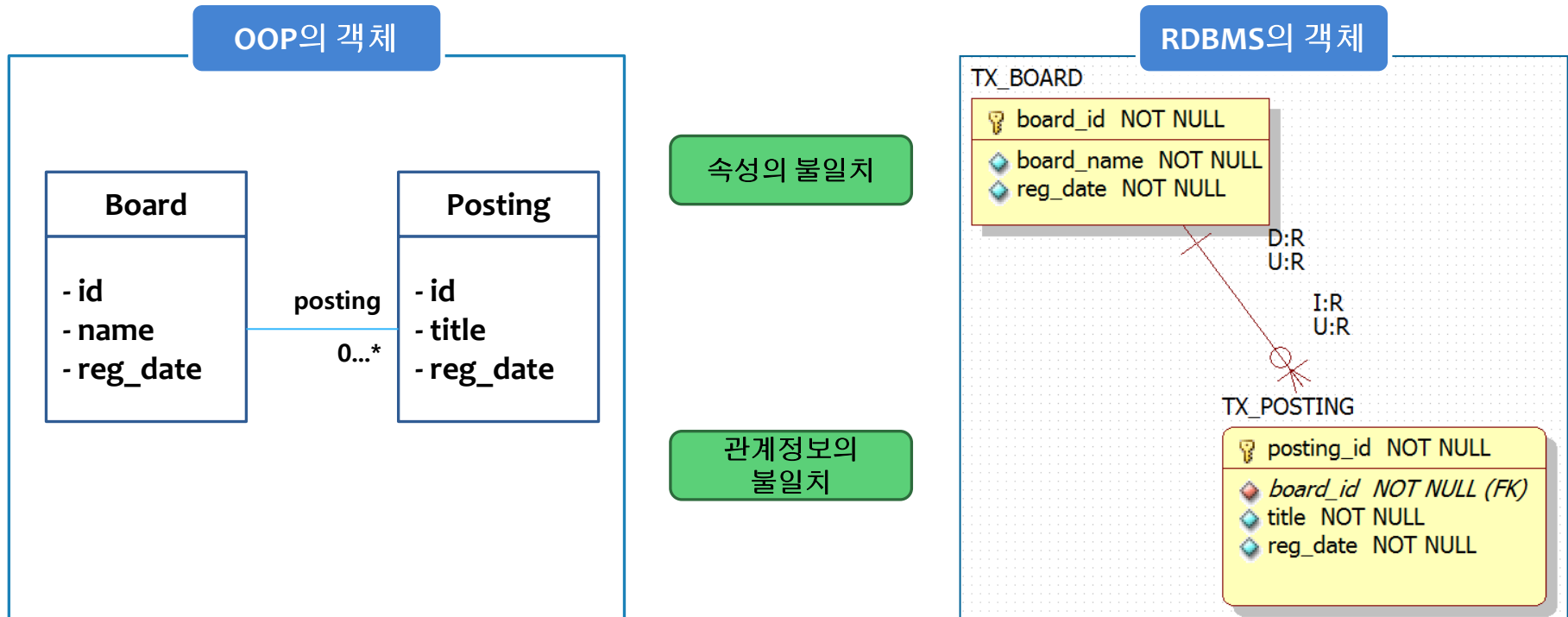


Spring - MyBatis

1-1. MyBatis 소개

- 애플리케이션이 처리한 결과는 필요할 때 다시 사용하기 위해 데이터베이스에 저장
- 자바 애플리케이션에서 RDBMS와 연동하기 위해서는 JDBC API 사용
- 애플리케이션 개발자는 낮은 수준의 데이터 접근 작업보다 복잡한 비즈니스 로직 개발
- OOP에서의 객체와 RDBMS에서의 객체를 불일치를 해결하기 위해 MyBatis와 Hibernate와 같은 데이터 접근 프레임워크 사용



1-1. MyBatis 소개

- iBatis(MyBatis 이전)는 2001년 클린턴 비긴이 시작한 암호 및 보안 프로젝트의 한 부분
- 2006년 부터 자바, .NET, Ruby 언어를 지원하는 프레임워크로 발전
- 2010년 자바의 Annotation 기능 지원을 포함, 기능 확장 → MyBatis로 배포
- 아파치 프로젝트로 시작 → 구글 → GitHub 프로젝트로 발전



MyBatis



Clinton Begin

Rioter at Riot Games

Calgary, Canada Area | Computer Software

500+
connections

Current

Riot Games

Previous

[Yet to be Announced Startup], Outpace Systems, Inc., Riot Games

Recommendations

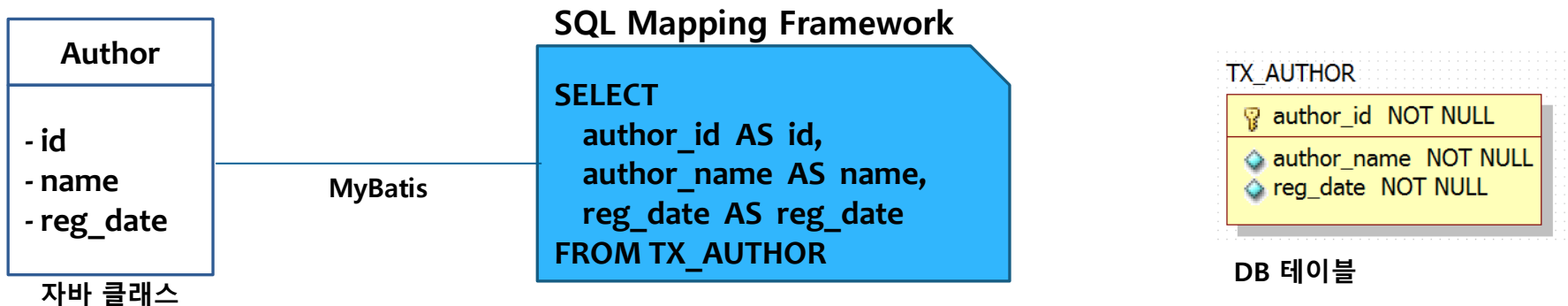
21 people have recommended Clinton

Websites

Blog
Company Website

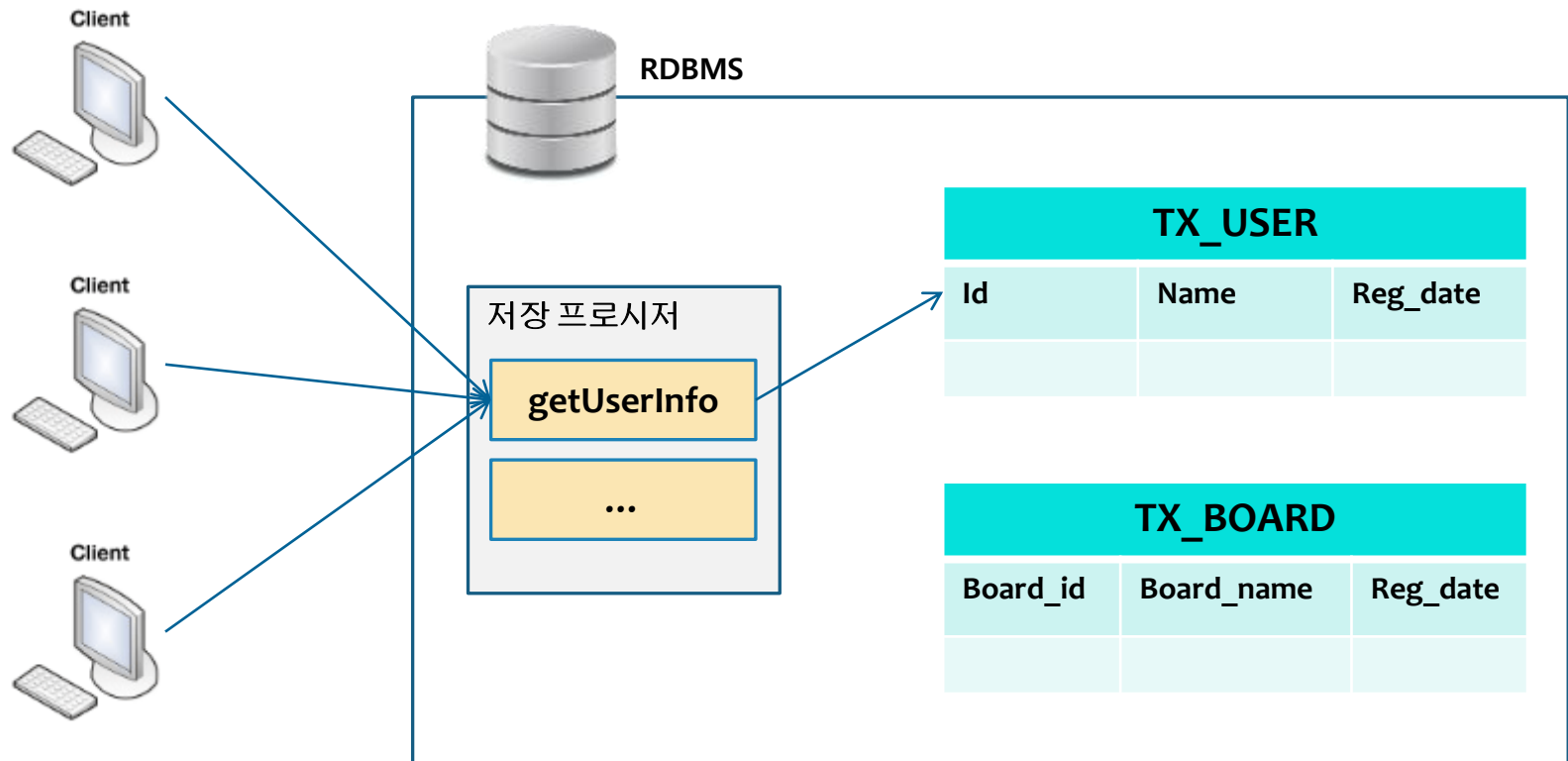
1-1. MyBatis 소개

- MyBatis는 객체와 SQL 매핑을 지원하는 SQL Mapping 프레임워크
- SQL을 캡슐화한 후, 외부로 분리하여 설계 → SQL은 애플리케이션 외부에 위치
- JDBC 코드와 매핑에 반복적으로 사용되는 코드는 XML 및 Annotation을 사용하여 제거 가능
- 구성
 - 설정파일(XML)
 - 매퍼
 - 결과 매핑, 매핑 구문
 - 파라미터 타입
 - 결과 타입



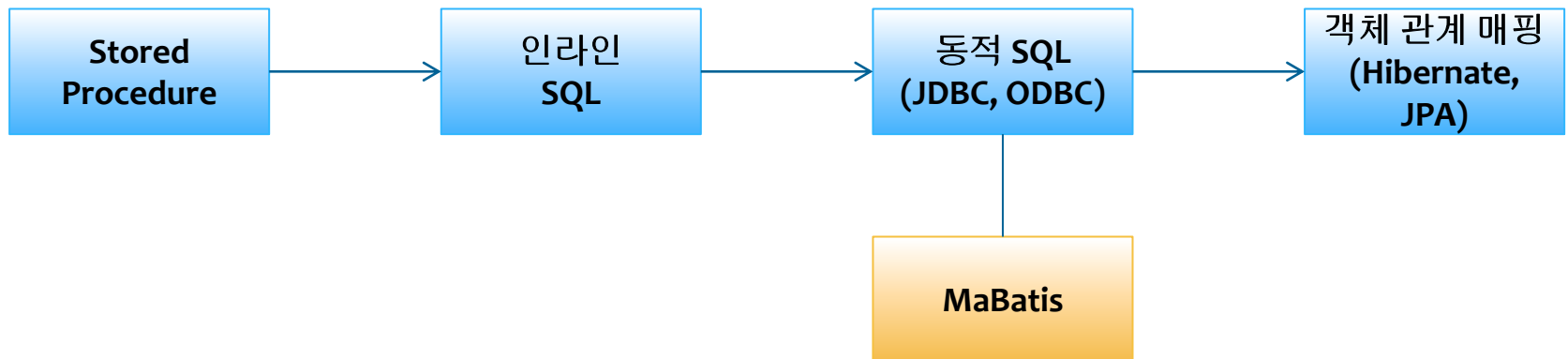
1-2. 데이터 접근기술 흐름

- 모든 RDBMS는 데이터베이스에 접근하기 위해 SQL 사용
- 저장 프로시저(Stored Procedure)는 클라이언트에서 데이터베이스에 내장되어 있는 프로시저(함수)를 직접 사용하는 방법
 - SP를 사용하면 비즈니스 로직이 프로시저에 포함되어 확장에 어려움
 - 데이터베이스와 애플리케이션 간의 역할과 책임이 불명확해 짐



1-2. 데이터 접근기술 흐름

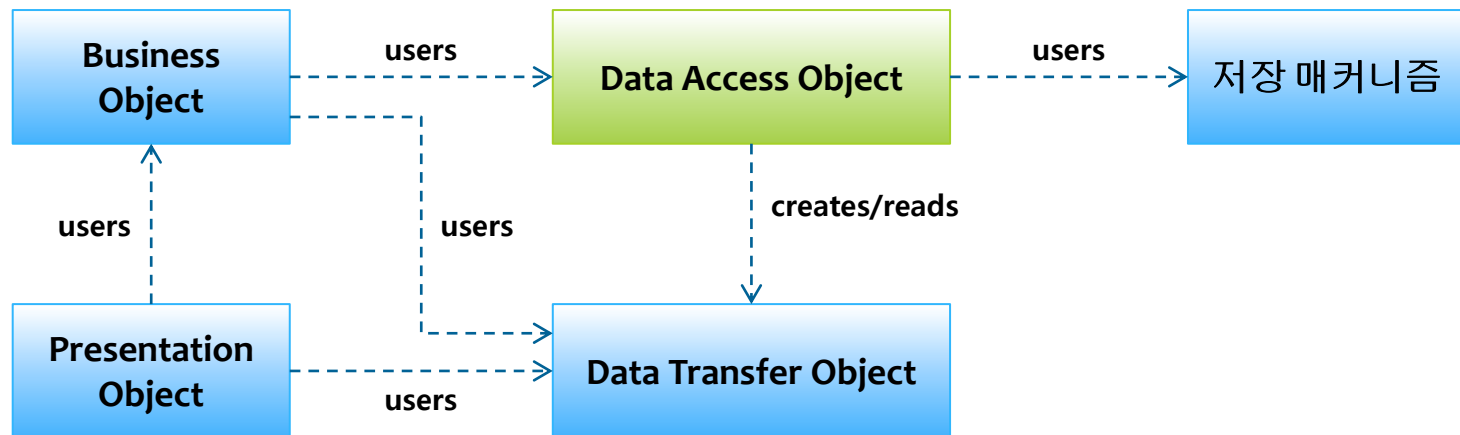
- 인라인 SQL
 - SP의 한계를 극복하여 일반 프로그래밍 언어에 SQL을 내장하는 방법
- 동적 SQL
 - 인라인 SQL의 전처리기를 사용하지 않고 SQL를 다루는 방법
 - 현재 프로그래밍 언어에서 가장 많이 사용되는 RDBMS 접근 방법
- 객체 관계 매핑 (ORM)
 - 객체 지향 애플리케이션에서 데이터베이스에 접근하는 방법



