- **1장 JPA 소개**
  - **1. SQL과 문제점**
    - 1.1 반복 반복 그리고 반복
    - 1.2 SQL에 의존적인 개발
    - 1.3 JPA와 문제 해결
  - **2. 패러다임의 불일치**
    - 2.1 상속
      - 2.1.1 JPA와 상속
    - 2.2 연관관계
      - 2.2.1 객체를 테이블에 맞추어 모델링
      - 2.2.2 객체 지향 모델링
      - 2.2.3 JPA와 연관관계
    - 2.3 객체 그래프 탐색
      - 2.3.1 JPA와 객체 그래프 탐색
    - 2.4 비교하기
      - 2.4.1 JPA와 비교하기
    - 2.5 정리
  - **3. JPA란 무엇인가?**
    - 3.1 JPA 소개
    - 3.2 JPA를 왜 사용해야 하는가?
  - **ORM 궁금증과 오해 (참고)**

1장 JPA 소개

나는 주로 자바로 애플리케이션을 개발하고 관계형 데이터베이스를 데이터 저장소로 사용하면서 오랫동안 SQL을 다루었다. 초기에는 JDBC API를 직접 사용해서 코딩하기도 했는데, 애플리케이션의 비즈니스 로직보다 SQL과 JDBC API를 작성하는데 더 많은 시간을 보냈다. 그러다가 iBatis(지금은 MyBatis)나 스프링의 JdbcTemplate 같은 SQL Mapper를 사용하면서 JDBC API 사용 코드를 많이 줄일 수 있었다.

하지만 여전히 등록, 수정, 삭제, 조회(CRUD)용 SQL은 반복해서 작성해야 했고, 이런 과정은 너무 지루하고 비생산적이었다. 그래서 테이블 이름을 입력하면 CRUD SQL을 자동으로 생성해주는 도구를 만들어서 사용하기도 했는데, 개발 초기에는 사용할 만했지만, 개발이 어느 정도 진행된 상태에서 애플리케이션의 요구사항이 추가되는 것까지 해결해 주지는 못했다.

나는 객체 모델링을 공부하면서 큰 고민에 빠지게 되었는데, 왜 실무에서 테이블 설계는 다들 열심히 하면서 제대로 된 객체 모델링은 하지 않을까? 왜 객체 지향의 장점을 포기하고 객체를 단순히 테이블에 맞추어 데이터 전달 역할만 하도록 개발할까? 라는 의문이 들었다.

그래서 새로운 프로젝트에 자신 있게 객체 모델링을 적용해 보았지만, 객체 모델링을 세밀하게 진행할수록 객체를 데이터베이스에 저장하거나 조회하기는 점점 더 어려워졌고, 객체와 관계형 데이터베이스의 차이를 메우기 위해 더 많은 SQL을 작성해야 했다. 결국, 객체 모델링을 SQL로 풀어내는데 너무 많은 코드와 노력이 필요했고, 객체 모델은 점점 데이터 중심의 모델로 변해갔다.

그런 중에 이미 많은 자바 개발자들이 같은 고민을 하고 있다는 것을 알게 되었고, 객체와 관계형 데이터베이스 간의 차이를 중간에서 해결해주는 ORM(Object-relational mapping) 프레임워크라는 것도 알게 되었다. 참고로 지금부터 설명할 JPA는 자바 진영의 ORM 기술 표준이다.

JPA는 지루하고 반복적인 CRUD SQL을 알아서 처리해 줄 뿐만 아니라 객체 모델링과 관계형 데이터베이스 사이의 차이점도 해결해 주었다.

JPA는 실행 시점에 자동으로 SQL을 만들어서 실행하는데, JPA를 사용하는 개발자는 SQL을 직접 작성하는 것이 아니라 어떤 SQL이 실행될지 생각만 하면 된다. 참고로 JPA가 실행하는 SQL은 쉽게 예측할 수 있다.

실무에 JPA를 적용하면서 처음에는 어려운 점도 있었지만, JPA를 사용해서 얻은 보상은 정말 컸다. 우선 CRUD SQL을 작성할 필요가 없고, 조회된 결과를 객체로 매핑하는 작업도 대부분 자동으로 처리해 주므로 데이터 저장 계층에 작성해야 할 코드가 1/3로 줄어들었다.

처음에는 성능에 대한 걱정도 있었는데 대부분 대안이 있었다. 예를 들어 JPA가 제공하는 네이티브 SQL이라는 기능을 사용해서 직접 SQL을 작성할 수도 있었고, 데이터베이스 쿼리 힌트도 사용할 수 있는 방법이 있었다. 결국, 애플리케이션보다는 데이터베이스 조회 성능이 이슈였는데 이것은 JPA의 문제라기보다는 SQL을 직접 사용해도 발생하는 문제들이었다.

JPA를 사용해서 얻은 가장 큰 성과는 애플리케이션을 SQL이 아닌 객체 중심으로 개발하니 생산성과 유지보수가 확연히 좋아졌고 테스트를 작성하기도 편리해진 점이다. 이런 장점 덕분에 버그도 많이 줄어들었다. 그리고 개발 단계에서 MySQL 데이터베이스를 사용하다가 오픈 시점에 Oracle 데이터베이스를 사용하기로 정책이 변경된 적이 있었는데, JPA 덕분에 코드를 거의 수정하지 않고 데이터베이스를 손쉽게 변경할 수 있었다. SQL을 직접 다룰 때는 상상하기 어려운 일이었다.

JPA는 자바 진영에서 힘을 모아서 만든 ORM 기술 표준이다. 그리고 스프링 진영에서도 스프링 프레임워크 자체는 물론이고, 스프링 데이터 JPA라는 기술로 JPA를 적극적으로 지원한다. 또한, 전자정부 표준 프레임워크의 ORM 기술도 JPA를 사용한다.

반복적인 CRUD SQL을 작성하고 객체를 SQL에 매핑하는데 시간을 보내기에는 우리의 시간이 너무 아깝다. 이미 많은 자바 개발자들이 오랫동안 비슷한 고민을 해왔고 문제를 해결하려고 많은 노력을 기울여왔다. 그리고 그 노력의 결정체가 바로 JPA다. JPA는 표준 명세만 570페이지에 달하고, JPA를 구현한 하이버네이트는 이미 10년 이상 지속해서 개발되고 있으며, 핵심 모듈의 코드 수가 이미 십만 라인을 넘어섰다. 귀찮은 문제들은 이제 JPA에게 맡기고 더 좋은 객체 모델링과 더 많은 테스트를 작성하는데 우리의 시간을 보내자. 개발자는 SQL Mapper가 아니다.

나는 JPA를 사용해서 애플리케이션을 개발하는 지금이 너무 즐겁다. 독자분들도 JPA를 한 번만 제대로 사용해보면 다시는 과거로 돌아가고 싶어 하지 않으리라고 생각한다.

그럼 지금부터 SQL을 직접 다룰 때 어떤 문제가 발생하는지, 객체와 관계형 데이터베이스 사이에는 어떤 차이가 있는지 자세히 알아보자.

1. SQL과 문제점

관계형 데이터베이스는 가장 대중적이고 신뢰할 만한 안전한 데이터 저장소다. 그래서 자바로 개발하는 애플리케이션은 대부분 관계형 데이터베이스를 데이터 저장소로 사용한다.

데이터베이스에 데이터를 관리하려면 SQL을 사용해야 한다. 자바로 작성한 애플리케이션은 JDBC API를 사용해서 SQL을 데이터베이스에 전달하는데 자바 서버 개발자들에게 이것은 너무나 당연한 이야기이고 대부분 능숙하게 SQL을 다룰 줄 안다.