1-3. 데이터 접근 패턴

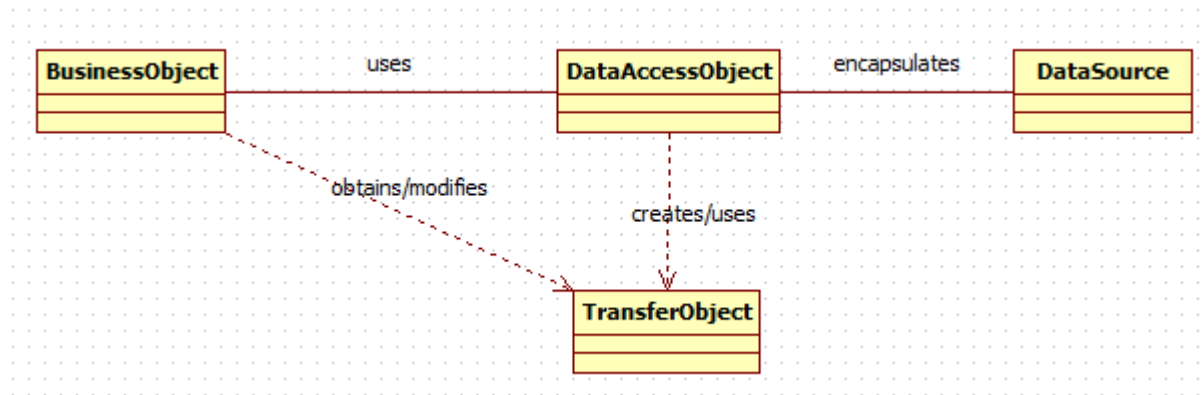
- Data Access Object(DAO) 패턴

- 비즈니스 로직과 저수준의 데이터 액세스 로직을 분리
- DB에 접근하는 코드와 해당 데이터를 가지고 작업하는 코드를 분리
- DAO 객체는 DB 접근에 대한 책임을 가지며, 어떠한 비즈니스 메소드도 포함하지 않음
- 비즈니스 로직은 별도로 위치하고 독립적으로 테스트가 가능하며 저장 레이어가 변경되어도 재사용이 가능



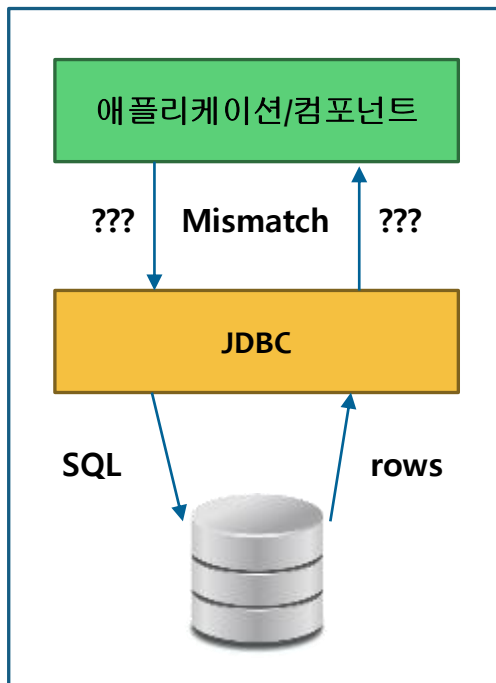
1-3. 데이터 접근 패턴

- Business Object는 데이터를 요구하는 클라이언트로 데이터 원본에 접근하여 데이터를 얻거나 저장
- DAO는 데이터소스에 대한 투명한 접근이 가능하도록 Business Object에 대한 데이터 접근 구현을 추상화
- 데이터 소스는 RDBMS, OODBMS, XML Repository, File System 등 데이터 원본에 대한 구현을 의미
- DTO는 DAO에서 Business Object에 데이터를 전달하거나 수정하기 위해 사용

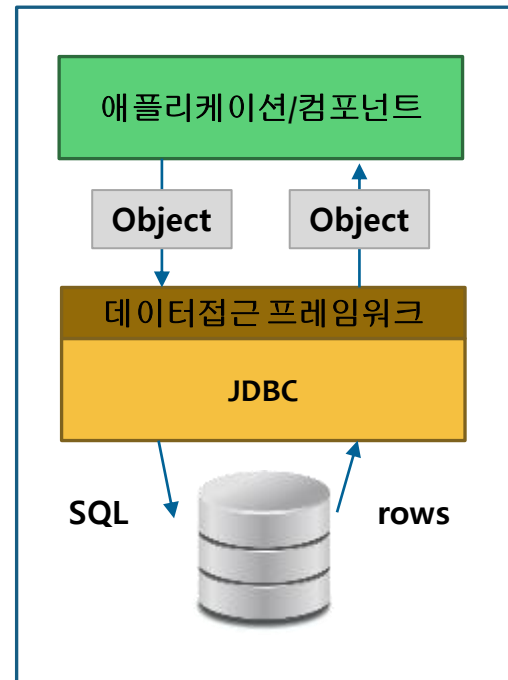


1-4. JDBC의 문제점

- 자바 애플리케이션은 JDBC를 이용하여 데이터베이스에 접근
- JDBC(Java Database Connectivity)는 SQL을 이용하여 데이터베이스에 데이터를 입/출력
- 순수한 JDBC 애플리케이션은 소스코드에 SQL문과 결과를 객체로 매핑하는 로직을 포함
- 이러한 설계 방식은 SQL 개발과 유지보수를 어렵게 함



직접 JDBC 사용 시



데이터접근 프레임워크 사용 시

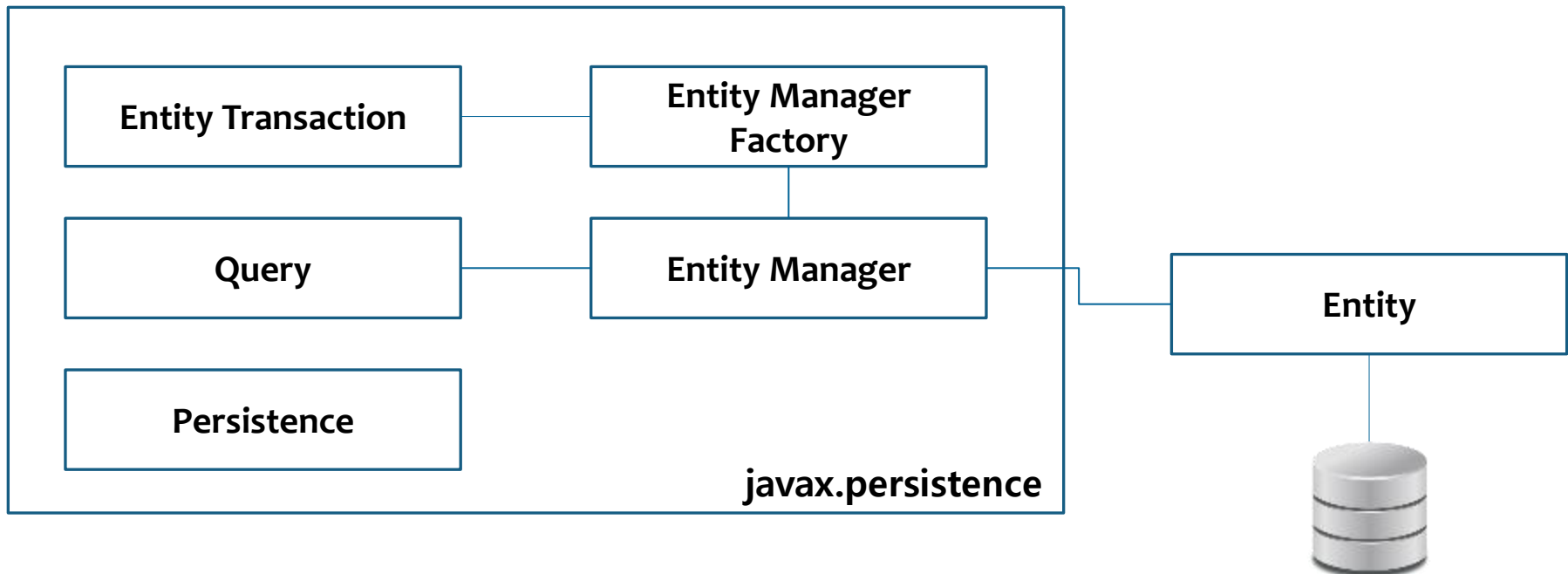
1-5. SQL 매핑과 JPA

- OOP 언어는 상속과 다형성과 같은 객체지향 개념을 이용하여 도메인 모델을 표현
- RDBMS에서 테이블 형태로 데이터를 표현하기 때문에 객체/관계형 패러다임 불일치가 발생
- SQL Mapping 프레임워크는 SQL 작성/매핑에 대한 비용이 발생 → 패러다임 불일치 해결 안됨
- 패러다임 불일치를 해소하기 위해 객체/관계형 매핑 모델(ORM) 사용

객체지향 모델	관계형 모델
객체, 클래스	행(ROW), 테이블
속성	컬럼
Identity	Primary Key
참조에 의한 관계	Foreign Key
상속/다형성	지원하지 않음

1-5. SQL 매핑과 JPA

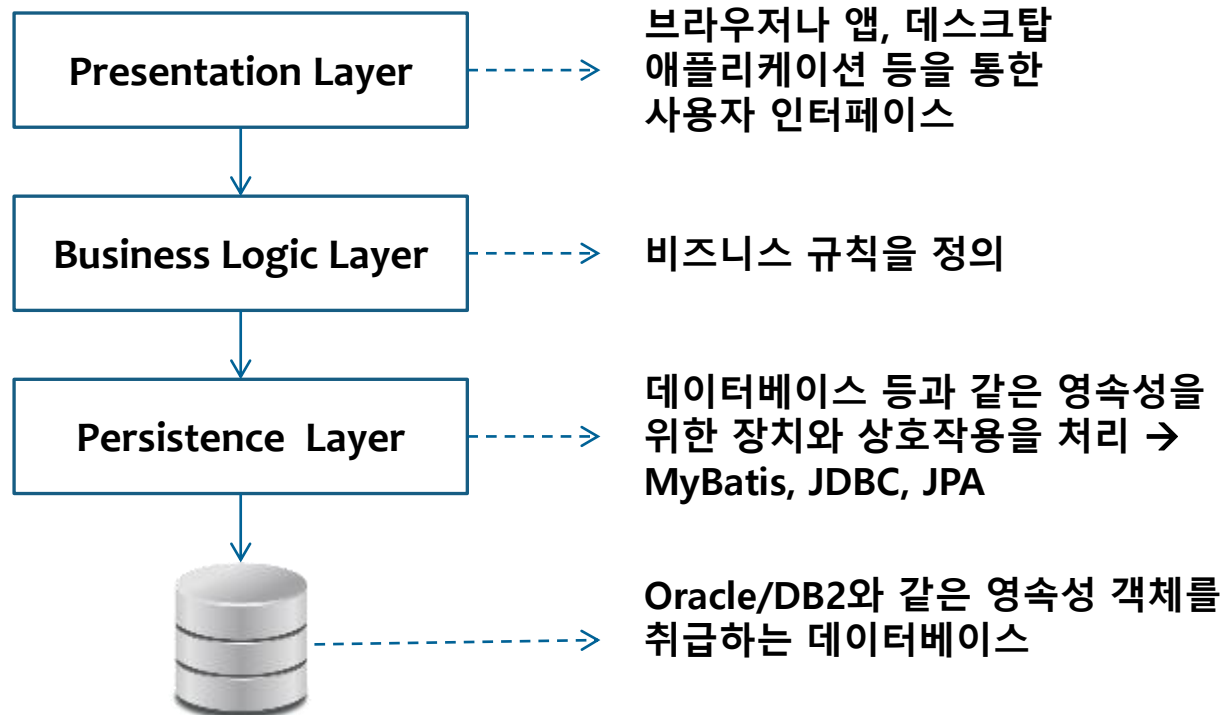
- ORM 프레임워크는 자바 저장 표준인 JPA(Java Persistence API)를 구현
→ Hibernate, TopLink 등
- ORM 프레임워크가 SQL을 자동으로 생성
→ Query 결과를 해당 객체로 매핑하는 코드 불필요
- 이런 방식의 프레임워크는 개발자가 비즈니스 로직에 집중하게 해주기 때문에 개발 생산성이 높음
- 객체모델링을 바탕으로 하지만, 관계형 모델링에 대한 깊은 이해가 필요



2-1. Layered Architecture

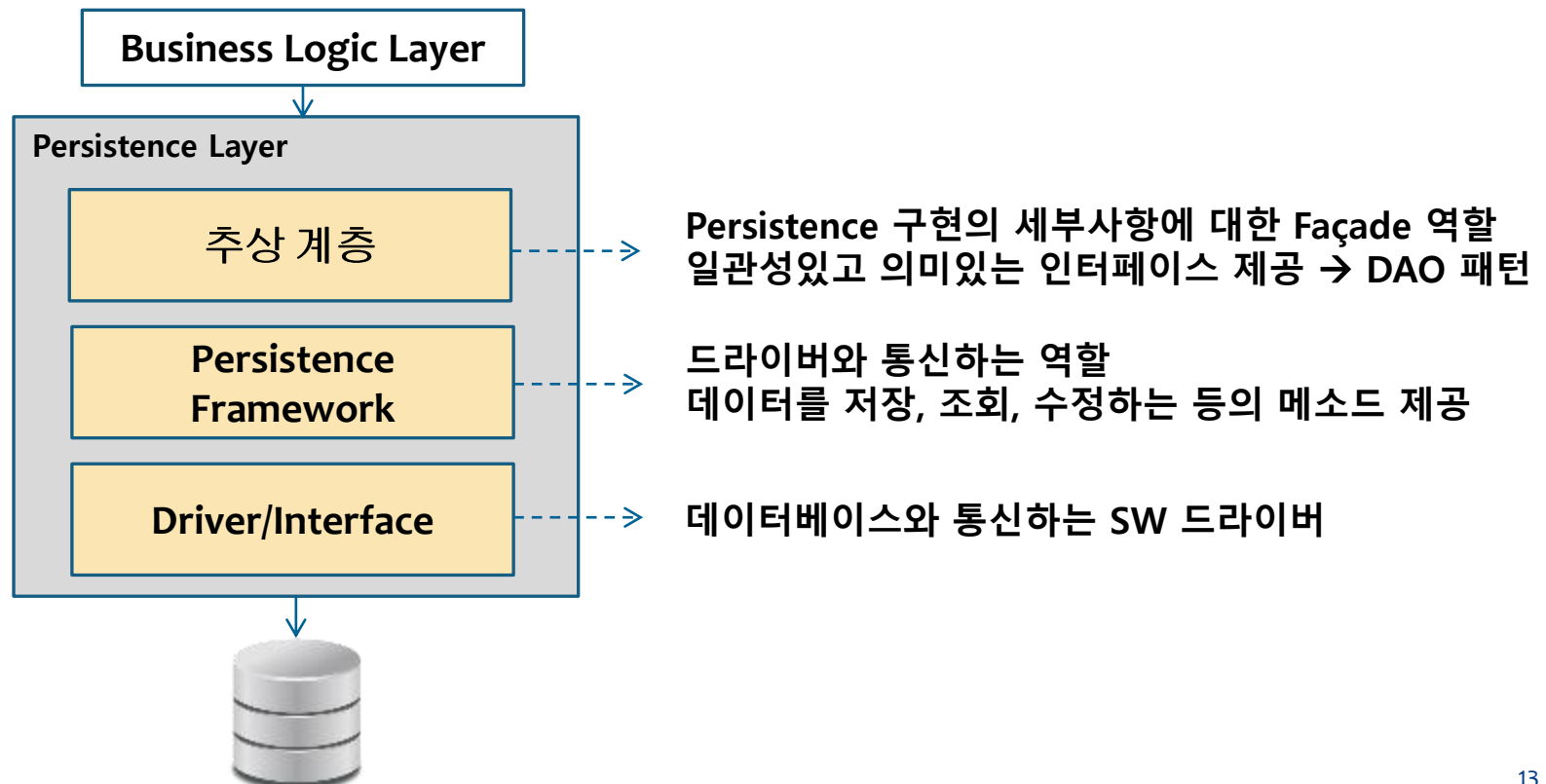
- 애플리케이션에서 관련된 것들을 묶어서 관심사를 분리한 계층형 아키텍처를 설계
- 하나의 계층은 자신의 고유 역할을 수행하고 바로 하위의 계층을 제외하고 다른 계층에 대해서는 관심을 갖지 않음
- 계층 간의 통신은 인터페이스를 통해 이루어지며 상위에서 하위 계층에만 의존
- 계층 간의 변경에 대한 영향도가 적어 변경 개발을 할 경우 생산성을 향상시키고 계층 내에 다른 기술로 대체가 용이

전통적인
엔터프라이즈
애플리케이션의 4계층
아키텍처



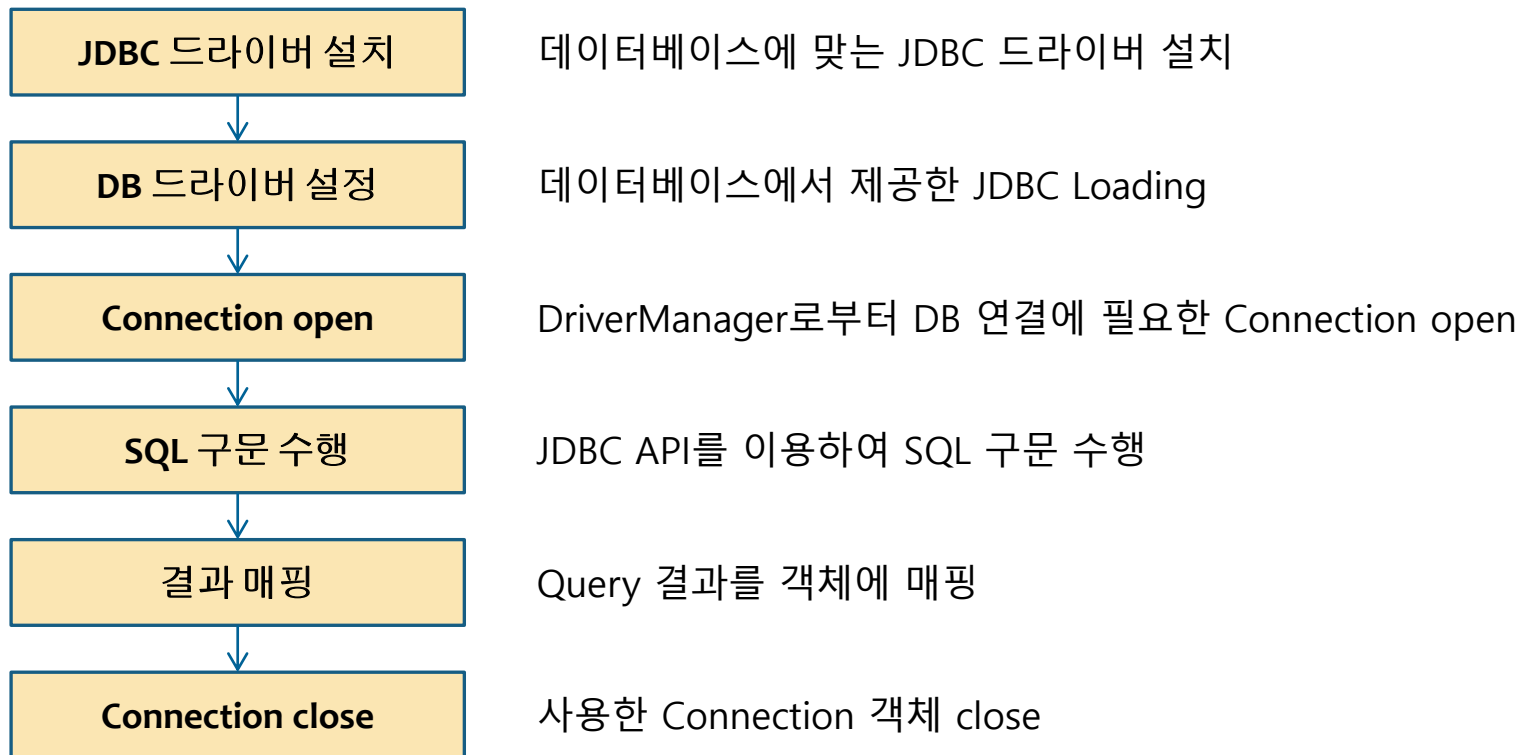
2-1. Layered Architecture

- Persistence Layer 영역을 MyBatis와 같은 프레임워크가 담당
- Persistence Layer의 관심사는 저장소와 객체에 저장된 데이터
- 데이터가 어떻게 저장되고, 어떻게 전송되는지에 대한 세부사항을 다른 계층에 노출하지 않음
- Business Logic Layer에서 처리 해야 할 일을 Persistence Layer에서 처리하지 않도록 구현



2-2. 데이터접근 프로그래밍 - JDBC

- JDBC는 데이터베이스에 접속할 수 있도록 지원되는 자바 API
- 데이터베이스에서 데이터를 조회, 저장, 갱신, 삭제하는 방법 제공
- JDBC 인터페이스는 자바와 데이터베이스를 연결하기 위한 기능을 정의 → 데이터베이스를 제공하는 벤더에서 이를 구현
- SQL은 자바 소스코드 내에 위치



2-2. 데이터접근 프로그래밍 - JDBC

```
public List<User> findAllUser() {  
    Connection conn = null;  
    PreparedStatement pstmt = null;  
    ResultSet rs = null;  
    List<User> users = new ArrayList<User>();  
    try {  
        conn = dataSource.getConnection();  
        StringBuffer sb = new StringBuffer();  
        sb.append("SELECT a.user_id, a.user_name, a.email, a.password ");  
        sb.append("FROM tx_user a");  
        pstmt = conn.prepareStatement(sb.toString());  
        rs = pstmt.executeQuery();  
        while (rs.next()) {  
            User user = new User();  
            user.setUserId(rs.getString(1));  
            user.setUserName(rs.getString(2));  
            user.setUserEmail(rs.getString(3));  
            user.setPassword(rs.getString(4));  
  
            users.add(user);  
        }  
    } catch (SQLException ex) {  
        ex.printStackTrace();  
    } finally {  
        if (rs != null) rs.close();  
        ...  
    }  
  
    return users;  
}
```

Connection 객체 open

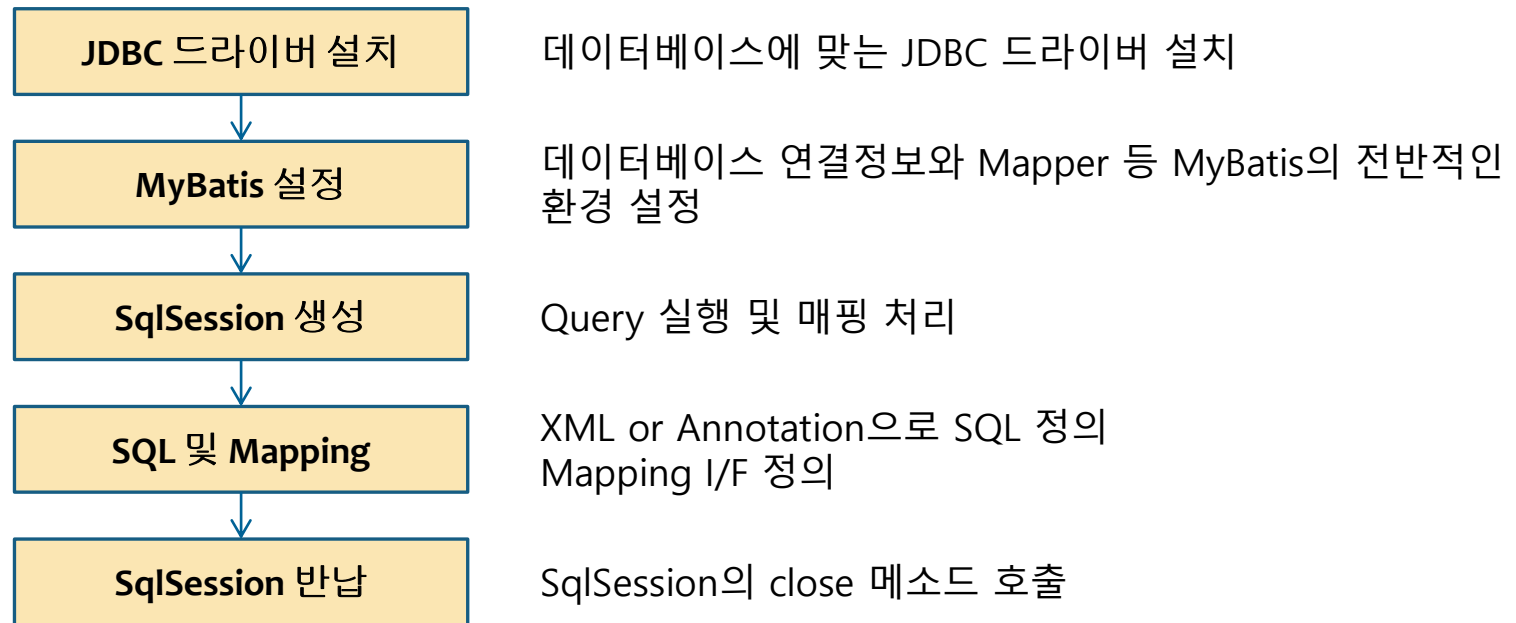
SQL 작성

Query 결과를 객체에 매핑

Connection 자원 close

2-2. 데이터접근 프로그래밍 - MyBatis

- JDBC에서 드라이버 설정 및 Connection open과 같은 절차는 MyBatis 설정 파일에 작성
- JDBC API를 사용하는 부분 → MyBatis가 제공하는 객체(SqlSession)로 처리
- JDBC의 Query와 Mapping은 MyBatis에서의 XML 파일 or Annotation으로 설정
- JDBC에서의 Connection close 작업은 SqlSession 객체를 close



2-2. 데이터접근 프로그래밍 - MyBatis

```
public List<User> findAllUser() {  
    SqlSession session = null;  
    List<User> users = null;  
    try {  
        Reader reader = Resources.getResourceAsReader(  
            "com/lectopia/mybatis/mybatis-config.xml");  
        SqlSessionFactory sqlSessionFactory =  
            new SqlSessionFactoryBuilder().builder(reader);  
        session = sqlSessionFactory.openSession();  
        UserMapper mapper = session.getMapper(UserMapper.class);  
        users = mapper.findAllUsers();  
    } catch (IOException ex) {  
        ex.printStackTrace();  
    } finally {  
        if (session != null)  
            session.close();  
    }  
  
    return users;  
}
```

DB 설정과 트랜잭션 등,
MyBatis 동작 규칙 정의된
XML 파일 Loading

SQL 수행 및 Mapping
처리를 위한 MyBatis 객체
생성

Mapping I/F 취득

SQL 수행

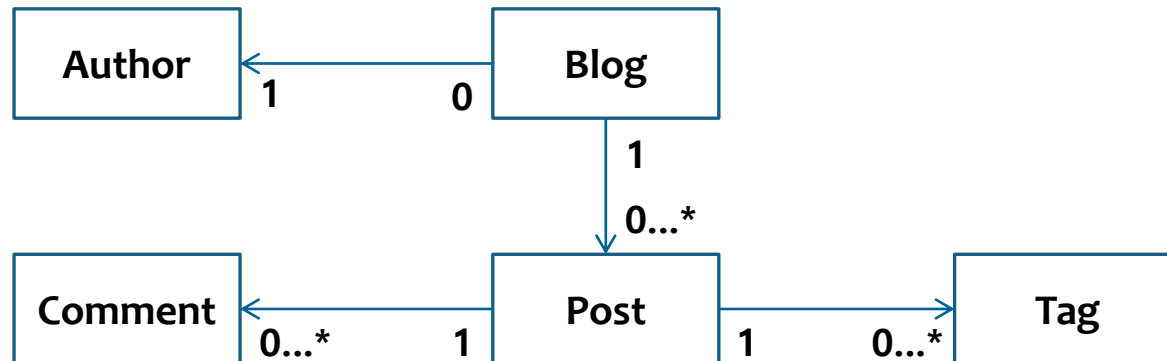
Connection 자원 close

```
<mapper namespace="com.lectopia.mybatis.blog.mapper.UserMapper">  
    <select id="findAllUsers" resultType="User">  
        select user_id, user_name, user_password  
        from tx_user  
    </select>  
</mapper>
```

SQL 구문 작성

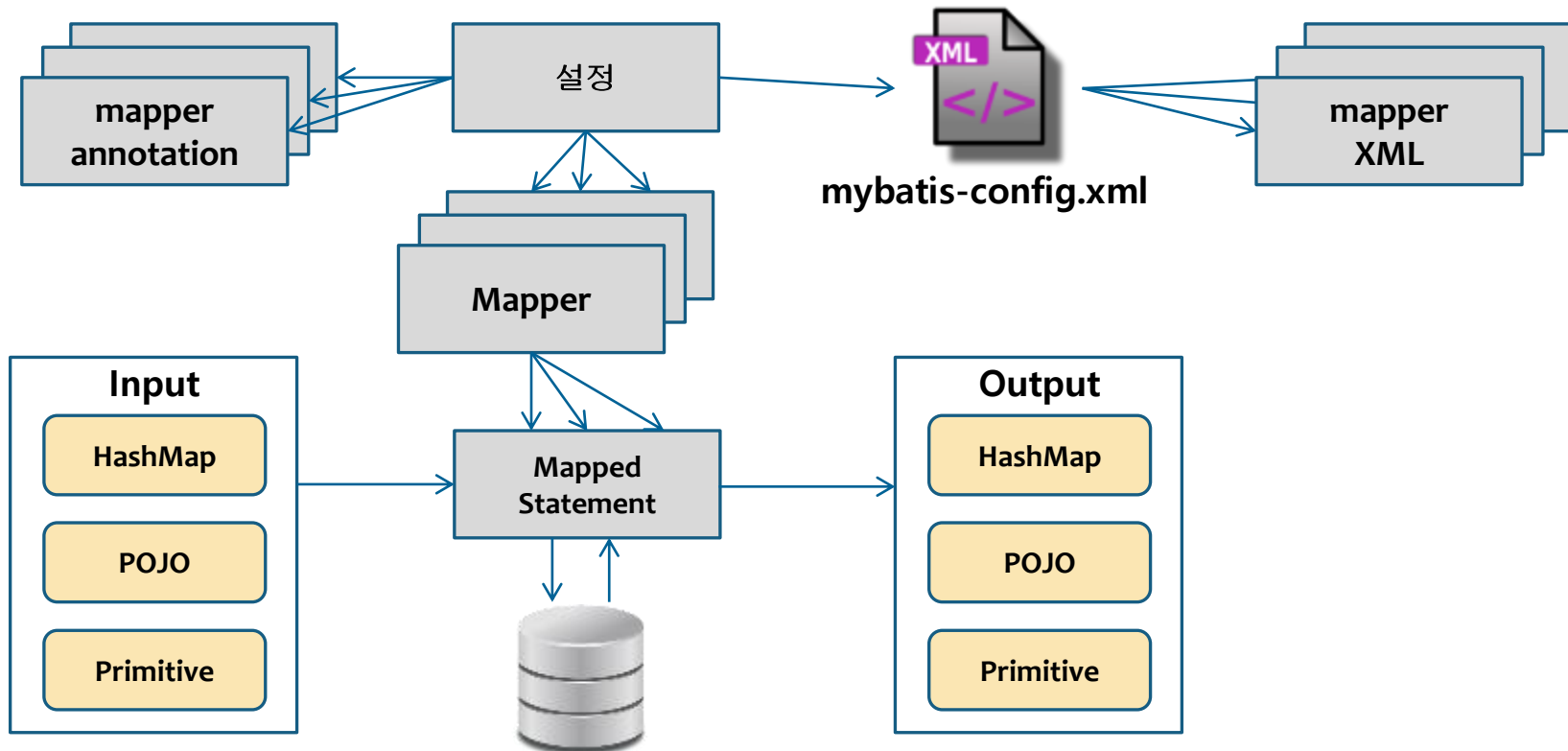
2-3. 실습

- 실습한 시스템은 Blog 시스템
- 하나의 블로그에는 한명의 작성자(소유자)가 있으며, 다수개의 글을 작성 가능
- 하나의 글에는 다수개의 댓글과 태그 작성 가능