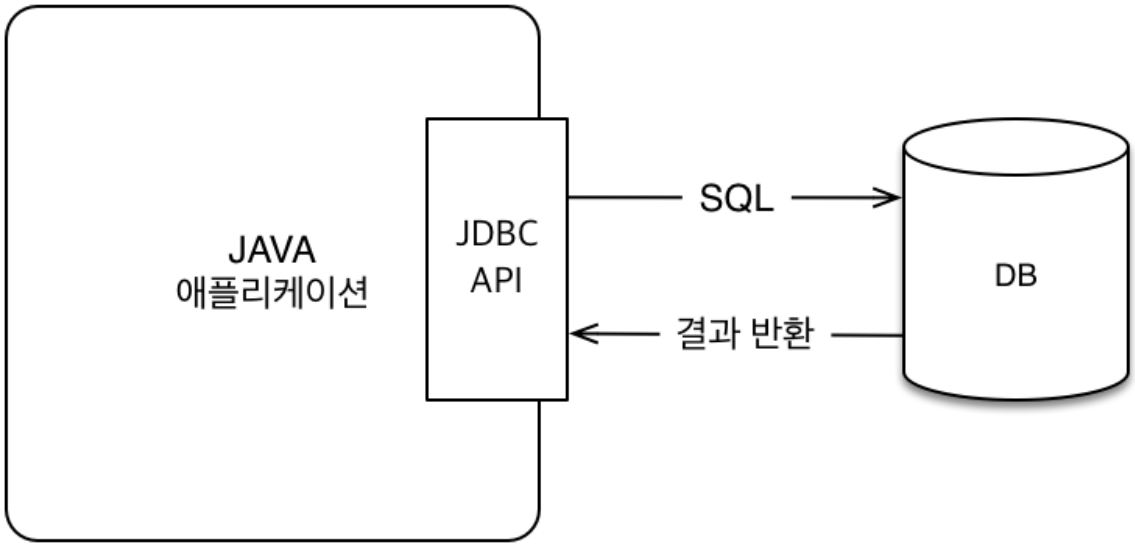

그림 - JDBC API와 SQL

1.1 반복 반복 그리고 반복

SQL을 직접 다룰 때의 문제점을 알아보기 위해 자바와 관계형 데이터베이스를 사용해서 회원 관리 기능을 개발해보자. 회원 테이블은 이미 만들어져 있다고 가정하고 회원을 CRUD(등록, 수정, 삭제, 조회)하는 기능을 개발해보자.

먼저 자바에서 사용할 회원(`Member`) 객체를 만들자

```
public class Member {

    private String memberId;
    private String name;
    ...
}
```

회원 객체를 데이터베이스에 관리할 목적으로 회원용 DAO(데이터 접근 객체)를 만들자

```
public class MemberDAO {
    public Member find(String memberId){...}
}
```

이제 `MemberDAO` 의 `find()` 메서드를 완성해서 회원을 조회하는 기능을 개발해보자. 보통 다음 순서로 개발을 진행할 것이다.

*1. 회원 조회용 SQL을 작성한다.

```
SELECT MEMBER_ID, NAME FROM MEMBER M WHERE MEMBER_ID = ?
```

*2. JDBC API를 사용해서 SQL을 실행한다.

```
ResultSet rs = stmt.executeQuery(sql);
```

*3. 조회 결과를 `Member` 객체로 매핑한다.

```
String memberId = rs.getString("MEMBER_ID");
String name = rs.getString("NAME");

Member member = new Member();
member.setMemberId(memberId);
member.setName(name);
...
```

회원 조회 기능을 완성했다. 다음으로 회원 등록 기능을 만들어보자.

```
public class MemberDAO {
    public Member find(String memberId){...}
    public void save(Member member){...} //추가
}
```

*1. 회원 등록용 SQL을 작성한다.

```
String sql = "INSERT INTO MEMBER(MEMBER_ID, NAME) VALUES(?,?);
```

*2. 회원 객체의 값을 꺼내서 등록 SQL에 전달한다.

```
pstmt.setString(1, member.getMemberId());
pstmt.setString(2, member.getName());
```

*3. JDBC API를 사용해서 SQL을 실행한다.

```
pstmt.executeUpdate(sql);
```

회원을 조회하는 기능과 등록하는 기능을 만들었다. 다음으로 회원을 수정하고 삭제하는 기능도 추가해 보자. 아마도 SQL을 작성하고 JDBC API를 사용하는 비슷한 일을 반복해야 할 것이다.

회원 객체를 데이터베이스가 아닌 자바 컬렉션에 보관한다면 어떨까? 컬렉션은 다음 한줄로 객체를 저장할 수 있다.

```
list.add(member);
```

하지만 데이터베이스는 객체 구조와는 다른 데이터 중심의 구조를 가지므로 객체를 데이터베이스에 직접 저장하거나 조회할 수는 없다. 따라서 개발자가 객체 지향 애플리케이션과 데이터베이스 중간에서 SQL과 JDBC API를 사용해서 변환 작업을 직접 해주어야 한다.

문제는 객체를 데이터베이스에 CRUD 하려면 너무 많은 SQL과 JDBC API를 코드로 작성해야 한다는 점이다. 그리고 테이블마다 이런 비슷한 일을 반복해야 하는데, 개발하려는 애플리케이션에서 사용하는 데이터베이스 테이블이 100개라면 무수히 많은 SQL을 작성해야 하고 이런 비슷한 일을 100번은 더 반복해야 한다. 데이터 접근 계층(DAO)을 개발하는 일은 이렇듯 지루함과 반복의 연속이다.

1.2 SQL에 의존적인 개발

앞에서 만든 회원 객체를 관리하는 `MemberDAO`를 완성하고 애플리케이션의 나머지 기능도 개발을 완료했다. 그런데 갑자기 회원의 연락처도 함께 저장해달라는 요구사항이 추가되었다.

등록 코드 변경

회원의 연락처를 추가하려고 회원 테이블에 `TEL` 컬럼을 추가하고 회원 객체에 `tel` 필드를 추가했다.

```
public class Member {

    private String memberId;
    private String name;
    private String tel; //추가 //**
    ...
}
```

다음으로 연락처를 저장할 수 있도록 INSERT SQL을 수정하고

```
String sql = "INSERT INTO MEMBER(MEMBER_ID, NAME, TEL) VALUES(?,?,?);
```

회원 객체의 연락처 값을 꺼내서 등록 SQL에 전달했다.

```
pstmt.setString(3, member.getTel());
```

연락처를 데이터베이스에 저장하려고 SQL과 JDBC API를 수정했다. 그리고 연락처가 잘 저장되는지 테스트하고 데이터베이스에 연락처 데이터가 저장된 것도 확인했다.

조회 코드 변경

다음으로 회원 조회 화면을 수정해서 연락처 필드가 출력되도록 했다. 그런데 모든 연락처의 값이 `null` 로 출력된다. 생각해보니 조회 SQL에 연락처 컬럼을 추가하지 않았다.

회원 조회용 SQL을 수정하고

```
SELECT MEMBER_ID, NAME, TEL FROM MEMBER WHERE MEMBER_ID = ?
```

연락처의 조회 결과를 `Member` 객체에 추가로 매핑했다.

```
...
String tel = rs.getString("TEL");
member.setTel(tel); //추가
...
```

이제 화면에 연락처 값이 출력된다.

수정 코드 변경

기능이 잘 동작한다고 생각했는데 이번에는 연락처가 수정되지 않는 버그가 발견되었다고 한다. 자바 코드를 보니 `MemberDAO.update(member)` 메서드에 수정할 회원 정보와 연락처를 잘 전달했다. `MemberDAO` 를 열어서 UPDATE SQL을 확인해보니 `TEL` 컬럼을 추가하지 않아서 연락처가 수정 되지 않는 문제였다. UPDATE SQL과 `MemberDAO.update()` 의 일부 코드를 변경해서 연락처가 수정되도록 했다.

만약 회원 객체를 데이터베이스가 아닌 자바 컬렉션에 보관했다면 필드를 추가한다고 해서 이렇게 많은 코드를 수정할 필요는 없을 것이다.

```
list.add(member); //등록
Member member = list.get(xxx); //조회
member.setTel("xxx") //수정
```

연관된 객체

회원은 어떤 한 팀에 필수로 소속되어야 한다는 요구사항이 추가되었다. 팀 모듈을 전담하는 개발자가 팀을 관리하는 코드를 개발하고 커밋했다. 코드를 받아보니 `Member` 객체에 `team` 필드가 추가되어 있다.

회원 정보를 화면에 출력할 때 연관된 팀 이름도 함께 출력하는 기능을 추가해보자.

```
class Member {

    private String memberId;
    private String name;
    private String tel;
    private Team team;  //추가 //**
    ...
}

//추가된 팀
class Team {
    ...
    private String teamName;
    ...
}
```

다음 코드를 추가해서 화면에 팀의 이름을 출력했다.

```
이름 : member.getName();
소속 팀 : member.getTeam().getTeamName(); //추가 //**
```

코드를 실행해보니 `member.getTeam()`의 값이 항상 `null`이다. 회원과 연관된 팀이 없어서 그럴것이라 생각하고 데이터베이스를 확인해보니 모든 회원이 팀에 소속되어 있다.

문제를 찾다가 `MemberDAO`에 `findWithTeam()`이라는 새로운 메서드가 추가된 것을 확인했다.

```
public class MemberDAO {
    public Member find(String memberId){...}
    public Member findWithTeam(String memberId){...} //**
}
```

`MemberDAO` 코드를 열어서 확인해보니 회원을 출력할 때 사용하는 `find()` 메서드는 회원만 조회하는 SQL을 그대로 유지했고

== 회원만 조회하는 SQL ==

```
SELECT MEMBER_ID, NAME, TEL FROM MEMBER M
```

새로운 `findWithTeam()` 메서드는 회원과 연관된 팀을 함께 조회했다.