3-1. Configuration – MyBatis를 구성하는 5가지 요소

- 설정파일(mybatis-config.xml)은 데이터베이스 설정과 Mapper.xml 정의 등 MyBatis의 동작 규칙을 정의
- Mapper는 SQL을 정의한 XML or Annotation으로 정의한 Interface를 의미
- Mapper Statement는 SQL과 SQL의 수행결과를 객체에 Mapping하는 구문을 의미
- Input, Output: Map 객체, 사용자정의 객체, 원시타입 등을 지원



3-2. SqlSessionFactory 및 SqlSession

- SqlSession 객체는 MyBatis를 사용하기 위한 가장 기본적인 자바 인터페이스
- SqlSession 객체를 통해 SQL 명령어 실행, Mapper를 얻고, 트랜잭션 관리
- SqlSessionFactory 인스턴스를 통해 SqlSession 객체의 인스턴스를 생성
- SqlSessionFactory는 SqlSessionFactoryBuilder를 통해 생성



```
Reader reader = Resources.getResourceAsReader(  
    "com/lectopia/mybatis/mybatis-config.xml");  
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().builder(reader);  
SqlSession session = sqlSessionFactory.openSession();
```

3-2. SqlSessionFactory 및 SqlSession

- **SqlSession** 객체는 SQL문 수행에 필요한 CRUD 메소드 제공
- CRUD 메소드는 SQL 매핑 XML 파일에 정의된 select, insert, update, delete 구문을 수행
- **selectOne**, **selectList** 메소드는 데이터 1건 및 N건 조회를 위해 사용
- **insert**, **update**, **delete**는 데이터 추가, 수정, 삭제를 위해 사용

```
SqlSession session = sqlSessionFactory.openSession();
Author author = null;
try {
    author = (Author)session.selectOne(
        "com.lectopia.mybatis.blog.mapper.AuthorMapper.findAuthor", id);
} finally {
    session.close();
}

return author;
```

```
<mapper namespace="com.lectopia.mybatis.blog.mapper.AuthorMapper">
    <select id="findAuthor" parameterType="String" resultType="Author">
        select id, password, name, email
        from tx_authroe
        where id = #{id, jdbcType=VARCHAR}
    </select>
</mapper>
```

3-3. Mapper Interface

- **SqlSession 객체에서 제공하는 insert, update, delete 등의 메소드는 문자열로 매핑 구문을 호출**
→ 문자열 처리 오류나 타입 변환에 오류가 발생할 수 있음
- **인터페이스를 통한 명시적인 Parameter/Return type 선언으로 안전한 타입 변경 가능**
- **SqlSession 객체에서 자동 생성된 Mapper 프록시 객체를 반환 받아 사용 가능**



```
SqlSession session = sqlSessionFactory.opneSession();
Author author = null;
try {
    AuthorMapper mapper = session.getMapper(AuthorMapper.class);
    author = mapper.findAuthor(id);
} finally {
    session.close();
}

return author;
```

3-3. Mapper Interface

- Annotation을 이용하여 SQL 구문을 수행 가능
- XML 매핑 구문의 네임스페이스와 Mapper Interface의 이름과 패키지가 동일해야 함
- 메소드 이름과 XML 매핑 구문의 ID가 같고 Parameter/Return type이 동일해야 함
- Annotation 매핑 구문과 XML 매핑 구문을 중복해서 사용 시 오류 발생

```
package com.lectopia.mybatis.blog.mapper;

public interface AuthorMapper {
    Author findAuthor(String id);
    @select("SELECT id, password, name FROM TX_AUTHOR")
    List<Author> findAll();
    void regist(Author author);
    void update(Author author);
    void delete(String id);
}
```

```
<mapper namespace="com.lectopia.mybatis.blog.mapper.AuthorMapper">
    <select id="findAuthor" parameterType="String" resultType="Author">
        select id, password, name, email
        from tx_authroe
        where id = #{id, jdbcType=VARCHAR}
    </select>
</mapper>
```

3-4. 직관적인 Sql 매핑 구문

- SQL의 join 결과를 반복적으로 사용하여 내포 된 결과를 매핑 가능 → Nested Result Mapping
- <constructor> 요소를 이용하여 생성자를 이용하여 객체 생성 가능
- <association> 요소는 1:1 관계 매핑을 지원
- <collection> 요소는 1:N 관계 매핑을 지원

```
<resultMap id="detailedBlogResultMap" type="Blog">
  <constructor>
    <idArg column="id" javaType="int" />
  </constructor>

  <result property="title" column="blog_title" />

  <association property="author" column="blog_author_id"
    javaType="Author">
    <id property="id" column="author_id" />
    <result property="username" column="blog_username" />
    <result property="password" column="blog_password" />
    ...
  </association>

  <collection property="posts" ofType="Post">
    <id property="id" column="post_id" />
    <result property="subject" column="post_subject" />
    <association property="author" column="post_author_id"
      javaType="Author">
      ...
    </association>
  </collection>
</resultMap>
```

```
<!-- very complex statment -->
<select id="selectBlogDetails" parameterType="int"
  resultMap="detailedBlogResultMap">
  SELECT
    b.id as blog_id,
    b.title as blog_title,
    b.author_id as blog_author_id,
    a.id as author_id,
    a.username as author_username,
    a.password as author_password,
    p.id as post_id,
    ...
  FROM TX BLOG b
  left outer join Author a on b.author_id = a.id
  left outer join Post p on b.id = p.blog_id
  left outer join Comment c on p.id = c.post_id
  left outer join Post_tag pt on pt.post_id = p.id
  left outer join Tag t a on pt.tag_id = t.id
  WHERE b.id = #{id}
</select>
```


3-4. 직관적인 Sql 매핑 구문

- XML 매핑 파일에서 동적 SQL을 위한 엘리먼트를 지원
- 동적 SQL을 활용하면 조회 조건이 복잡하더라도 1개의SQL ID만으로 작성 가능
- JSTL 문법을 사용하여 직관적으로 작성 가능
- 구문 빌더 API(SelectBuilder, SqlBuilder)를 사용 가능

```
<select id="findAuthroByCondition" parameterType="hashmap"
        resultType="Author">
    SELECT id, password, name, email
    FORM TX_AUTHOR
    <where>
        <if test="id != null">
            AND id=#{id}
        </if>
        <if test="name != null">
            AND name=#{name}
        </if>
    </where>
</select>
```

3-5. 트랜잭션

- `SqlSessionFactory`의 `openSession()` 메소드를 호출하면 트랜잭션이 명시적으로 시작 됨
- 트랜잭션을 종료하기 위해서는 `SqlSession` 객체의 `commit()`, `rollback()` 메소드 호출
- `SqlSession`을 생성할 때 `openSession(true)`를 이용하면 `autoCommit` 모드로 동작
- Spring과 같은 WAS와 연동하여 사용 시 트랜잭션을 WAS에 위임

```
SqlSession session = sqlSessionFactory.openSession();
try {
    session.insert(
        "com.lectopia.mybatis.blog.mapper.PostMapper.registPost", post);
    session.commit();
} finally {
    session.close();
}
```

```
<insert id="registPost" parameterType="Post"
        useGeneratedKeys="true" keyProperty="id">
    INSERT INTO tx_post (subject, content author_id, blog_id)
    VALUES ( #{subjct}, #{content}, #{author_id}, #{blog.id}
    <selectKey keyProperty="id" resultType="int">
        SELECT LAST_INSERT_ID()
    </selectKey>
</insert>
```

3-6. 다중환경설정 지원

- MyBatis는 여러 개의환경 설정을 지원
- 여러 DB환경을 설정 해 두고, 상황에 따라 선택하여 사용
- 여러 환경을 설정하여 사용할 수 있지만, SqlSessionFactory 인스턴스는 한 개만 사용 가능
- 데이터베이스 별로 한의 SqlSessionFactory 인스턴스가 필요

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC">
    <dataSrouce type="POOLED">
      ...
    </environment>
    <environment id="production">
      <transactionManager type="EXTERNAL">
      <dataSrouce type="JNDI">
        ...
      </environment>
    </environments>
```

```
Reader reader = Resources.getResourceAsReader(
    "com/lectopia/mybatis/mybatis-config.xml");
SqlSessionFactory sqlSessionFactory =
    new SqlSessionFactoryBuilder().builder(reader, "production");
```

4-1. SqlSessionFactory

- MyBatis 애플리케이션은 SqlSessionFactory를 중심으로 수행 됨
- SqlSessionFactory 인스턴스는 XML 설정파일을 참조하여 SqlSessionFactoryBuilder가 생성
- XML 설정 파일 외에도 데이터소스(DB 접속 정보) 정보를 참조하는 Configuration 객체를 참조하여 생성할 수 있음
- 재사용 대상이기 때문에 Singleton 패턴, Spring 또는 Goold Guice 등의 DI 프레임워크를 이용하여 관리

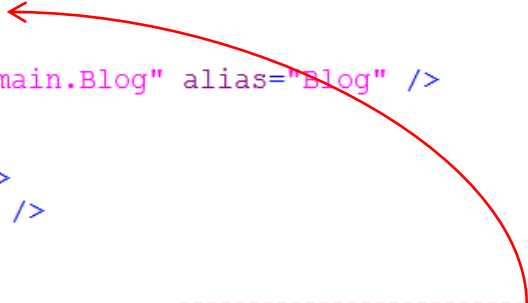
```
Reader reader = Resources.getResourceAsReader(  
    "com/lectopia/mybatis/mybatis-config.xml");  
SqlSessionFactory sqlSessionFactory =  
    new SqlSessionFactoryBuilder().builder(reader, "production");
```

```
DataSource ds = BlogDataSourceFactory.getBlogDataSource();  
TransactionFactory factory = new JdbcTransactionFactory();  
Environment env = new Environment("development", factory, ds);  
Configuration conf = new Configuration(env);  
Configuration.addmapper(AuthorMapper.class);  
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(conf);
```

4-2. MyBatis 설정 파일 – mybatis-config.xml

- MyBatis 설정 파일은 MyBatis 설정의 중앙 허브 역할
- 데이터베이스 정보, Mapper 정보 등을 프레임워크에 제공
- 설정 파일 명(mybatis-config.xml)은 변경 가능
- typeAlias 설정을 추가하면 “패키지명+클래스명”의 긴 문자열을 간단하게 줄여서 사용 가능

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <properties resource="datasource.proeprties" />
  <typeAliases>
    <typeAlias type="com.lectopia.mybatis.blog.domain.Blog" alias="Blog" />
  </typeAliases>
  <settings>
    <setting name="useColumnLabel" value="true" />
    <setting name="useGeneratedKeys" value="true" />
  </settings>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        ...
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper resource="com/lectopia/mybatis/example/BlogMapper.xml"/>
  </mappers>
</configuration>
```



```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/mybatis
username=mybatisuser
password=mybatis
```

4-2. MyBatis 설정 파일 – mybatis-config.xml

- **MyBatis 설정 파일은 다양한 설정과 속성을 가짐**
- **총 11개의 엘리먼트**
- **mappers에 작성한 파일명은 MyBatis 동작 시 필요한 SQL 목록이므로 반드시 포함되어야 함**

속성명	설명
properties	설정정보 (mybatis-config.xml)에서 참조하는 property 파일 지정
settings	MyBatis 실행 옵션을 설정
typeAliases	XML 설정에서 Mapping에 사용되는 클래스의 별칭 설정
typeHandlers	DB의 Type과 Java Type의 변환 시 사용되는 handler 설정
objectFactory	인스턴스 생성 시 사용할 Factory 클래스 설정
plugins	mapped statement 실행 시 intercept 호출을 제공
environments	복수의 환경을 설정하고 상황에 따른 SqlSessionFactory를 생성할 수 있도록 지원
mappers	SQL 구문 및 매핑 정보를 가진 XML 설정 파일 참조 선언

4-2. MyBatis 설정 파일 – properties

- 설정 파일에서 공통 속성을 정의 하거나 외부 파일에서 속성 값을 사용
- 외부 property 파일을 읽으려면 properties 요소의 resource 속성을 사용
- 공통 속성을 외부 파일에 두지 않고 하위 요소인 property 요소를 사용하여 property를 추가 가능
- properties에 설정 된 값을 사용할 때는 \${key} 형식으로 사용

```
<properties resource="config.proeprties">
  <property name="username" value="mybatisuser" />
  <property name="password" value="mybatis" />
</properties>
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC"/>
    <dataSource type="POOLED">
      <property name="driver" value="${driver}"/>
      <property name="url" value="${url}"/>
      <property name="username" value="${username}"/>
      <property name="password" value="${password}"/>
    </dataSource>
  </environment>
</environments>
```

4-2. MyBatis 설정 파일 – properties

- **property**는 3가지 방법으로 설정 가능
- 동일한 속성이 여러 곳에 정의 된 경우 아래의 순서로 **override** 되어 적용 됨

property 요소로 정의

```
<properties resource="config.proeprties">  
  <property name="username" value="mybatisuser" />  
  <property name="password" value="mybatis" />  
</properties>
```

classpath 내 별도의 파일 (config.properties)

```
driver=com.mysql.jdbc.Driver  
url=jdbc:mysql://localhost:3306/mybatis  
username=mybatisuser  
password=mybatis
```

SqlSessionFactoryBuider의 builder 메소드 파라미터로 전달

```
Properties props = new Properties();  
props.setProperty("username", "mybatisuser");  
  
SqlSessionFactory factory =  
    new SqlSessionFactoryBuilder().build(reader, environment, props);
```