== 회원과 팀을 함께 조회하는 SQL ==

```
SELECT M.MEMBER_ID, M.NAME, M.TEL, T.TEAM_ID, T.TEAM_NAME
FROM MEMBER M
JOIN TEAM T
  ON M.TEAM_ID = T.TEAM_ID
```

결국 DAO를 열어서 SQL을 확인하고 나서야 원인을 알 수 있었고, 회원 조회 코드를 `MemberDAO.find()` 에서 `MemberDAO.findWithTeam()` 으로 변경해서 문제를 해결했다.

SQL과 문제점 - 정리

`Member` 객체가 연관된 `Team` 객체를 사용할 수 있을지 없을지는 전적으로 사용하는 SQL에 달려있다. 이런 방식의 가장 큰 문제는 데이터 접근 계층을 사용해서 SQL을 숨겨도 어쩔 수 없이 DAO를 열어서 어떤 SQL이 실행되는지 확인해야 한다는 점이다.

`Member` 나 `Team` 처럼 비즈니스 요구사항을 모델링한 객체를 엔티티라 하는데, 지금처럼 **SQL에 모든 것을 의존하는 상황에서는 개발자들이 엔티티를 신뢰하고 사용할 수 없다.** 대신에 DAO를 열어서 어떤 SQL이 실행되고 어떤 객체들이 함께 조회되는지 일일이 확인해야 한다. 이것은 진정한 의미의 계층 분할이 아니다. 물리적으로 SQL과 JDBC API를 데이터 접근 계층에 숨기는 데 성공했을지는 몰라도 논리적으로는 엔티티와 아주 강한 의존관계를 가지고 있다. 이런 강한 의존관계 때문에 회원을 조회할 때는 물론이고 회원 객체에 필드를 하나 추가할 때도 DAO의 CRUD 코드와 SQL 대부분을 변경해야 하는 문제가 발생한다.

애플리케이션에서 SQL을 직접 다룰 때 발생하는 문제점을 요약하면 다음과 같다.

- 진정한 의미의 계층 분할이 어렵다.
- 엔티티를 신뢰할 수 없다.
- SQL에 의존적인 개발을 피하기 어렵다.

1.3 JPA와 문제 해결

이 책에서 소개할 JPA는 이런 문제들을 어떻게 해결할까? 참고로 JPA에 대한 자세한 소개는 이 장 뒷부분에 하겠다. 우선은 JPA가 문제를 어떻게 해결하는지 간단히 알아보자.

JPA를 사용하면 객체를 데이터베이스에 저장하고 관리할 때, 개발자가 직접 SQL을 작성하는 것이 아니라 JPA가 제공하는 API를 사용하면 된다. 그러면 JPA가 개발자 대신에 적절한 SQL을 생성해서 데이터베이스에 전달한다.

JPA가 제공하는 CRUD API를 간단히 알아보자.

저장 기능

```
jpa.persist(member); //저장
```

`persist()` 메서드는 객체를 데이터베이스에 저장한다. 이 메서드를 호출하면 JPA가 객체와 매핑정보를 보고 적절한 INSERT SQL을 생성해서 데이터베이스에 전달한다. 매핑정보는 어떤 객체를 어떤 테이블에 관리할지 정의한 정보인데 자세한 내용은 다음 장에서 설명한다.

조회 기능

```
String memberId = "helloId";
Member member = jpa.find(Member.class, memberId); //조회
```

`find()` 메서드는 객체 하나를 데이터베이스에서 조회한다. JPA는 객체와 매핑정보를 보고 적절한 SELECT SQL을 생성해서 데이터베이스에 전달하고 그 결과로 `Member` 객체를 생성해서 반환한다.

수정 기능

```
Member member = jpa.find(Member.class, memberId);
member.setName("이름변경") //수정
```

JPA는 별도의 수정 메서드를 제공하지 않는다. 대신에 객체를 조회해서 값을 변경만 하면 트랜잭션을 커밋할 때 데이터베이스에 적절한 UPDATE SQL이 전달된다. 이 마법 같은 일이 어떻게 가능할까? 자세한 내용은 3. 영속성 관리 장에서 설명한다.

연관된 객체 조회

```
Member member = jpa.find(Member.class, memberId);
Team team = member.getTeam(); //연관된 객체 조회
```

JPA는 연관된 객체를 사용하는 시점에 적절한 SELECT SQL을 실행한다. 따라서 JPA를 사용하면 연관된 객체를 마음껏 조회할 수 있다. 이 마법 같은 일이 어떻게 가능할까? 자세한 내용은 9. 프록시와 연관관계 관리 장에서 설명한다.

지금까지 JPA의 CRUD API를 간단히 알아보았다. 수정 기능과 연관된 객체 조회에서 설명한 것처럼 JPA는 SQL을 개발자 대신 작성해서 실행해주는 것 이상의 기능들을 제공한다.

다음은 객체와 관계형 데이터베이스의 패러다임 차이 때문에 발생하는 다양한 문제를 살펴보고 JPA는 이런 문제들을 어떻게 해결하는지 알아보자.

2. 패러다임의 불일치

애플리케이션은 발전하면서 그 내부의 복잡성도 점점 커진다. 지속 가능한 애플리케이션을 개발하는 일은 끊임없이 증가하는 복잡성 과의 싸움이다. 복잡성을 제어하지 못하면 결국 유지 보수하기 어려운 애플리케이션이 된다.

객체 지향 프로그래밍은 추상화, 캡슐화, 정보은닉, 상속, 다형성 등 시스템의 복잡성을 제어할 수 있는 다양한 장치들을 제공한다. 그래서 현대의 복잡한 애플리케이션은 대부분 객체 지향 언어로 개발한다.

비즈니스 요구사항을 정의한 도메인 모델도 객체로 모델링하면 객체 지향 언어가 가진 장점들을 활용할 수 있다. 문제는 이렇게 정의한 도메인 모델을 저장할 때 발생한다. 예를 들어 특정 유저가 시스템에 회원 가입하면 회원이라는 객체 인스턴스를 생성한 후에 이 객체를 메모리가 아닌 어딘가에 영구 보관해야 한다.

객체는 속성(필드)과 기능(메서드)을 가진다. 객체의 기능은 클래스에 정의되어 있으므로 객체 인스턴스의 상태인 속성만 저장했다가 필요할 때 불러와서 복구하면 된다.

객체가 단순하면 객체의 모든 속성 값을 꺼내서 파일이나 데이터베이스에 저장하면 되지만, 부모 객체를 상속받았거나, 다른 객체를 참조하고 있다면 객체의 상태를 저장하기는 쉽지 않다. 예를 들어 회원 객체를 저장해야 하는데 회원 객체가 팀 객체를 참조하고 있다면, 회원 객체를 저장할 때 팀 객체도 함께 저장해야 한다. 단순히 회원 객체만 저장하면 참조하는 팀 객체를 잃어버리는 문제가 발생한다.

자바는 이런 문제까지 고려해서 객체를 파일로 저장하는 직열화 기능과 저장된 파일을 객체로 복구하는 역 직열화 기능을 지원한다. 하지만 이 방법은 직열화된 객체를 검색하기 어렵다는 문제가 있으므로 현실성이 없다.

현실적인 대안은 관계형 데이터베이스에 객체를 저장하는 것인데, 관계형 데이터베이스는 데이터 중심으로 구조화 되어 있고, 집합적인 사고를 요구한다. 그리고 객체 지향에서 이야기하는 추상화, 상속, 다형성 같은 개념이 없다.

객체와 관계형 데이터베이스는 지향하는 목적이 서로 다르므로 둘의 기능과 표현 방법도 다르다. 이것을 객체와 관계형 데이터베이스의 패러다임 불일치 문제라 한다. 따라서 객체 구조를 테이블 구조에 저장하는데는 한계가 있다.

애플리케이션은 자바라는 객체 지향 언어로 개발하고 데이터는 관계형 데이터베이스에 저장해야 한다면, 패러다임의 불일치 문제를 개발자가 중간에서 해결해야 한다. 문제는 이런 객체와 관계형 데이터베이스 사이의 패러다임 불일치 문제를 해결하는데 너무 많은 시간과 코드를 소비하는 데 있다.

지금부터 패러다임의 불일치로 인해 발생하는 문제를 구체적으로 살펴보자. 그리고 JPA를 통한 해결책도 함께 알아보자.

2.1 상속

객체는 상속이라는 기능을 가지고 있지만 테이블은 상속이라는 기능이 없다. (일부 데이터베이스는 상속 기능을 지원하지만 객체의 상속과는 약간 다르다.)

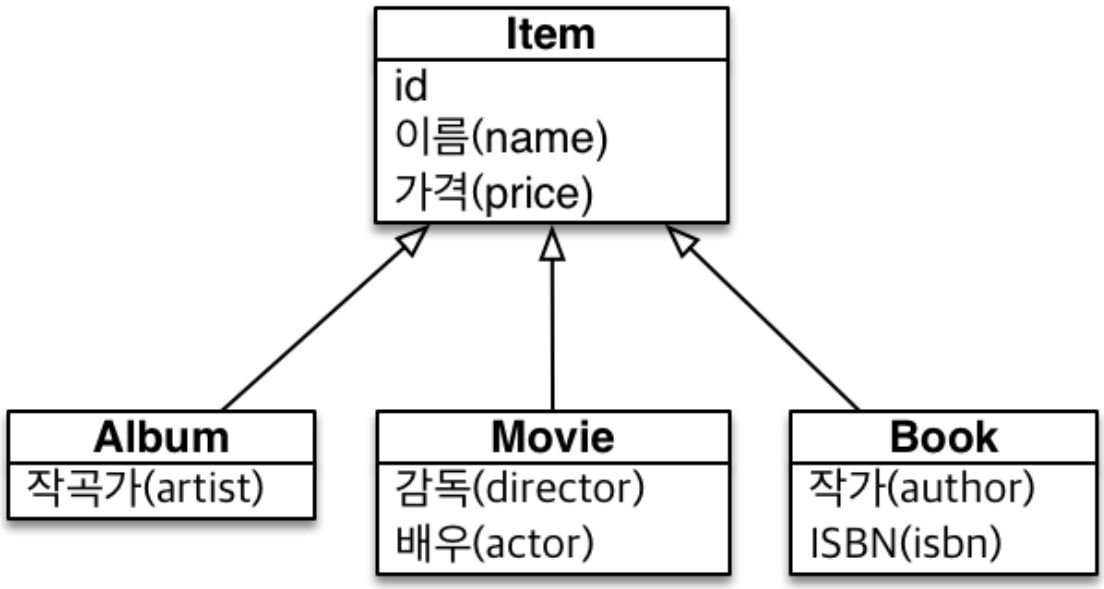

그림 - 객체 상속 모델

그나마 데이터베이스 모델링에서 이야기하는 슈퍼타입 서브타입 관계를 사용하면 객체 상속과 가장 유사한 형태로 테이블을 설계할 수 있다. 여기서 `ITEM` 테이블의 `DTYPE` 컬럼을 사용해서 어떤 자식 테이블과 관계가 있는지 정의했다. 예를 들어 `DTYPE` 의 값이 `MOVIE` 면 영화 테이블과 관계가 있다.

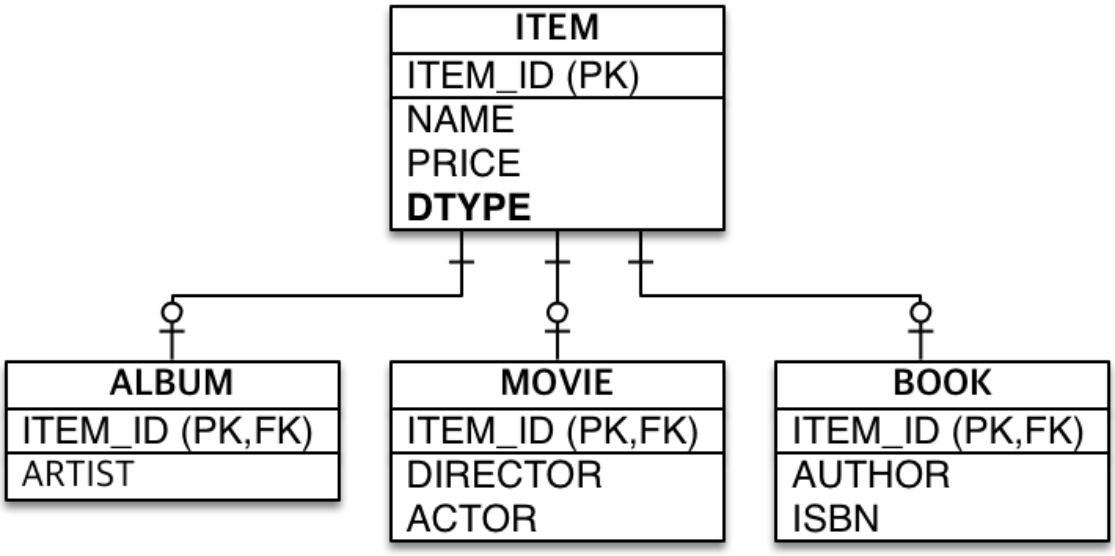

그림 - 객체 상속 테이블 모델

== 객체 모델 ==

```
abstract class Item {
    Long id;
    String name;
```

```
    int price;
}

class Album extends Item {
    String artist;
}

class Movie extends Item {
    String director;
    String actor;
}

class Book extends Item {
    String author;
    String isbn;
}
```

`Album` 객체를 저장하려면 이 객체를 분해해서 다음 두 SQL을 만들어야 한다.

```
INSERT INTO ITEM ...
INSERT INTO ALBUM ...
```

`Movie` 객체도 마찬가지다.

```
INSERT INTO ITEM ...
INSERT INTO MOVIE ...
```

JDBC API를 사용해서 이 코드를 완성하려면 부모 객체에서 부모 데이터만 꺼내서 `ITEM` 용 INSERT SQL을 작성하고 자식 객체에서 자식 데이터만 꺼내서 `ALBUM` 용 INSERT SQL을 작성해야 하는데, 작성해야할 코드량이 만만치 않다. 그리고 자식 타입에 따라서 `DTYPE` 도 저장해야 한다.

조회하는 것도 쉬운 일은 아니다. 예를 들어 `Album` 을 조회한다면 `ITEM` 과 `ALBUM` 테이블을 조인해서 조회한 다음 그 결과로 `Album` 객체를 생성해야 한다.

이런 과정이 모두 패러다임의 불일치를 해결하려고 소모하는 비용이다. 만약 해당 객체들을 데이터베이스가 아닌 자바 컬렉션에 보관한다면 부모 자식이나 타입에 대한 고민 없이 해당 컬렉션을 그냥 사용하면 된다.

```
list.add(album);
list.add(movie);
```

```
Album album = list.get(albumId);
```

2.1.1 JPA와 상속

JPA는 상속과 관련된 패러다임의 불일치 문제를 개발자 대신 해결해준다. 개발자는 마치 자바 컬렉션에 객체를 저장하듯이 JPA에게 객체를 저장하면 된다.

JPA를 사용해서 `Item`을 상속한 `Album` 객체를 저장해보자. 앞서 설명한 `persist()` 메서드를 사용해서 객체를 저장하면 된다.

```
jpa.persist(album);
```

JPA는 다음 SQL을 실행해서 객체를 `ITEM`, `ALBUM` 두 테이블에 나누어 저장한다.

```
INSERT INTO ITEM ...
INSERT INTO ALBUM ...
```

다음으로 `Album` 객체를 조회해보자. 앞서 설명한 `find()` 메서드를 사용해서 객체를 조회하면 된다.

```
String albumId = "id100";
Album album = jpa.find(Album.class, albumId);
```

JPA는 `ITEM`과 `ALBUM` 두 테이블을 조인해서 필요한 데이터를 조회하고 그 결과를 반환한다.

```
SELECT I.*, A.*
  FROM ITEM I
  JOIN ALBUM A ON I.ITEM_ID = A.ITEM_ID
```

2.2 연관관계

객체는 참조를 사용해서 다른 객체와 연관관계를 가지고 **참조에 접근해서 연관된 객체를 조회**한다. 반면에 **테이블은 외래 키**를 사용해서 다른 테이블과 연관관계를 가지고 **조인을 사용해서 연관된 테이블을 조회**한다.

참조를 사용하는 객체와 외래 키를 사용하는 관계형 데이터베이스 사이의 패러다임 불일치는 객체 지향 모델링을 거의 포기하게 만들 정도로 극복하기 어렵다. 예제를 통해 문제점을 파악해보자.