4-2. MyBatis 설정 파일 – settings

- settings 요소는 SqlSessionFactory객체가 SqlSession 객체를 생성할 때 객체의 특성을 결정
- 특별한 경우가 아니면 기본 값 사용

속성명	설명	적용값
cacheEnabled	캐시 사용 여부 설정, 매피별로 캐시를 사용하지 않으려면 false로 설정	<u>true</u> /false
lazyLoadingEnabled	늦은 로딩 설정 여부, false 설정 시 모든 관계를 가진 객체를 즉시 로딩 됨	<u>true</u> /false
aggressiveLazyLoading	활성화 상태로 두게 되면 늦은 로딩을 하는 객체는 호출에 따라 점진적으로 로딩, 반면에 개별 프로퍼티는 요청할 때 로드 됨	<u>true</u> /false
multipleResultSetsEnabled	한 개의 구문에서 여러 개의 ResultSet를 허용할 지 여부 (드라이버가 지원해야 함)	<u>true</u> /false
useColumnLabel	컬럼명 대신에 컬럼 라벨(Alias 명)을 사용, 드라이버마다 다르게 작동하므로 확인 필요	<u>true</u> /false
useGeneratedKeys	생성 키 사용 여부 설정, MySQL은 auto_increment, Oracle은 sequence를 제공	true/ <u>false</u>
autoMappingBehavior	자동매핑 설정 (PARTIAL: 중첩 된 결과를 제외하고 간단한 매핑 처리, FULL: 처리가능한 모든 자동매핑 처리)	NONE, <u>PARTIAL</u> , FULL
defaultExecutorType	기본 실행 옵션 설정 (SIMPLE: Statement 객체 재사용하지 않음, REUSE: PreparedStatement 재사용, BATCH: Statement 재사용하고 작업을 일괄 처리)	<u>SIMPLE</u> , REUSE, BATCH
defaultStatementTimeout	DB에서 응답을 기다리기 위한 시간 설정	0이상 정수 (default NULL)
mapUnderScoreToCamelCase	컬럼명으로 사용된 언더바 표기법을 객체의 카멜표기법으로 자동 매핑할지 여부	true/ <u>false</u>

4-2. MyBatis 설정 파일 – typeAliases

- **typeAliases** 요소는 패키지 경로를 포함한 긴 문자열을 약어(별칭)로 선언
- **Sql** 매핑 설정에서 “패키지명+클래스명” 대신 별칭 사용 가능
- **Annotation** 사용하여 설정 가능

```
<configuration>
  <typeAliases>
    <typeAlias type="com.lectopia.mybatis.blog.domain.Blog" alias="Blog" />
  </typeAliases>
  ...
</configuration>
```

```
@Alias("Blog")
public class Blog {
  ...
}
```

4-2. MyBatis 설정 파일 – typeAliases

- 원시 타입이나 일반적인 자바타입들은 타입에 대한 별칭(Alias)이 MyBatis에 내장되어 있음
- 내장되어 있는 별칭은 별도로 설정 하지 않음

별칭	매핑 된 타입
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean

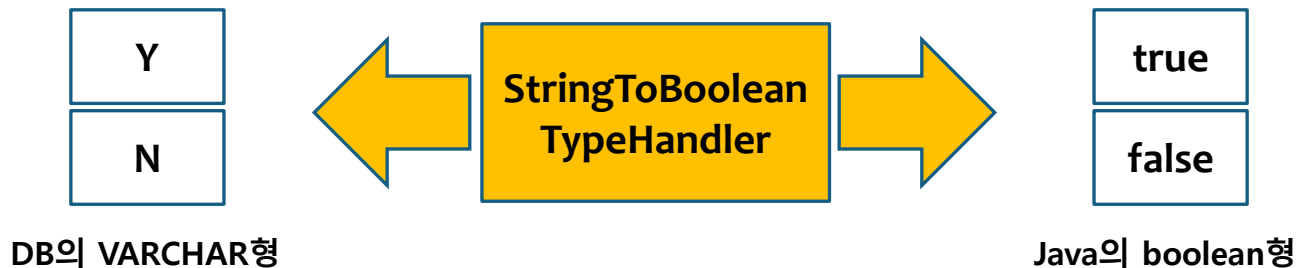
별칭	매핑 된 타입
string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean

별칭	매핑 된 타입
date	Date
decimal	Decimal
bigdecimal	BigDecimal
object	Object
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

4-2. MyBatis 설정 파일 – typeHandler

- 파라미터를 설정하거나 결과 값을 가져올 때 DB의 데이터 타입과 자바 타입을 상호 변환하기 위해 사용
- 자바가 제공하는 대부분의 타입을 지원
- DB의 데이터 타입과 호환되는 타입이 없다면 별도의 타입 핸들러를 정의하여 사용

```
<configuration>
...
<typeHandlers>
  <typeHandler javaType="boolean" jdbcType="VARCHAR"
    handler="com.lectopia.mybatis.blog.handler.StringToBooleanTypeHandler" />
</typeHandlers>
...
</configuration>
```



4-2. MyBatis 설정 파일 – typeHandler

• 기본제공 typeHandler - ①

TypeHandler	Java Type	JDBC Type
BooleanTypeHandler	Boolean, boolean	BOOLEAN
ByteTypeHandler	Byte, byte	NUMBER or BYTE
ShortTypeHandler	Short, short	NUMBER or SHORT INTEGER
IntegerTypeHandler	Integer, int	NUMBER or INTEGER
LongTypeHandler	Long, long	NUMBER or LONG INTEGER
FloatTypeHandler	Float, float	NUMBER or FLOAT
DoubleTypeHandler	Double, double	NUMBER or DOUBLE
BigDecimalTypeHandler	BigDecimal	NUMBER or DECIMAL
StringTypeHandler	String	CHAR, VARCHAR
ClobTypeHandler	String	CLOB, LONGVARCHAR
NStringTypeHandler	String	NVARCHAR, NCHAR

4-2. MyBatis 설정 파일 – typeHandler

• 기본제공 typeHandler - ②

TypeHandler	Java Type	JDBC Type
NClobTypeHandler	String	NCLOB
ByteArrayTypeHandler	byte[]	byte stream type
BlobTypeHandler	byte[]	BLOB, LONGVARBINARY
DateTypeHandler	Date(java.util)	TIMESTAMP
DateOnlyTypeHandler	Date(java.util)	DATE
TimeTypeHandler	Date(java.util)	TIME
SqlTimestampTypeHandler	Timestamp(java.sql)	TIMESTAMP
SqlDateTypeHandler	Date(java.sql)	DATE
SqlTimeTypeHandler	Time(java.sql)	TIME
ObjectTypeHandler	any	OTHER
EnumTypeHandler	Enumeration Type	VARCHAR(code 가 저장됨)

4-2. MyBatis 설정 파일 – typeHandler

- DB의 데이터 타입과 호환되는 타입이 없다면 별도의 타입 핸들러를 정의하여 사용 → 사용자 정의 TypeHandler
- `org.apache.ibatis.type.TypeHandler <T>` 인터페이스를 구현하여 사용

```
public class CustomTypeHandler implements TypeHandler {
    public void setParameter(PreparedStatement pstmt, int i, Object param,
                            JdbcType jdbcType) throws SQLException {
        pstmt.setString(i, (String)param);
    }

    public Object getResult(ResultSet rs, String columnName)
        throws SQLException {
        return rs.getString(columnName);
    }

    public Object getResult(CallableStatement cs, int columnIndex)
        throws SQLException {
        return cs.getString(columnType);
    }
}

<typeHandlers>
  <typeHandler javaType="String" jdbcType="VARCHAR"
    handler="com.lectopia.mybatis.blog.handler.CustomTypeHandler" />
</typeHandlers>
```

4-2. MyBatis 설정 파일 – objectFactory

- 결과 데이터를 설정할 때 별도의 과정을 추가할 목적으로 사용
- 기본 ObjectFactory는 대상 클래스의 기본 생성자를 이용하여 인스턴스화
- 기본 ObjectFactory의 설정을 변경하기 위해서 customObjectFactory로 대체
- ObjectFactory 인터페이스를 구현하거나 DefaultObjectFactory 클래스를 확장하여 구현체를 생성

```
public class CustomObjectFactory extends DefaultObjectFactory {  
    public Object create(Class type) {  
        return super.create(type);  
    }  
  
    public Object create(Class type, List<Class> constructorArgTypes,  
        List<Object> constructorArgs) {  
        return super.create(type, constructorArgTypes, constructorArgs);  
    }  
  
    public void setProperties(Properties properties) {  
        super.setProperties(properties);  
    }  
}
```

```
<objectFactory type="com.lectopia.mybatis.blog.objectfactory.CustomObjectFactory">  
    <property name="someProperty" value="100" />  
</objectFactory>
```


4-2. MyBatis 설정 파일 – plugins

- MyBatis가 매핑 구문을 실행하는 과정에서 특정 시점의 처리를 가로챌 수 있음
- `org.apache.ibatis.plugin.Interceptor` 인터페이스를 구현
- 플러그인이 작동하는 시점과 대상 객체를 지정하려면 `@Intercepts` annotation과 `@Signature` annotation을 사용
- MyBatis는 mapped statement 실행 중에 다음과 같은 intercept 호출을 지원
 - Executor: update, query, flushStatements, commit, rollback, getTransaction, close, isConnected
 - ParameterHandler: getParameterObject, setParameters
 - ResultSetHandler: handleResultSets, handleOutputParameters
 - StatementHandler: prepare, parameterize, batch, update, query

4-2. MyBatis 설정 파일 – plugins

- 플러그인을 이용하는 것은 낮은 수준의 MyBatis 코어 영역을 수정하므로 주의
- MyBatis 코어 행위를 변경하려면 Configuration 클래스 전체를 재정의 해야 함
- 클래스의 내부 메소드를 다시 정의하고
`sqlSessionFactoryBuild.build(myConfig)` 메소드에 객체를 넣어 설정

```
@Intercepts({@Signature(type=Executor.class,
                        method="update", args={MappedStatement.class, Object.class}) })
public class CustomPlugin implements Interceptor {
    public Object intercept(Invocation invocation) throws Throwable {
        return invocation.proceed();
    }

    public Object plugin(Object target) {
        return Plugin.wrap(target, this);
    }

    public void setProperties(Properties props) {
    }
}
```

```
<plugins>
  <plugin interceptor="com.lectopia.mybatis.blog.plugin.CustomPlugin">
    <property name="someProperty" value="100" />
  </plugin>
</plugins>
```

4-2. MyBatis 설정 파일 – environments

- 여러 개의 DB를 사용하거나, 개발과 테스트 등 제품환경에 맞는 여러 개의 환경으로 설정될 수 있음
- DB당 하나의 SqlSessionFactory 객체를 생성
- Environment 파라미터가 없으면 기본 환경으로 설정 됨

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder.build(reader, environment);  
SqlSessionFactory factory = new SqlSessionFactoryBuilder.build(reader, environment, props);
```

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder.build(reader);  
SqlSessionFactory factory = new SqlSessionFactoryBuilder.build(reader, props);
```

```
<environments default="development">  
  <environment id="development">  
    <transactionManager type="JDBC" />  
    <dataSource type="POOLED">  
      <property name="driver" value="${driver}" />  
      <property name="url" value="${url}" />  
      <property name="username" value="${username}" />  
      <property name="password" value="${password}" />  
    </dataSource>  
  </environment>  
</environments>
```

4-2. MyBatis 설정 파일 – environments

- 트랜잭션 관리자 클래스를 설정하는 **transactionManager**
 - JDBC
 - MANAGED
- JDBC 타입
 - MyBatis API에서 제공하는 commit과 rollback 메소드 등을 사용해서 트랜잭션 관리
 - 데이터소스를 통해 획득 가능한 Connection에 의존적
- MANAGED 타입
 - 컨테이너에서 트랜잭션의 라이프 사이클을 관리

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC" />
    <dataSource type="POOLED">
      <property name="driver" value="${driver}" />
      <property name="url" value="${url}" />
      <property name="username" value="${username}" />
      <property name="password" value="${password}" />
    </dataSource>
  </environment>
</environments>
```

4-2. MyBatis 설정 파일 – environments

- “dataSource” 요소에 DB 접근에 필요한 Connection 객체의 정보 서릿 ◦
 - type과 type에 따른 property 설정 가능
 - UNPOOLED, POOLED, JNDI
- UNPOOLED 타입
 - 요청마다 커넥션을 열고 닫는 방식
 - 고성능을 필요로 하지 않는 간단한 애플리케이션에서 사용

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC" />
    <dataSource type="UNPOOLED">
      <property name="driver" value="${driver}" />
      <property name="url" value="${url}" />
      <property name="username" value="${username}" />
      <property name="password" value="${password}" />
    </dataSource>
  </environment>
</environments>
```

4-2. MyBatis 설정 파일 – environments

- **POOLED 타입**

- 커넥션 풀을 이용해서 커넥션의 인스턴스를 관리
- `poolMinimumActionConnections`: 활성 Connection의 개수
- `poolMaximumIdleConnections`: idle 상태의 최대 Connection의 개수
- `poolMaximumCheckoutTime`: Connection 객체의 최대 Check out 시간 (default: 20000ms)
- `poolTimeToWait`: pooling시 최대 대기 시간 (default: 20000ms)
- `poolPingQuery`: database와의 연결이 유효한지 알아보기 위해 실행하는 질의문 (default: "NO PING QUERY SET")
- `poolPingEnabled`: ping query 사용 여부 설정 (default: false)
- `poolPingConnectionsNotUsedFor`: 얼마나 자주 ping query를 사용할 지 설정 (default: 0)

- **JNDI 타입**

- JNDI Context를 통해 컨테이너에서 제공하는 데이터소스를 사용
- `initial_context`: initialContext로 부터 context를 lookup하기 위해 사용 (ex. `java:com/env`)
- `data_source`: 데이터소스를 찾을 수 있는 Context path (ex. `jdbc/mysqlDB`)

4-2. MyBatis 설정 파일 – mappers

- “mappers” 요소에는 mapped statement 설정 파일의 위치를 선언
 - resource
 - url
 - class
 - name
- package 요소를 사용하면 해당 패키지 내의 매퍼를 자동으로 검색

```
<mappers>
  <mapper resource="com/lectopia/mybatis/blog/mapper/AuthorMapper.xml" />
  <mapper resource="com/lectopia/mybatis/blog/mapper/BlogMapper.xml" />
  <mapper url="file:///var/sqlmaps/AuthorMapper.xml" />
  <mapper url="file:///var/sqlmaps/BlogMapper.xml" />
  <mapper class="com.lectopia.mybatis.blog.mapper.AuthorMapper" />

  <package name="com.lectopia.mybatis.blog" />
</mappers>
```

4-3. Sql 매퍼 설정

- Sql 매퍼를 정의하기 위해서 XML이나 인터페이스를 선택할 수 있음
- SqlSession 인터페이스에서 id로 접근 혹은 매퍼 인터페이스의 메소드 이름으로 접근
- 매퍼 인터페이스의 메소드에 어노테이션으로 설정 정보를 입력
- Sql 설정 요소