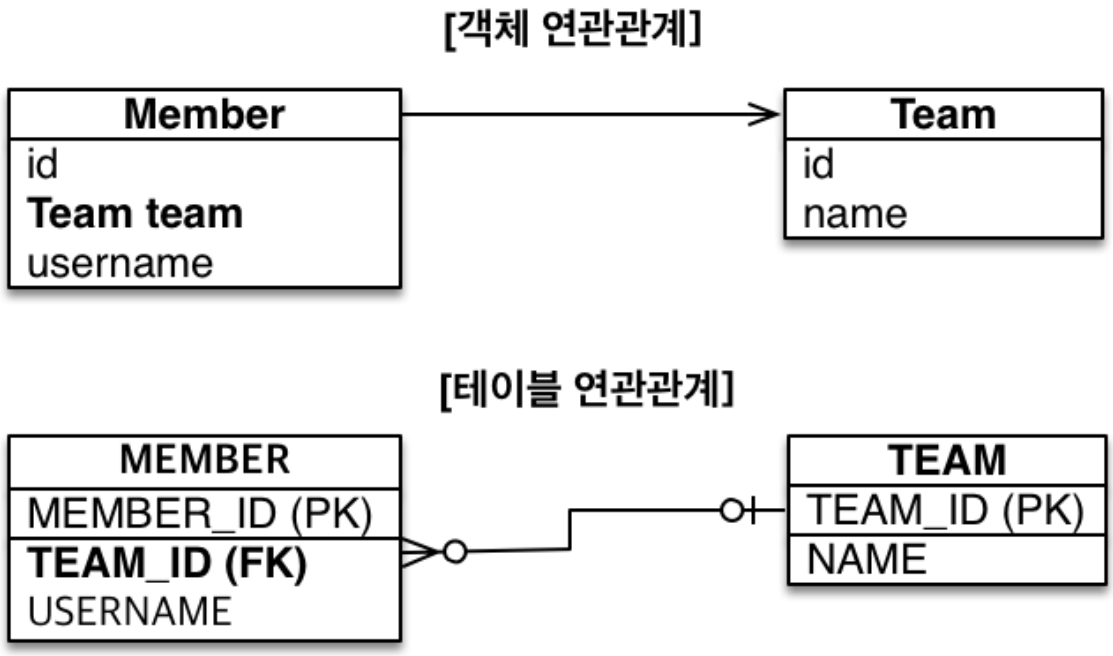

그림 - 연관관계

`Member` 객체는 `Member.team` 필드에 `Team` 객체의 참조를 보관해서 `Team` 객체와 관계를 맺는다. 따라서 이 참조 필드에 접근하면 `Member` 와 연관된 `Team` 을 조회할 수 있다.

```
class Member {
    Team team;
    ...
}

class Team {
    ...
}
```

```
member.getTeam(); //member -> team 접근
```

`MEMBER` 테이블은 `MEMBER.TEAM_ID` 외래 키 컬럼을 사용해서 `TEAM` 테이블과 관계를 맺는다. 이 외래 키를 사용해서 `MEMBER` 테이블과 `TEAM` 테이블을 조인하면 `MEMBER` 테이블과 연관된 `TEAM` 테이블을 조회할 수 있다.

```
SELECT M.*, T.*
  FROM MEMBER M
  JOIN TEAM T ON M.TEAM_ID = T.TEAM_ID
```

조금 어려운 문제도 있는데, 객체는 참조가 있는 방향으로만 조회할 수 있다. 방금 예에서 `member.getTeam()` 은 가능하지만 반대 방향인 `team.getMember()` 는 참조가 없으므로 불가능하다. 반면에 테이블은 외래 키 하나로 `MEMBER JOIN TEAM` 도 가능하지만 `TEAM JOIN MEMBER` 도 가능하다.

2.2.1 객체를 테이블에 맞추어 모델링

객체와 테이블의 차이를 알아보기 위해 객체를 단순히 테이블에 맞추어 모델링 해보자.

```
class Member {
    String id;       //MEMBER_ID 컬럼 사용
    Long teamId;     //TEAM_ID FK 컬럼 사용 //**
    String username;//USERNAME 컬럼 사용
}

class Team {
    Long id;      //TEAM_ID PK 사용
    String name; //NAME 컬럼 사용
}
```

`MEMBER` 테이블의 컬럼을 그대로 가져와서 `Member` 클래스를 만들었다. 이렇게 객체를 테이블에 맞추어 모델링 하면 객체를 테이블에 저장하거나 조회할 때는 편리하다.

그런데 여기서 `TEAM_ID` 외래 키의 값을 그대로 보관하는 `teamId` 필드에는 문제가 있다. 관계형 데이터베이스는 조인이라는 기능이 있으므로 외래 키의 값을 그대로 보관해도 된다. 하지만 객체는 연관된 객체의 참조를 보관해야 다음 처럼 참조를 통해 연관된 객체를 찾을 수 있다.

```
Team team = member.getTeam();
```

특정 회원이 소속된 팀을 조회하는 가장 객체 지향적인 방법은 이처럼 참조를 사용하는 것이다.

`Member.teamId` 필드처럼 `TEAM_ID` 외래 키까지 관계형 데이터베이스가 사용하는 방식에 맞추면 `Member` 객체와 연관된 `Team` 객체를 참조를 통해서 조회할 수 없다. 이런 방식을 따르면 좋은 객체 모델링은 기대하기 어렵고 결국 객체 지향의 특징을 잃어버리게 된다.

2.2.2 객체 지향 모델링

객체는 참조를 통해서 관계를 맺는다. 따라서 다음처럼 참조를 사용하도록 모델링해야 한다.

```
class Member {
    String id;       //MEMBER_ID 컬럼 사용
    Team team;       //참조로 연관관계를 맺는다. //**
    String username;//USERNAME 컬럼 사용
}

class Team {
    Long id;     //TEAM_ID PK 사용
    String name; //NAME 컬럼 사용
}
```

`Member.team` 필드를 보면 외래 키의 값을 그대로 보관하는 것이 아니라 연관된 `Team`의 참조를 보관한다. 이제 회원과 연관된 팀을 조회할 수 있다.

```
Team team = member.getTeam();
```

그런데 이처럼 객체 지향 모델링을 사용하면 객체를 테이블에 저장하거나 조회하기가 쉽지 않다. 왜냐하면 `Member` 객체는 `team` 필드로 연관관계를 맺고 `MEMBER` 테이블은 `TEAM_ID` 외래 키로 연관관계를 맺기 때문인데, 객체 모델은 외래 키가 필요 없고 단지 참조만 있으면 된다. 반면에 테이블은 참조가 필요 없고 외래 키만 있으면 된다.

결국, 개발자가 중간에서 변환 역할을 해야 한다.

저장

객체를 데이터베이스에 저장하려면 `team` 필드를 `TEAM_ID` 외래 키 값으로 변환해야 한다.

다음처럼 외래 키 값을 찾아서 INSERT SQL을 만들어야 하는데 `MEMBER` 테이블에 저장해야 할 `TEAM_ID` 외래 키는 `TEAM` 테이블의 기본 키 이므로 `member.getTeam().getId()`로 구할 수 있다.

```
member.getId();           //MEMBER_ID PK에 저장
member.getTeam().getId(); //TEAM_ID FK에 저장
member.getUsername();     //USERNAME 컬럼에 저장
```

조회

조회할 때는 `TEAM_ID` 외래 키 값을 `team` 참조로 변환해서 객체에 보관해야 한다.

```
SELECT M.*, T.*
  FROM MEMBER M
  JOIN TEAM T ON M.TEAM_ID = T.TEAM_ID
```

```
public Member find(String memberId) {
    ...
    Member member = new Member();
    ...
    //데이터베이스에서 조회한 회원 관련 정보를 모두 입력

    Team team = new Team();
    ...
    //데이터베이스에서 조회한 팀 관련 정보를 모두 입력

    //회원과 팀 관계 설정
    member.setTeam(team);
    return member;
}
```

이 과정도 모두 패러다임 불일치를 해결하려고 소모하는 비용이다. 만약 자바 컬렉션에 회원 객체를 저장한다면 이런 비용이 전혀 들지 않는다.

2.2.3 JPA와 연관관계

JPA는 연관관계와 관련된 패러다임의 불일치 문제를 해결해준다. 다음 코드를 보자.

```
member.setTeam(team); //회원과 팀 연관관계 설정
jpa.persist(member);  //회원과 연관관계 함께 저장
```

개발자는 회원과 팀의 관계를 설정하고 회원 객체를 저장하면 된다. JPA는 `team`의 참조를 외래 키로 변환해서 적절한 INSERT SQL을 데이터베이스에 전달한다.

객체를 조회할 때 외래 키를 참조로 변환하는 일도 JPA가 처리해준다.

```
Member member = jpa.find(Member.class, memberId);
Team team = member.getTeam();
```

지금까지 설명한 문제들은 SQL을 직접 다루어도 열심히 코드만 작성하면 어느정도 극복할 수 있는 문제들이었다. 연관관계와 관련해서 극복하기 어려운 패러다임의 불일치 문제를 알아보자.

2.3 객체 그래프 탐색

객체에서 회원이 소속된 팀을 조회할 때는 다음처럼 참조를 사용해서 연관된 팀을 찾으면 되는데, 이것을 객체 그래프 탐색이라 한다.

```
Team team = member.getTeam();
```

객체 연관관계가 다음 그림같이 설계되어 있다고 가정해보자.

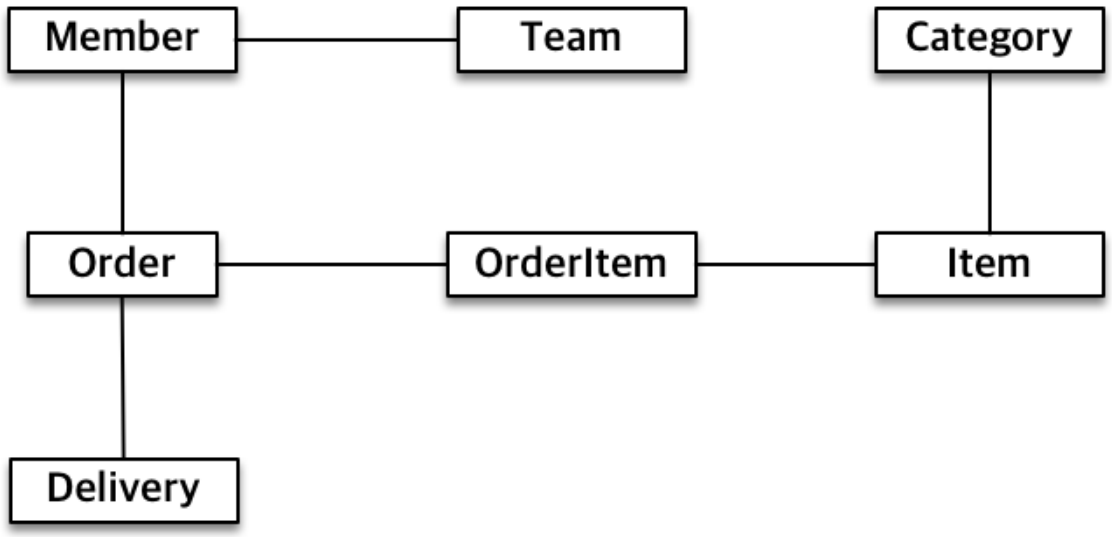

그림 - 객체 연관관계

```
member.getOrder().getOrderItem()... //자유로운 객체 그래프 탐색
```

객체는 마음껏 객체 그래프를 탐색할 수 있어야 한다. 그런데 마음껏 객체 그래프를 탐색할 수 있을까?

```
SELECT M.*, T.*
  FROM MEMBER M
  JOIN TEAM T ON M.TEAM_ID = T.TEAM_ID
```

예를 들어 `MemberDAO`에서 `member` 객체를 조회할 때 이런 SQL을 실행해서 회원과 팀에 대한 데이터만 조회했다면 `member.getTeam()`은 성공하지만 다음처럼 다른 객체 그래프는 데이터가 없으므로 탐색할 수 없다.

```
member.getOrder(); //null
```

SQL을 직접 다루면 처음 실행하는 SQL에 따라 객체 그래프를 어디까지 탐색할 수 있을지가 정해진다. 이것은 객체 지향 개발자에겐 너무 큰 제약이다. 왜냐하면 비즈니스 로직에 따라 사용하는 객체 그래프가 다른데 언제 끊어질지 모를 객체 그래프를 함부로 탐색할 수는 없기 때문이다.

다음 비즈니스 로직을 보자.

```
class MemberService {
    ...
    public void process() {
        Member member = memberDAO.find(memberId);
        member.getTeam(); //member -> team 객체 그래프 탐색이 가능한가?
        member.getOrder().getDelivery(); // ???
    }
}
```

`memberDAO`를 통해서 `member` 객체를 조회했지만, 이 객체와 연관된 `Team`, `Order`, `Delivery` 방향으로 객체 그래프를 탐색할 수 있을지 없을지는 이 코드만 보고는 전혀 예측할 수 없다. 결국, 어디까지 객체 그래프 탐색이 가능한지 알아보려면 데이터 접근 계층인 `DAO`를 열어서 SQL을 직접 확인해야 한다. 이것은 SQL에 의존적인 개발에서도 이야기했듯이 엔티티가 SQL에 논리적으로 종속되어서 발생하는 문제다.

그렇다고 `member`와 연관된 모든 객체 그래프를 데이터베이스에서 조회해서 애플리케이션 메모리에 올려두는 것은 현실성이 없다.

결국 `MemberDAO`에 회원을 조회하는 메서드를 상황에 따라 여러벌 만들어서 사용해야 한다.

```
memberDAO.getMember();  //Member만 조회
memberDAO.getMemberWithTeam();//Member와 Team 조회
memberDAO.getMemberWithOrderWithDelivery();//Member와 Order와 Delivery 조회
...
```

객체 그래프를 신뢰하고 사용할 수 있으면 이런 문제를 어느정도 해소할 수 있다. JPA는 이 문제를 어떻게 해결할까?

2.3.1 JPA와 객체 그래프 탐색

JPA를 사용하면 객체 그래프를 마음껏 탐색할 수 있다.

```
member.getOrder().getOrderItem()... //자유로운 객체 그래프 탐색
```

앞서 SQL과 문제점에서도 언급했듯이 JPA는 연관된 객체를 사용하는 시점에 적절한 SELECT SQL을 실행한다. 따라서 JPA를 사용하면 연관된 객체를 신뢰하고 마음껏 조회할 수 있다. 이 기능은 실제 객체를 사용하는 시점까지 데이터베이스 조회를 미룬다고 해서 **지연 로딩**이라 한다.

이런 기능을 사용하려면 객체에 JPA와 관련된 어떤 코드들을 심어야 하는 것은 아닐까? JPA는 지연 로딩을 투명(transparent)하게 처리한다. 다음 `Member` 객체를 보면 `getOrder()`의 구현 부분에 JPA와 관련된 어떤 코드도 직접 사용하지 않는다.

```
class Member {
    private Order order;

    public Order getOrder() {
        return order;
    }
}
```

```
Member member = jpa.find(Member.class, memberId); //처음 조회 시점에 SELECT MEMBER S

Order order = member.getOrder();
order.getOrderDate(); //Order를 사용하는 시점에 SELECT ORDER SQL
```

`Member`를 사용할 때 마다 `Order`를 함께 사용하면, 이렇게 한 테이블씩 조회하는 것 보다는 `Member`를 조회하는 시점에 SQL 조인을 사용해서 `Member`와 `Order`를 함께 조회하는 것이 효과적이다.

JPA는 연관된 객체를 즉시 함께 조회할지 아니면 실제 사용되는 시점에 지연해서 조회 할지를 간단한 설정으로 정의할 수 있다. 만약 `Member`와 `Order`를 즉시 함께 조회하겠다고 설정하면 JPA는 `Member`를 조회할 때 다음 SQL을 실행해서 연관된 `Order`도 함께 조회한다.

```
SELECT M.*, O.*
  FROM MEMBER M
  JOIN ORDER O ON M.MEMBER_ID = O.MEMBER_ID
```

2.4 비교하기

데이터베이스는 기본 키의 값으로 각 로우(row)를 구분한다. 반면에 객체는 동일성(identity) 비교와 동등성(equality) 비교라는 두가지 비교 방법이 있다.

- 동일성 비교는 `==` 비교다. 객체 인스터스의 주소 값을 비교한다.
- 동등성 비교는 `equals()` 메서드를 사용해서 객체 내부의 값을 비교한다.

따라서 테이블의 로우를 구분하는 방법과 객체를 구분하는 방법에는 차이가 있다.

```
class MemberDAO {

    public Member getMember(String memberId) {
        String sql = "SELECT * FROM MEMBER WHERE MEMBER_ID = ?";
        ...
        //JDBC API, SQL 실행
        return new Member(...);
    }
}
```

```
String memberId = "100";
Member member1 = memberDAO.getMember(memberId);
Member member2 = memberDAO.getMember(memberId);

member1 == member2; //다르다.
```

코드를 보면 기본 키 값이 같은 회원을 객체로 두 번 조회했다. 그런데 둘을 동일성(`==`) 비교하면 `false` 가 반환된다. 왜냐하면 `member1` 과 `member2` 는 같은 데이터베이스 로우에서 조회했지만, 객체 측면에서 볼때 둘은 다른 인스턴스기 때문이다. (`memberDAO.getMember()` 를 호출할 때 마다 `new Member()` 로 인스턴스가 생성된다.)

따라서 데이터베이스의 같은 로우를 조회했지만 객체의 동일성 비교에는 실패한다. 만약 객체를 컬렉션에 보관했다면 동일성 비교에 성공했을 것이다.

```
Member member1 = list.get(0);
Member member2 = list.get(0);

member1 == member2 //같다.
```

이런 패러다임의 불일치 문제를 해결하기 위해 데이터베이스의 같은 로우를 조회할 때 마다 같은 인스턴스를 반환하도록 구현하는 것은 쉽지 않다. 여기에 여러 트랜잭션이 동시에 실행되는 상황까지 고려하면 문제는 더 어려워진다.