요소명	설명
sql	다른 구문에서 재사용하기 위한 SQL 문
cache	namespace에 캐싱 정보 설정
cache-ref	다른 namespace에 설정 된 캐싱 정보에 대한 참조
resultMap	DB의 결과 정보 (result set)에서 객체로 정보를 변환하기 위한 방법 정의
insert	매핑 된 INSERT 구문
update	매핑 된 UPDATE 구문
delete	매핑 된 DELETE 구문
select	매핑 된 SELECT 구문

4-3. Sql 매퍼 설정

- MyBatis에서 namespace는 필수이며, 매핑 정보에 접근하기 위해 매퍼 인터페이스의 “패키지명+클래스명”을 사용
- 여러 개의 매퍼에서 SQL ID가 중복되어도 namespace가 다르다면 사용 가능
- 매퍼 인터페이스를 호출하면 “패키지명+클래스명”, 메소드명으로 일치하는 namespace와 SQL ID를 찾아서 호출

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.lectopia.mybatis.blog.mapper.BlogMapper">
  <select id="selectBlog" parameterType="string" resultType="Blog">
    select * from Blog where id = #{id}
  </select>

  <select id="selectBlogAll" parameterType="string" resultType="Blog">
    select * from Blog
  </select>
</mapper>
```

4-3. Sql 매퍼 설정

- **select** 요소에는 DB 조회를 위한 구문을 설정
- 파라미터 표기법은 **`#{key}`**로 표기
- SQL에 **`#{속성명}`**으로 작성하면 파라미터로 전달되는 객체의 이름이 일치하는 속성의 값을 참조
- **resultType**은 SQL의 수행결과 데이터를 매핑할 객체를 지정
→ 패키지명 + 클래스명 또는 별칭

```
<select id="selectBlog" parameterType="string" resultType="Blog">
    select * from Blog where id = #{id}
</select>
```

속성	설명
id	namespace에서 유일한 식별자
parameterType	인자로 전달되는 클래스 이름 또는 별칭
resultType	결과로 반환되는 객체의 클래스 이름 또는 별칭
resultMap	Result Mapping 참조
flushCache	true로 설정 시 statement가 수행될 때마다 캐시가 flush 됨 (default: false)
useCache	true로 설정 시 statement의 결과가 캐싱됨 (default: true for select statements)
timeout	DB로 부터 응답이 돌아올 때까지 응답 대기시간
fetchSize	Driver에 배치 수행시 얼마나 많은 results를 사용할 지 정보를 제공 (default: unset)
statementType	수행 타입 설정 STATEMENT, PREPARED, CALLABLE (default: PREPARED)
resultSetType	FORWARD_ONLY, SCROLL_SENSITIVE, SCROLL_INSENSITIVE 중 설정 가능 (default: unset)

4-3. Sql 매퍼 설정

- 조회 이외의 삽입, 수정, 삭제를 수행하기 위해 insert, update, delete 요소를 사용

```
<insert id="insert" parameterType="Author">
    INSERT INTO tx_author (id, password, name, email)
    VALUES ({id}, {password}, {name}, {email})
</insert>

<update id="update" parameterType="Author">
    UPDATE tx_author SET PASSWORD={password}, NAME={name}
    WHERE ID={id}
</update>

<delete id="delete" parameterType="string">
    DELETE FROM tx_author
    WHERE ID={id}
</delete>
```

- insert, update, delete 요소의 공통 속성

속성	설명
id	namespace에서 유일한 식별자
parameterType	인자로 전달되는 클래스 이름 또는 별칭
flushCache	true로 설정 시 statement가 수행될 때마다 캐시가 flush 됨 (default: false)
timeout	DB로 부터 응답이 돌아올 때까지 응답 대기시간
statementType	수행 타입 설정 STATEMENT, PREPARED, CALLABLE (default: PREPARED)

4-4. insert 문 키 생성

- insert 요소는 자동 생성 키와 관련된 속성을 추가로 설정 가능
- MySql(auto_increment), Oracle(sequence), SQL Server(serial) 처럼 데이터베이스의 자동 증가 필드 기능
- 데이터베이스에서 자동 생성 키를 제공하는 경우 사용 가능
- 자동 생성 키를 지원할 경우
 - useGeneratedKeys="true"
 - keyProperty 설정

```
<insert id="registPost" parameterType="Post"
        useGeneratedKeys="true" keyProperty="id">
    INSERT INTO tx_post (subject, content, author_id, blog_id)
    VALUES (#{subject}, #{content}, #{author.id}, #{blog.id})
</insert>
```

• insert 요소 추가 속성

속성	설명
useGeneratedKeys	(insert only) JDBC의 getGeneratedKeys 메소드를 이용한 자동키 생성 사용여부 설정 (default: false)
keyProperty	(insert only) MyBatis가 getGeneratedKeys에 의해 얻은 값을 어떤 프로퍼티에 설정하는지 결정 (default: unset)
keyColumn	생성키를 가진 테이블의 컬럼명을 설정. 테이블의 첫번째 컬럼이 키 컬럼이 아닌 데이터베이스 (PostgreSQL)에서만 필요

4-4. insert 문 키 생성

- selectKey 요소는 데이터베이스에서 자동 생성 키를 제공하지 않는 경우, 키 값을 별도로 조회하기 위해 사용

```
<insert id="registPost" parameterType="Post">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    SELECT LAST_INSERT_ID()
  </selectKey>
  INSERT INTO tx_post (subject, content, author_id, blog_id)
  VALUES ({subject}, {content}, {author.id}, {blog.id})
</insert>

<insert id="registPost" parameterType="Post">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    SELECT CAST(RANDOM()*1000000 as INTEGER) a from SYSIBM.SYSDUMMY1
  </selectKey>
  INSERT INTO tx_post (subject, content, author_id, blog_id)
  VALUES ({subject}, {content}, {author.id}, {blog.id})
</insert>
```

- selectKey 요소 속성

속성	설명
keyProperty	selectKey 구문의 반환 값이 설정 될 대상 객체의 프로퍼티 이름
resultType	반환되는 객체의 타입
order	selectKey 구문을 언제 수행할 지 여부 (BEFFOE or AFTER)
statementType	수행 타입 설정 STATEMENT, PREPARED, CALLABLE (default: PREPARED)

4-5. SQL 코드 재사용

- 각각의 매핑 구문에서 SQL문장을 재사용 하기 위해서 sql 요소를 사용
- sql 요소로 재사용 가능한 문장을 정의 → 재사용할 각 문장 include 요소 사용

```
<sql id="authorColumns">
    ID, PASSWORD, NAME
</sql>
```

```
<select id="findAuthor" parameterType="int" resultType="Author">
    SELECT <include refid="authorColumns" />
    FROM tx_author
    WHERE id=#{id}
</select>
```

```
<sql id="select-order">
    select * from order
</sql>
<sql id="select-count">
    select count(*) as value from order
</sql>
<sql id="where-shipped-after-value">
    <where>
        <if test="shipDate != null">
            shipDate = #{value}
        </if>
    </where>
</sql>
<select id="getOrderShippedAfter" resultType="map">
    <include refid="select-order" />
    <include refid="where-shipped-after-value" />
</select>
<select id="getOrderCountShipped" resultType="int">
    <include refid="select-count" />
    <include refid="where-shipped-after-value" />
</select>
```

모든 주문 column

주문 횟수

특정일에 값이 있을 경우 주문

특정일 이후의 모든 주문 column

특정일 이후의 주문 횟수

4-6. 파라미터

- Integer와 String과 같은 원시타입이나 간단한 데이터타입은 프로퍼티를 가지지 않음
- MyBatis는 파라미터로 전달되는 객체와 테이블의 데이터의 타입을 자동으로 변환
- javaType 속성과 jdbcType속성을 이용해서 객체와 테이블의 데이터 타입 변환을 직접 지정
- typeHandler 속성을 이용하여 데이터의 타입을 변환하기 위한 핸들러 클래스 지정

```
<select id="findList" parameterType="int" resultType="hashmap">
    SELECT id, password, name
    FROM tx_author
    WHERE id=#{id}
</select>
```

```
<insert id="insertuser" parameterType="User">
    INSERT INTO tx_user (id, username, password)
    VALUES (#{id}, #{username}, #{password})
</insert>
```

```
#{regDate, javaType=Date, jdbcType=TIMESTAMP}
```

```
#{gender, javaType=boolean, jdbcType=VARCHAR, typeHandler=GenderTypeHandler}
```

4-6. 파라미터

- '#' 지시자로 파라미터를 설정하여 SQL에 전달되는 데이터를 매핑
- PreparedStatement의 ? 표기법으로 변경 되어 PreparedStatement Property로 만들어서 값을 설정
- 문자열을 매핑하더라도 ''를 사용하지 않음

```
<select id="selectUser" parameterType="string" resultType="User">
    SELECT id, password, name, email, type
    FROM tx_author
    WHERE id=#{value}
</select>
```



```
SELECT id, password, name, email, type
FROM tx_author
WHERE id='kenneth';
```

"kenneth"

```
User user = (User)session.selectOne("selectUser", userId);
```


4-6. 파라미터

- '\$' 지시자로 파라미터를 설정하면 문자열을 변경하거나 이스케이프 처리하지 않고 매핑
- PreparedStatement의 ? 표기법으로 변경 되는 과정 생략
- like 검색 시 '%' 문자열을 조합하기 위해서 사용
→ **SQL Injection 위험 주의!**

```
<select id="selectUsers" parameterType="string" resultType="User">
  SELECT id, password, name, email, type
  FROM tx_author
  WHERE name LIKE '%${value}%'
</select>
```



```
SELECT id, password, name, email, type
FROM tx_author
WHERE name LIKE '%0|%' ;
```

“0|”

```
List<User> users = (List<User>)session.selectList("selectUsers", name);
```

4-6. 파라미터

- '#' 지시자는 PreparedStatement의 ?로 변경되고 파라미터 값이 매핑
- '\$' 지시자는 파라미터 값이 SQL문에 문자열로 대체

```
<select id="selectAllotBaseballMatch" resultMap="allotMap">
  SELECT
    decode(ASCORE_${col$}, -1, 0, ASCORE_${col$}) as CASE
  FROM
    ${tableName}
  WHERE
    gm_id = #{gameId}
    AND gm_rnd = ${gameRound}
    AND hscore = #{row}
</select>
```

```
[DEBUG] JakartaCommonsLogginImpl.debug(27) | {pstm-100694} Executing Statement:
  SELECT
    decode(ASCORE_3, -1, 0, ASCORE_3) as CASE
  FROM
    gm_odds_bs_nmat
  WHERE
    gm_id = ? AND gm_rnd = ? AND hscore = ?

...
```

4-7. Result 매핑

- resultMap 요소는 resultSet의 결과와 객체 사이의 매핑 정보를 정의
- resultMap 요소는 조회된 컬럼 값과 매핑할 객체의 속성 값을 직접 정의
- resultType 요소는 조회된 컬럼과 이름이 일치하는 객체의 속성을 자동으로 매핑
- 컬럼 이름이 객체의 속성 이름과 다른 경우 SQL 구문의 별칭(alias)을 속성 이름으로 지정하면 자동 매핑 가능

```
<resultMap id="authorResultMap" type="Author">
  <id property="id" column="id" />
  <result property="password" column="password" />
  <result property="name" column="name" />
</resultMap>
```

```
<select id="find" parameterType="int" resultMap="authorResultMap">
  SELECT id, password, name
  FROM tx_author
  WHERE id=#{id}
</select>
```

```
<select id="find" parameterType="int" resultType="com.lectopia.mybatis.domain.Author">
  SELECT id, password, name
  FROM tx_author
  WHERE id=#{id}
</select>
```

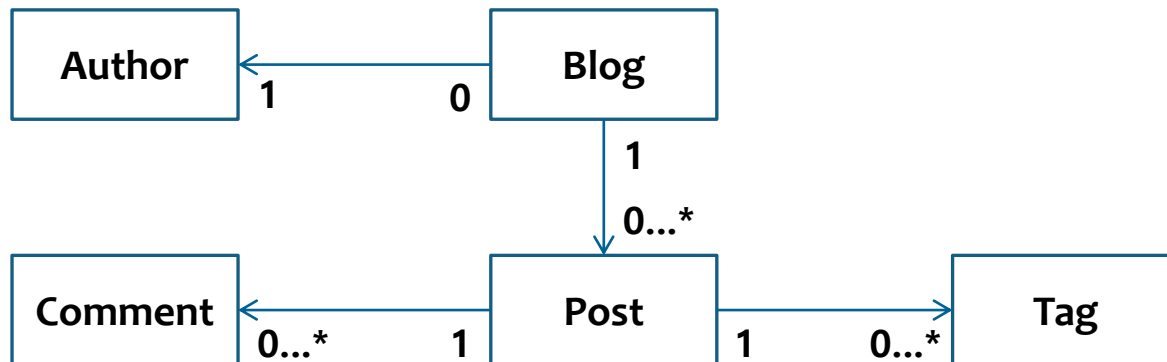
4-8. 실습

- 실습 절차

- 실습 데이터 준비
- SqlSessionFactoryProvider 생성 → SqlSessionFactory 제공
- 도메인 생성 → Author, Blog, Comment, Post, Tag
- 인터페이스 정의 → AuthorDao, BlogDao, PostDao
- Mapper 구현 및 Interface 구현체 구현
- Test Case 작성 및 Test

- 실습 과제

- AuthorMapper: sqlSession 메소드를 통한 select
- AuthorMapper.xml: SQL 재사용 select
- PostMapper: sqlSession 메소드를 통한 insert
- PostMapper.xml: selectKey 사용 insert



5-1. 결과 매핑 - 개요

- 조회한 컬럼 이름과 객체의 속성 이름 또는 setter 메소드가 설정하는 값과 일치하면 조회된 값을 자동 매핑
- 컬럼 이름과 객체 속성 이름이 다른 경우 SQL의 별칭을 사용하거나 resultMap 요소로 직접 매핑 정보를 정의
- mapUnderscoreToCamelCase 설정을 통해 DB의 () 표기법과 자바의 낙타표기법을 자동 매핑 가능
- 1:1 또는 1:N의 복잡한 관계를 매핑하려면 매핑을 추가로 정의

요소명	설명
resultMap	결과를 매핑하기 위한 가장 상위 요소. 매핑을 구분하기 위한 id속성과 매핑 대상 클래스를 정의하는 type 속성이 있음
id	기본 키에 해당하는 컬럼을 설정
result	기본 키가 아닌 컬럼에 대해 매핑
constructor	생성자를 통해 값을 매핑할 때 사용 (생성자 주입)
association	1:1 관계를 매핑하기 위해 사용
collection	1:N 관계를 매핑하기 위해 사용
discriminator	매핑 과정에서 조건을 지정해서 매핑할 때 사용

5-1. 결과 매핑 - resultMap

- resultMap 요소에 id속성을 부여, select 요소에서 resultMap 속성을 사용하여 수행결과를 매핑
- id 요소로 기본 키 컬럼을 지정, 이 컬럼과 매핑하는 객체 속성은 각 객체들을 구분하는 식별자로 사용
- result 요소의 property 속성은 객체 속성명을 정의, column 속성은 DB 컬럼명을 정의
- result 요소의 jdbcType 속성은 컬럼 타입을 정의

```
<resultMap id="authorResultMap" type="Author">
  <id property="id" column="id" />
  <result property="password" column="password" />
  <result property="name" column="name"
    jdbcType="VARCHAR" />
</resultMap>