2.4.1 JPA와 비교하기

JPA는 같은 트랜잭션일 때 같은 객체가 조회되는 것을 보장한다. 그러므로 다음 코드에서 `member1`과 `member2`는 동일성 비교에 성공한다.

```
String memberId = "100";
Member member1 = jpa.find(Member.class, memberId);
Member member2 = jpa.find(Member.class, memberId);

member1 == member2; //같다.
```

객체 비교하기는 분산 환경이나 트랜잭션이 다른 상황까지 고려하면 더 복잡해진다. 자세한 내용은 책을 진행하면서 차차 알아보자.

2.5 정리

객체 모델과 관계형 데이터베이스 모델은 지향하는 패러다임이 서로 다르다. 문제는 이 패러다임의 차이를 극복하려고 개발자가 너무 많은 시간과 코드를 소비한다는 점이다.

더 어려운 문제는 객체 지향 애플리케이션답게 정교한 객체 모델링을 하면 할수록 패러다임의 불일치 문제가 더 커진다는 점이다. 그리고 이 틈을 메우기 위해 개발자가 소모해야 하는 비용도 점점 더 많아진다. 결국, 객체 모델링은 힘을 잃고 점점 데이터 중심의 모델로 변해간다.

자바 진영에서는 오랜 기간 이 문제에 대한 숙제를 안고 있었고, 패러다임의 불일치 문제를 해결하기 위해 많은 노력을 기울여왔다. 그리고 그 결과물이 바로 JPA다. JPA는 패러다임의 불일치 문제를 해결해주고 정교한 객체 모델링을 유지하게 도와준다.

지금까지 패러다임의 불일치 문제를 설명하면서 JPA를 문제 해결 위주로 간단히 살펴보았다. 이제 본격적으로 JPA에 대해 알아보자.

3. JPA란 무엇인가?

JPA(Java Persistence API)는 자바 진영의 ORM 기술 표준이다.

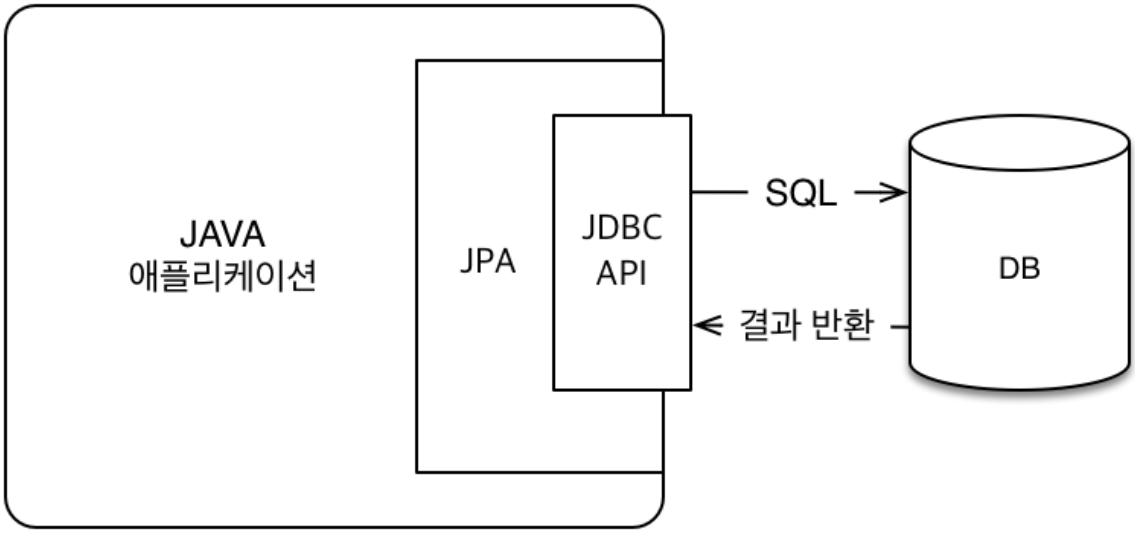

그림 - JPA

ORM이란 무엇인가?

ORM(Object-relational mapping)은 이름 그대로 객체와 관계형 데이터베이스를 매핑한다는 뜻이다. ORM 프레임워크는 객체와 테이블을 매핑해서 패러다임의 불일치 문제를 개발자 대신 해결해준다. 예를 들어 ORM 프레임워크를 사용하면 객체를 데이터베이스에 저장할 때 INSERT SQL을 직접 작성하는 것이 아니라 객체를 마치 자바 컬렉션에 저장하듯이 ORM 프레임워크에 저장하면 된다. 그러면 ORM 프레임워크가 적절한 INSERT SQL을 생성해서 데이터베이스에 객체를 저장해준다.