<select id="find" parameterType="int"
  resultMap="authorResultMap">
  SELECT id, password, name
  FROM tx_author
  WHERE id=#{id}
</select>
```

속성	설명
property	Java 객체에 매핑되는 속성명 (constructor 요소에는 사용 불가)
column	데이터베이스의 컬럼명 또는 별칭
javaType	Java에서 참조되는 클래스명 또는 별칭
jdbcType	JDBC 타입 설정 insert, update, delete 시 nullable column 인 경우 사용
typeHandler	typeHandler 클래스 설정 JDBC 타입을 Java 타입으로 변환 시 사용

5-1. 결과 매핑 - constructor

- 생성자를 사용해서 객체를 생성, 값을 매핑하기 위해 사용
- 생성자에 사용되는 파라미터의 순서에 주의 → javaType 명시
- idArg 요소는 resultMap 속성의 id와 동일한 역할
 - arg 요소는 result와 동일한 역할
- idArg, arg 요소의 순서는 생성자 파라미터 순서와 동일해야 함

```
<resultMap id="authorResultMap" type="Author">
  <constructor>
    <idArg column="author_id" javaType="string" />
    <arg column="author_password" javaType="string" />
  </constructor>
  <result property="name" column="name"
          jdbcType="VARCHAR" />
</resultMap>
```

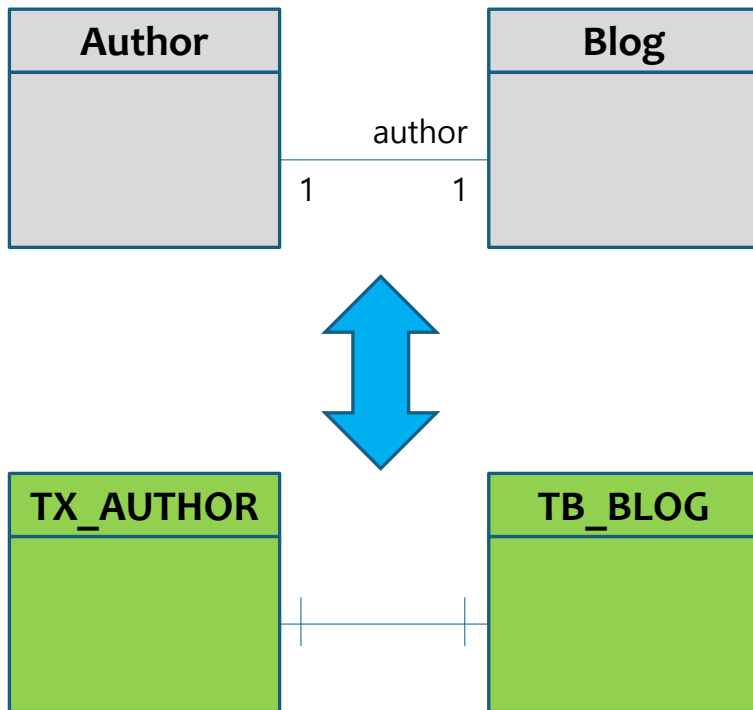
```
public class Author {
    private String id;
    private String password;
    private String name;

    public Author(String id, String password) {
        ...
    }
}
```

5-2. 1:1 관계 매핑 - 개요

- 테이블은 2가지 방법으로 조회 가능
- Nested Select 방법은 각각의 테이블을 별도의 select 문을 사용하여 조회
- Nested Reulst 방법은 JOIN 문을 사용하여 일괄적으로 조회

ex. 1:1 관계를 갖는 도메인 모델과 데이터 모델의 예



```
public class Author {  
    private String id;  
    private String password;  
    private String name;  
    ...  
}
```

```
public class Blog {  
    private int id;  
    private String title;  
    private Author author;  
    private List<Post> posts;  
    ...  
}
```


5-2. 1:1 관계 매핑 - association

- has a 관계를 표현 (1:1 관계)
 - Blog는 한 명의 Author를 가짐
- association 요소의 column 속성에는 참조 키에 해당하는 컬럼, JavaType 속성에는 대상 객체의 Type을 정의
- Nested Select는 연관된 객체의 결과를 다른 SQL을 실행하여 조회
- Nested Results는 연관된 객체의 결과를 JOIN 구문을 사용하여 조회하고 이 결과를 매핑

속성	설명
select	다른 연관된 객체를 조회하는 sql 구문의 id 복합키를 사용할 경우는 column="{prop1=col1, prop2=col2}" 형으로 전달
resultMap	중첩된 검색 결과를 매핑하기 위한 다른 resultMap의 id 여러 테이블의 join으로 이루어진 결과를 이용하는 것을 MyBatis에서는 중첩된 resultMap을 이용하여 처리 가능

5-2. 1:1 관계 매핑 – Nested Select

- 부모 객체를 조회하고 부모 객체의 resultMap에서 자식 객체를 조회하기 위한 Select 구문을 수행
- association 요소의 select 속성을 사용
- N + 1 Selects 문제가 발생
- N + 1 Selects 문제를 해결하기 위해 지연 로딩 (Lazy Loading)을 사용하거나 Nested Results 사용

```
<resultMap id="blogMap" type="Blog">
  <id property="id" column="id" />
  <result property="title" column="title" />
  <association property="author" column="author_id" javaType="Author"
    select="com.lectopia.mybatis.blog.mapper.AuthorMapper.findAuthor" />
</resultMap>
```

```
<select id="findBlogByAuthor" parameterType="int"
      resultMap="blogMap">
  SELECT id, title, author_id,
  FROM tx_blog
  WHERE author_id=#{authorId}
</select>
```

```
<select id="findAuthor" parameterType="string"
      resultMap="Author">
  SELECT id, password, name, email
  FROM tx_author
  WHERE id=#{id}
</select>
```

5-2. 1:1 관계 매핑 – Nested Result

- join 문을 사용하여 결과를 매핑하고, 재사용 여부에 따라 2가지 방법으로 매핑
- 재사용 하는 경우 자식 테이블의 resultMap을 별도로 정의
- association 요소의 resultMap 속성에 자식 테이블의 resultMap Id를 부여 (CASE1)
- 재사용 하지 않는 경우 association 요소 하위에 id, result 요소를 매핑 (CASE2)

```
<select id="selectBlogWithAuthor" parameterType="int"
        resultMap="blogWithAuthorMap">
    SELECT
        b.id, b.title, b.author_id,
        a.id, a.password, a.name
    FROM tx_blog b left outer join tx_author a on b.author_id = a.id
    WHERE b.id=#{aid}
</select>
```

5-2. 1:1 관계 매핑 – Nested Result

- resultMap 재사용

```
<resultMap id="blogWithAuthorMap" type="Blog">                                case #1
  <id property="id" column="id" />
  <result property="title" column="blog_title" />
  <association property="author" column="blog_author_id" javaType="Author"
    select="com.lectopia.mybatis.blog.mapper.AuthorMapper.authorResult" />
</resultMap>

<resultMap id="authorResult" type="Author">
  <id property="id" column="id" />
  <result property="password" column="author_password" />
  <result property="name" column="author_name" />
</resultMap>
```

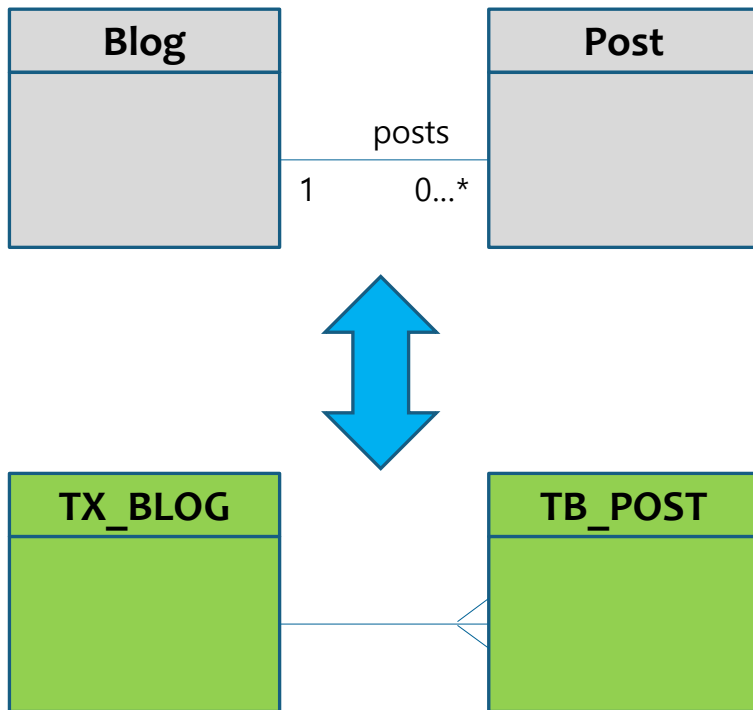
- resultMap 재사용 하지 않는 경우

```
<resultMap id="blogResult" type="Blog">                                case #2
  <id property="id" column="id" />
  <result property="title" column="blog_title" />
  <association property="author" column="blog_author_id" javaType="Author">
    <id property="id" column="author_id" />
    <result property="password" column="author_password" />
    <result property="name" column="author_name" />
  </association>
</resultMap>
```

5-3. 1:N 관계 매핑 – 개요

- 1:N 관계를 가진 객체 관계는 List 형태의 속성을 가짐
- 1:N 관계는 collection 요소를 사용하여 매핑
 - 1:1 관계 매핑을 위한 association 사용법과 유사

ex. 1:N 관계를 갖는 도메인 모델과 데이터 모델의 예



```
public class Blog {  
    private int id;  
    private String title;  
    private Author author;  
    private List<Post> posts;  
    ...  
}
```

```
public class Post {  
    private int id;  
    private Author author;  
    private Blog blog;  
    private String subejct;  
    private String content;  
    ...  
}
```

5-3. 1:N 관계 매핑 – collection

- association과 동일하게 동작
 - Nested Select, Nested Results 2가지 방법도 동일)
- Collection 관계를 표현
 - Blog는 다수의 Post를 가짐
- ofType 속성에는 collection을 구성하는 객체 타입을 정의
- column 속성에는 참조키에 해당하는 컬럼을 정의

속성	설명
ofType	목록을 구성하는 객체의 Type을 정의

```
<collection property="posts" column="blog_id" ofType="Post">  
  <id property="id" column="post_id" />  
  <result property="subject" column="post_subject" />  
  <result property="contents" column="post_contents" />  
</collection>
```

5-3. 1:N 관계 매핑 – Nested Select

- 2개 이상의 select 문을 실행
- 부모 객체를 조회하고 부모 객체의 resultMap에서 자식 객체를 조회하기 위한 select 문을 수행
- N + 1 Selects 문제가 발생

```
<resultMap id="blogMap" type="Blog">
    <collection property="posts" column="id" javaType="ArrayList" ofType="Post"
        select="com.lectopia.mybatis.blog.mapper.PostMapper.findPostsByBlogId" />
</resultMap>
```

```
<select id="findPostsByAuthorId" parameterType="int"
        resultMap="blogMap">
    SELECT id, title, author_id
    FROM tx_blog
    WHERE author_id=#{authorId}
</select>
```

```
<select id="findPostsByBlogId" parameterType="int"
        resultType="Post">
    SELECT *
    FROM tx_post
    WHERE blog_id=#{id}
</select>
```

5-3. 1:N 관계 매핑 – Nested Results

- Join 구문을 사용하여 결과를 매핑, 재사용 여부에 따라 2가지 방법으로 매핑
- 재사용 하는경우 자식 테이블의 resultMap을 별도로 정의
- association 요소의 resultMap 속성에 자식 테이블의 resultMap Id를 부여 (CASE1)
- 재사용 하지 않는 경우 collection 요소 하위로 Id, result 요소를 매핑 (CASE2)

```
<select id="selectBlogWithPost" parameterType="int"
        resultMap="blogWithPostMap">
    SELECT
        b.id, b.title, b.author_id,
        p.id, p.subject, p.contents
    FROM tx_blog b left outer join tx_post p on b.id = p.blog_id
    WHERE b.id=#{id}
</select>
```


5-3. 1:N 관계 매핑 – Nested Results

- resultMap 재사용

```
<resultMap id="blogWithPostMap" type="Blog">
    <id property="id" column="id" />
    <result property="title" column="blog_title" />
    <collection property="posts" ofType="Post" resultMap="psotMap" />
</resultMap>

<resultMap id="psotMap" type="Post">
    <id property="id" column="id" />
    <result property="subject" column="subject" />
    <result property="contents" column="contents" />
</resultMap>
```

case #1

- resultMap 재사용 하지 않는 경우

```
<resultMap id="blogWithPostMap" type="Blog">
    <id property="id" column="blog_id" />
    <result property="title" column="blog_title" />
    <collection property="psots" ofType="Post">
        <id property="id" column="post_id" />
        <result property="subject" column="post_subject" />
        <result property="contents" column="post_contents" />
    </collection>
</resultMap>
```

case #2

5-4. 동적 매핑

- 객체가 상속관계를 가진 경우, SQL 결과를 매핑하는 과정에서 하위 타입을 결정하는 경우가 있음
- discriminator 요소는 java의 switch 구문과 유사
 - 상속관계의 클래스를 매핑할 때 사용
- javaType 속성은 java에서 동일성 비료를 위해 필요
 - column 속성은 비교 대상이 되는 DB 컬럼
- case 요소의 value 속성에 비교할 값을 정의, resultType 속성에는 결과로 생성할 객체의 타입을 정의

```
<resultMap id="vehicleResult" type="Vehicle">
  <id property="id" column="blog_id" />
  <result property="vin" column="vin" />
  <result property="year" column="year" />
  <result property="make" column="make" />
  <result property="model" column="model" />
  <result property="color" column="color" />
  <discriminator javaType="int" column="vehicle_type">
    <case value="1" resultType="carResult">
      <result property="doorCount" column="door_count" />
    </case>
    <case value="2" resultType="truckResult">
      <result property="boxsize" column="box_size" />
      <result property="extendedCab" column="extended_cab" />
    </case>
    <case value="3" resultType="suvResult">
      <result property="allWheelDrive" column="all_wheel_drive" />
    </case>
  </discriminator>
</resultMap>
```

5-5. 동적 SQL – 개요

- 동적 SQL은 조건에 따라 SQL을 동적으로 생성해야 하는 경우에 사용
- iBatis 2.x에 비해 줄어든 요소와 JSTL과 유사한 형태의 문법으로 동적 SQL을 생성
- 매퍼 XML에 동적 SQL을 위한 요소를 작성
- if, choose, when, otherwise, trim, foreach 요소를 사용

요소명	설명
if	조건을 만족할 경우에 SQL을 추가
choose, when, otherwise	여러 가지 조건 중 한 조건을 만족할 경우에 SQL을 추가
where	where 조건이 여러 문장일 경우 'where', 'and', 'or' 문장 추가 가능
set	update 구문에 여러 개의 set 문장을 작성할 경우 각 문장과 구분자(',')를 적절하게 추가
trim	SQL 문장을 생성하면서 각 문장의 앞(prefix) 또는 뒤(suffix)에 추가 또는 삭제
foreach	collection 객체를 반복하면서 SQL 문장을 생성

5-5. 동적 SQL – if

- if 요소의 test 속성에 정의한 조건식이 참(true)이면 요소 하위에 정의한 SQL 문장을 추가
 - ex. parameterType에 해당하는 HashMap 객체에 id 속성이 존재하면 where 구문을 추가하여 수행
- if 조건을 만족하지 못하는 경우 else if나 else를 정의할 수 없음
 - else if 요소가 존재하지 않음
- else if나 else와 같은 동작을 정의하기 위해서는 choose-when-otherwise 요소 사용

```
<select id="findAuthorByCondition" parameterType="hashmap"
        resultMap="Author">
    SELECT id, password, name, email
    FROM tx_author
    <if test="id != null">
        WHERE id=#{id}
    </if>
</select>

<select id="findBlogByCondition" parameterType="Blog"
        resultMap="blogMap">
    SELECT *
    FROM tx_blog
    <if test="title != null">
        WHERE title like '%${title}%'
    </if>
    <if test="author != null and author.id != null">
        WHERE author_id = ${author.id}
    </if>
</select>
```

5-5. 동적 SQL – choose, when, otherwise

- 동적 SQL에는 else if, else 요소가 없으므로 이를 대체할 목적으로 사용
- 복수의 옵션 중 하나의 선택을 필요로 할 경우 사용
- Java의 switch 구문과 유사

ex. title 값이 있으면 title만을 조회, 만족하는 조건이 없으면 otherwise 조건 수행

```
<select id="findBlogByCondition" parameterType="Blog"
        resultMap="blogMap">
    SELECT *
    FROM tx_blog
    <choose>
        <when test="title != null">
            WHERE title like '%${title}%'
        </when>
        <when test="autho != null and autho.id != null">
            WHERE author_id = #{author.id}
        </when>
        <otherwise>
            WHERE title = '' AND author_id = '0'
        </otherwise>
    </choose>
</select>
```

5-5. 동적 SQL – where

- if 또는 choose 구문을 통해서 해결되지 않는 동적 SQL 구문 처리
- 생성되는 where 조건이 여러 문장일 경우 각 문장 앞에 [WHERE], [AND], [OR] 구문을 생성하거나 삭제
- 조건에 따라 AND, OR로 시작되는 문장이 where 구문의 처음에 위치하면 AND, OR를 삭제하고 where로 대체
- where 구문의 처음에 위치하는 문장이 [AND] 나 [OR]가 아닐 경우 trim 요소를 사용

```
<select id="findBlogByCondition" parameterType="Blog"
        resultMap="blogMap">
    SELECT *
    FROM tx_blog
    <choose>
        <when test="title != null">
            title like '%${title}%'
        </when>
        <when test="autho != null and autho.id != null">
            AND author_id = #{author.id}
        </when>
    </choose>
</select>
```

5-5. 동적 SQL – set

- update 구문에서 동적 set 구문 생성을 위해 사용
- update 구문에서 마지막에 명시된 심표를 제거

```
<update id="updateAuthor" parameterType="Author">
    UPDATE tx_author
    <set>
        <if test="name != null"> name = #{name}, </if>
        <if test="password != null"> password = #{password}, </if>
        <if test="email != null"> email = #{email}, </if>
    </set>
    WHERE id = #{id}
</update>
```



```
<update id="updateAuthor" parameterType="Author">
    UPDATE tx_author
    <trim prefix="SET" suffixOverrides=",">
        <if test="name != null"> name = #{name}, </if>
        <if test="password != null"> password = #{password}, </if>
        <if test="email != null"> email = #{email}, </if>
    </trim>
    WHERE id = #{id}
</update>
```

5-5. 동적 SQL – trim

- 하위 요소의 수행 결과 구문이 있다면 prefix 속성값을 문장의 가장 앞에 추가
- 하위 요소의 수행 결과 구문에 prefixOverrides 속성값과 일치하는 문장이 있다면 이를 제거
- 하위 요소의 수행 결과 구문이 있다면 suffix 속성값을 문장의 가장 뒤에 추가
- 하위 요소의 수행 결과 구문에 suffixOverrides 속성값과 일치하는 문장이 있다면 이를 제거

```
<select id="findBlogByCondition" parameterType="Blog"
        resultMap="blogMap">
    SELECT *
    FROM tx_blog
    <trim prefix="WHERE" prefixOverrides="AND | OR">
        <if test="title != null">
            title LIKE '%${title}%'
        </if>
        <if test="author != null AND author.id != null">
            author_id = #{author.id}
        </if>
    </trim>
</select>
```


5-5. 동적 SQL – foreach

- 배열, 리스트 등 collection 객체에 대해 반복이 필요할 경우 사용
 - 이 객체의 타입을 collection 속성에 정의
- item 속성 값은 하위 요소에서 collection에 저장된 데이터를 참조할 때 사용
- collection 객체를 이용하여 SQL 문장을 작성할 때 가장 앞과 뒤에는 open, close 속성값을 추가
- collection 객체를 이용하여 SQL 문장을 작성할 때 각 데이터 사이에는 separator 속성값을 추가

```
<select id="findAuthorByIds" parameterType="Author">
    SELECT id, password, name, email
    FROM tx_author
    WHERE id IN
        <foreach item="id" separator="," collection="list" open="(" close=")">
            #{id}
        </foreach>
</select>
```

5-6. 어노테이션 매핑

- 매퍼 인터페이스에 어노테이션을 사용하여 설정을 정의
- 패키지과 인터페이스 명을 네임스페이스로 사용하고 메소드명이 SQL id가 됨
- 작성 규칙은 XML 결과 매핑과 유사
 - 어노테이션 결과 매핑은 재사용할 수 없음
- Join 문을 사용할 수 없음 (@One, @Many는 Nested Select 방식)

```
public interface PostMapper {
    @Results({
        @Result(property="id", column="id", id=true),
        @Result(property="subject", column="subject"),
        @Result(property="contents", column="contents"),
        @Result(property="author", column="author_id",
            one=@One(select="com.lectopia.mybatis.blog.AuthorDao.findAuthor")),
        @Result(property="blog", column="blog_id",
            one=@One(select="com.lectopia.mybatis.blog.BlogDao.findBlog"))
    })
    @Select("SELECT id, subject, contents FROM tx_post WHERE blog_id=#{id} ORDER BY id")
    List<Post> findPostsByBlogId(int id);
}
```

5-6. 어노테이션 매핑

- 어노테이션 매핑은 XML 매핑 엘리먼트들을 대부분 대체 가능
- MyBatis는 XML기반의 프레임워크이기 때문에 XML의 각 요소를 이해하면 어노테이션 매핑도 쉽게 적용 가능
- 복잡하고 유연해야 하는 경우에 대해 제한적이기 때문에 간단한 구문에만 적용하는 것이 좋음
- MyBatis에서 제공하는 어노테이션은 표 참조

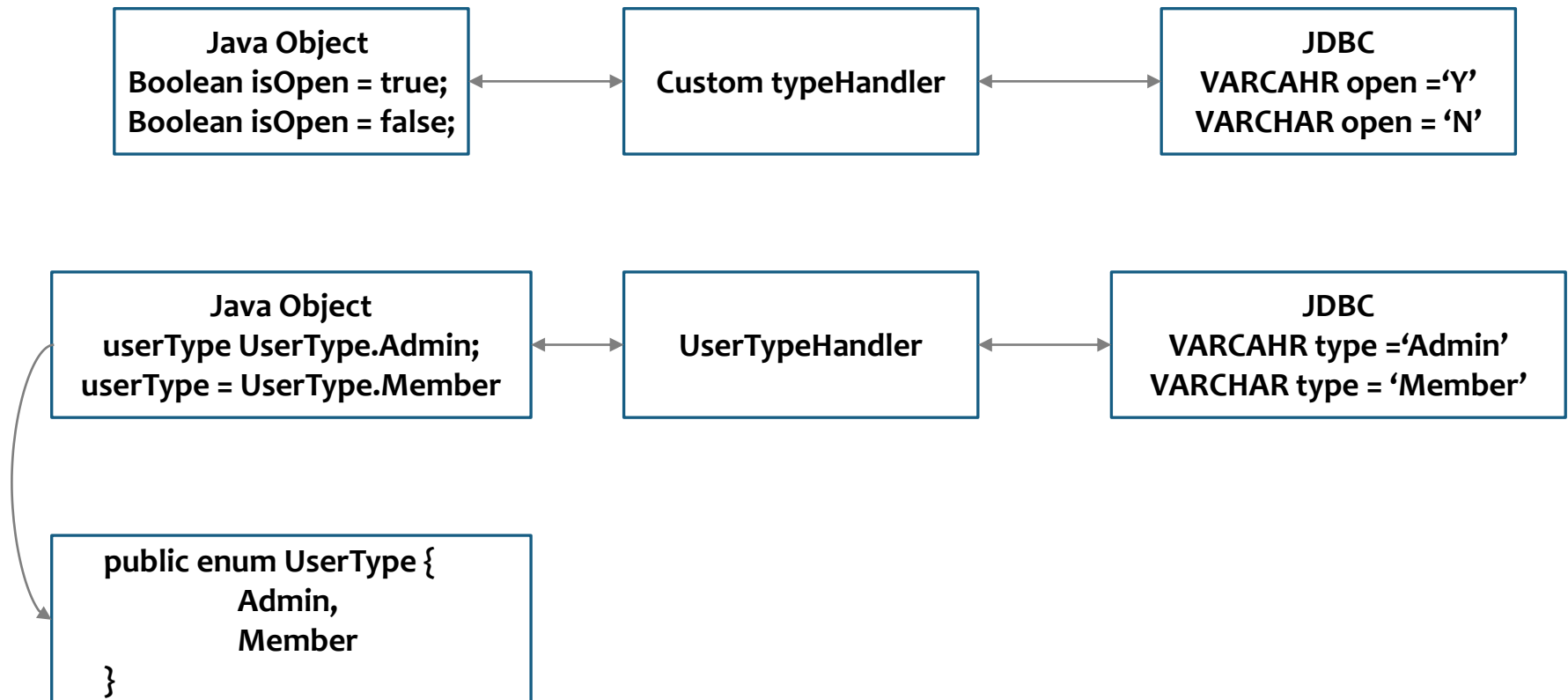
Annotation	대상	XML	설명
@CacheNamespace	Class	<cache>	현 클래스의 캐시정보 설정 (속성) implementation, eviction, flushInterval, size, readWrite
@CacheNamespaceRef	Class	<cacheRef>	다른 클래스의 캐시 정보를 참조 (속성) value
@ConstructorArgs	Method	<constructor>	생성자에 인자를 전달 (속성) value
@Arg	Method	<arg>, <idArg>	생성자의 인자로 전달되는 프로퍼티 값 설정 (속성) id, column, javaType, typeHandler
@TypeDiscriminator	Method	<discriminator>	필드의 값에 따라 하위 매핑결과를 다르게 배치 가능 (속성) column, javaType, jdbcType, typeHandler, cases
@Case	Method	<case>	필드 값에 따라 하위 매핑 결과 지정 (속성) value, type, results

5-6. 어노테이션 매핑

Annotation	대상	XML	설명
@Results	Method	<resultMap>	Result 매핑 리스트. 속성) value
@Result	Method	<result>, <id>	프러퍼티와 DB 컬럼의 매핑 정보 속성) id, column, property, javaType, jdbcType, typeHandler
@One	Method	<association>	1:1 연관관계 매핑. 속성) select
@Many	Method	<collection>	1:N 연관관계 매핑. 속성) select
@Options	Method	Mapped statement 속성	메소드별 수행되는 옵션 속성) useCache, flushCache, resultSetType, statementType, fetchSize, timeout, useGeneratedKeys, keyProperty
@Insert @Update @Delete @Select	Method	<insert> <update> <delete> <select>	Query
@InsertProvider @UpdateProvider @DeleteProvider @SelectProvider	Method	<insert> <update> <delete> <select>	Query를 동적으로 구성하는 Query 제공자 설정 속성) type, method

5-7. Type Handler – 개요

- 매핑문에서 Java 타입과 JDBC 타입간 변환을 담당
- 대부분 MyBatis에서 제공하는 Type Handler로 변환 가능
 - StringTypeHandler, IntegerTypeHandler
- MyBatis에서 지원하지 못하는 경우 사용자 정의 Type Handler 구현
- Java 객체의 boolean 타입을 DB에서는 'Y' 또는 'N'으로 매핑하여 사용 시 유용



5-7. Type Handler – BooleanTypeHandler 구현

```
public interface TypeHandler {
    public void setParameter(PreparedStatement ps, int i, Object param, JdbcType jdbcType)
        throws SQLException;
    public Object getResult(ResultSet rs, String columnName) throws SQLException;
    public Object getResult(CallableStatement cs, int columnIndex) throws SQLException;
}

public class BooleanTypeHandler implements TypeHandler {
    public static final String YES = "YES";
    public static final String NO = "NO";

    @Override
    public void setParameter(PreparedStatement ps, int i, Object param, JdbcType jdbcType)
        throws SQLException {
        if ((Boolean)param)
            ps.setString(i, YES);
        else
            ps.setString(i, NO);
    }

    @Override
    public Object getResult(ResultSet rs, String columnName) throws SQLException {
        if (YES.equals(rs.getString(columnName))) {
            return Boolean.TRUE;
        } else {
            return Boolean.FALSE;
        }
    }

    ...
}
```

Java 타입 → JDBC 타입 변환

JDBC 타입 → Java 타입 변환

5-7. Type Handler – 구현예제

- mybatis-config.xml에 등록하면 전역적으로 사용 가능
- 각 Mapper XML 파일에 등록하면 해당 네임스페이스에서만 사용 가능
- 단일 ResultMap이나 ParameterMap에 등록하면 해당 요소에서만 사용 가능

ex. TypeHandler를 전역으로 정의하여 사용한 예

```
<typeHandler
    handler="com.lectopia.mybatis.blog.handler.BooleanTypeHandler"
    javaType="boolean" jdbcType="VARCHAR" />
```

```
<resultMap id="postMap" type="Post">
    <result property="isTag" column="TAG_YN"
        javaType="boolean" jdbcType="VARCHAR" />
```

javaType, jdbcType이 일치하는 경우 typeHandler 적용 가능

```
#{isTag, javaType=boolean, jdbcType=VARCHAR},
```

인라인 파라미터에도 적용 가능

5-7. Type Handler – 구현예제

ex. 네임스페이스에 정의하여 사용한 예

```
<typeAlias alias="BooleanToYnHandler"
            type="com.lectopia.mybatis.blog.handler.BooleanToYnHandler" />

<parameterMap id="PostParam" class="Post">
  <parameter property="isTag" typeHandler="BooleanToYnHandler" />
</parameterMap>
```

명시적으로 typeHandler 정의

ex. 다음은 단일 ResultMap이나 ParameterMap에 정의하면 사용한 예

```
<resultMap id="postMap" type="Post">
  <result property="isTag" column="TAG_YN"
          typeHandler="com.lectopia.mybatis.blog.handler.BooleanToYnHandler" />
</result>
</resultMap>
```


5-8. 실습

- 실습 과제

- PostMapper: mapper 인터페이스를 통한 (어노테이션 호출) Select, Result 문
- BlogMapper.xml: Advanced Result 매핑
- BlogMapper: 어노테이션을 통한 매핑
- 동적 Query
- Customer Type Handler
 - : 남자, 여자를 속성으로 가진 Gender enum에 대한 TypeHandler
 - : M, F 생성