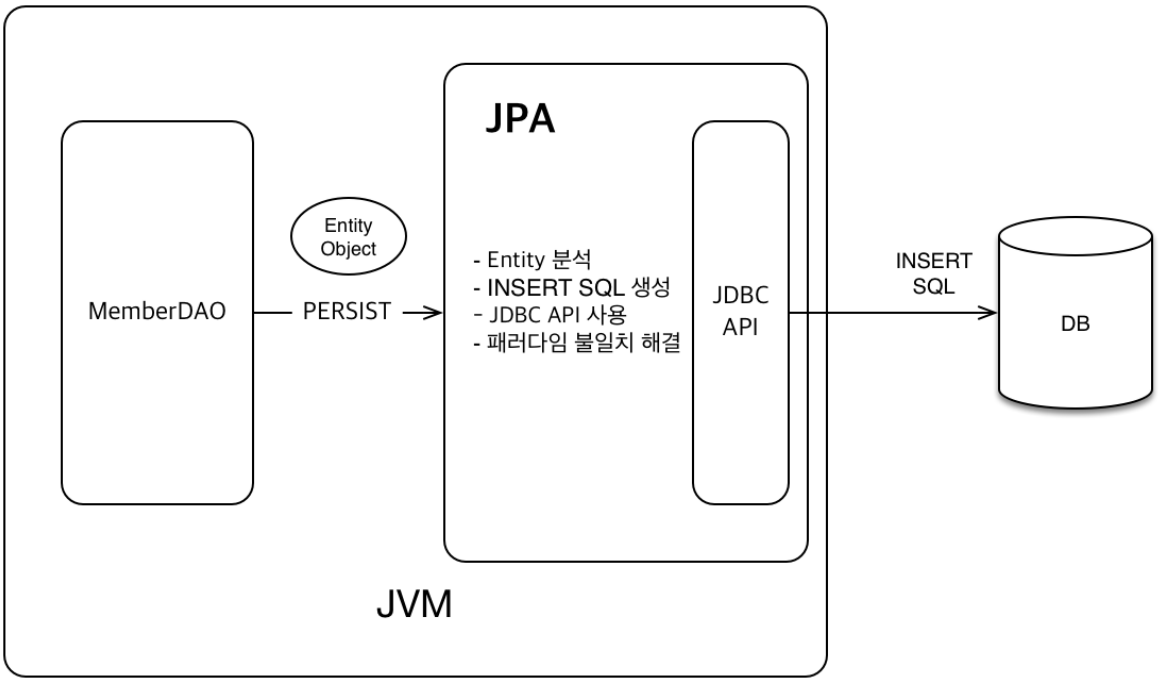

그림 - JPA 저장

```
jpa.persist(member); //저장
```

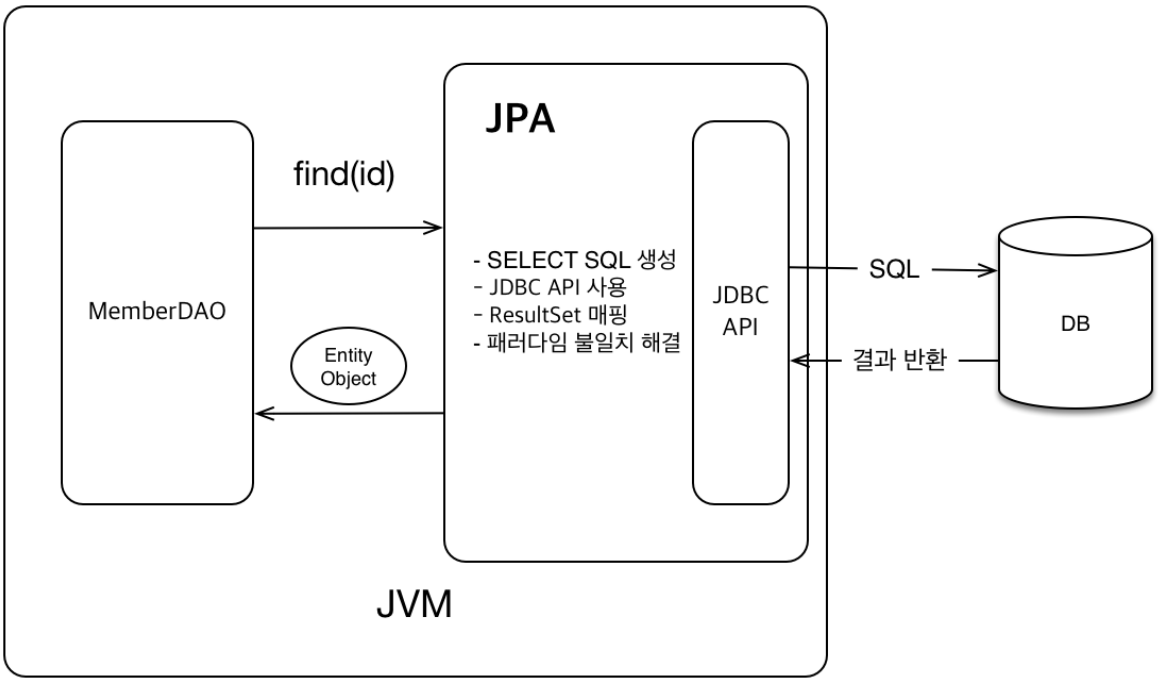

그림 - JPA 조회

```
Member member = jpa.find(memberId); //조회
```

ORM 프레임워크는 단순히 SQL을 개발자 대신 생성해서 데이터베이스에 전달해주는 것뿐만 아니라 앞서 이야기한 다양한 패러다임의 불일치 문제들도 해결해준다. 따라서 객체 측면에서는 정교한 객체 모델링을 할 수 있고 관계형 데이터베이스는 데이터베이스에 맞도록 모델링하면 된다. 그리고 둘을 어떻게 매핑해야 하는지 매핑 방법만 ORM 프레임워크에게 알려주면 된다. 덕분에 개발자는 데이터 중심인 관계형 데이터베이스를 사용해도 객체 지향 애플리케이션 개발에 집중할 수 있다.

어느 정도 성숙한 객체 지향 언어에는 대부분 ORM 프레임워크들이 있는데 각 프레임워크의 성숙도에 따라 단순히 객체 하나를 CRUD 하는 정도의 기능만 제공하는 것부터 패러다임 불일치 문제를 대부분 해결해주는 ORM 프레임워크도 있다.

자바 진영에도 다양한 ORM 프레임워크들이 있는데 그중에 하이버네이트 프레임워크가 가장 많이 사용된다. 하이버네이트는 거의 대부분의 패러다임 불일치 문제를 해결해주는 성숙한 ORM 프레임워크다.

3.1 JPA 소개

과거 자바 진영은 엔터프라이즈 자바 빈즈(EJB)라는 기술 표준을 만들었는데 그 안에는 엔티티 빈이라는 ORM 기술도 포함되어 있었다. 하지만 너무 복잡하고 기술 성숙도도 떨어졌으며 자바 엔터프라이즈(J2EE) 애플리케이션 서버에서만 동작했다.

이때 하이버네이트(Hibernate)[1]라는 오픈소스 ORM 프레임워크가 등장했는데 EJB의 ORM 기술과 비교해서 가볍고 실용적인데다 기술 성숙도도 높았다. 또한 자바 엔터프라이즈 애플리케이션 서버 없이도 동작해서 많은 개발자가 사용하기 시작했다. 결국 EJB 3.0에서 하이버네이트를 기반으로 새로운 자바 ORM 기술 표준이 만들어졌는데 이것이 바로 JPA다.

자바 ORM 기술 표준

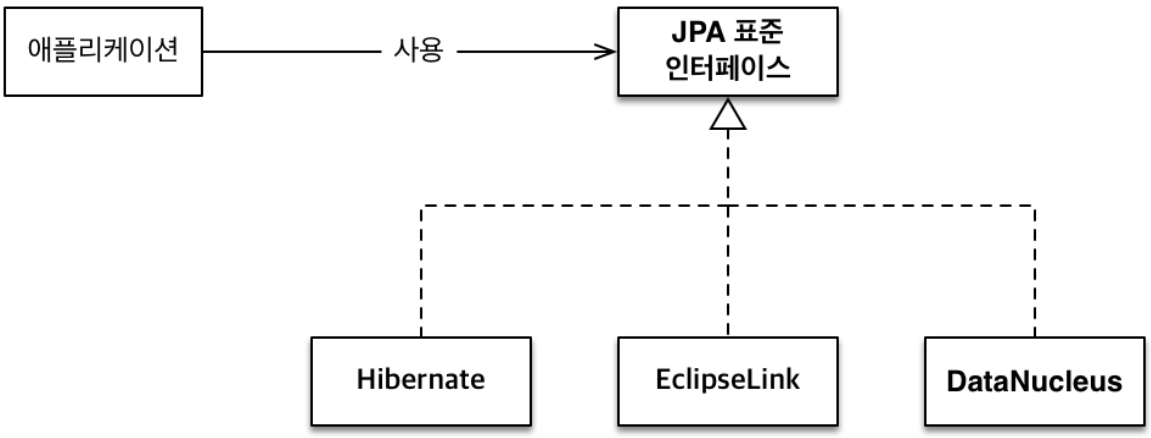

그림 - JPA 표준 인터페이스와 구현체

JPA는 자바 ORM 기술에 대한 API 표준 명세다. 쉽게 이야기해서 인터페이스를 모아둔 것이다. 따라서 JPA를 사용하려면 JPA를 구현한 ORM 프레임워크를 선택해야 한다. 현재 JPA 2.1을 구현한 ORM 프레임워크는 하이버네이트, EclipseLink[2], DataNucleus[3]가 있는데 이 중에 하이버네이트가 가장 대중적이다. 이 책은 **JPA 2.1** 버전에 구현 프레임워크로 **하이버네이트**를 사용한다.

JPA라는 표준 덕분에 특정 구현 기술에 대한 의존도를 줄일 수 있고 다른 구현 기술로 손쉽게 이동할 수 있는 장점이 있다. 그리고 JPA 표준은 일반적이고 공통적인 기능의 모음이다. 따라서 표준을 먼저 이해하고 필요에 따라 JPA 구현체가 제공하는 고유의 기능을 알아가면 된다.

JPA 버전별 특징을 간략하게 정리하면 다음과 같다.

- JPA 1.0(JSR 220) 2006년 : 초기 버전이다. 복합 키와 연관관계 기능이 부족했다.
- JPA 2.0(JSR 317) 2009년 : 대부분의 ORM 기능을 포함하고 JPA Criteria가 추가되었다.
- JPA 2.1(JSR 338) 2013년 : 스토어드 프로시저 접근, 컨버터(Converter), 엔티티 그래프 기능이 추가되었다.

3.2 JPA를 왜 사용해야 하는가?

생산성

JPA를 사용하면 다음 코드처럼 자바 컬렉션에 객체를 저장하듯이 JPA에게 저장할 객체를 전달하면 된다. INSERT SQL을 작성하고 JDBC API를 사용하는 지루하고 반복적인 일은 JPA가 대신 처리해준다.

```
jpa.persist(member); //저장
Member member = jpa.find(memberId); //조회
```

따라서 지루하고 반복적인 코드와 CRUD용 SQL을 개발자가 직접 작성하지 않아도 된다. 더 나아가서 JPA에는 `CREATE TABLE` 같은 DDL 문을 자동으로 생성해주는 기능도 있다. 이런 기능들을 사용하면 데이터베이스 설계 중심의 패러다임을 객체 설계 중심으로 역전 시킬 수 있다.

유지보수

SQL에 의존적인 개발에서도 이야기했듯이 SQL을 직접 다루면 엔티티에 필드를 하나만 추가해도 관련된 등록, 수정, 조회 SQL과 결과를 매핑하기 위한 JDBC API 코드를 모두 변경해야 했다. 반면에 JPA를 사용하면 이런 과정을 JPA가 대신 처리해주므로 필드를 추가하거나 삭제해도 수정해야할 코드가 줄어든다. 따라서 개발자가 작성해야 했던 SQL과 JDBC API 코드를 JPA가 대신 처리해주므로 유지보수 해야하는 코드 수가 줄어든다.

또한, JPA가 패러다임의 불일치 문제를 해결해주므로 객체 지향 언어가 가진 장점들을 활용해서 유연하고 유지보수 하기 좋은 도메인 모델을 편리하게 설계할 수 있다.

패러다임의 불일치 해결

지금까지 패러다임의 불일치 문제가 얼마나 심각한지 다루었고, JPA를 통한 해결책도 간단히 보았다. JPA는 상속, 연관관계, 객체 그래프 탐색, 비교하기와 같은 패러다임의 불일치 문제를 해결해준다.

책 전반에 걸쳐서 JPA가 패러다임의 불일치 문제를 어떻게 해결하는지 자세히 알아보자.

성능

JPA는 애플리케이션과 데이터베이스 사이에서 다양한 성능 최적화 기회를 제공한다.

JPA는 애플리케이션과 데이터베이스 사이에서 동작한다. 이렇게 애플리케이션과 데이터베이스 사이에 계층이 하나 더 있으면 최적화 관점에서 시도해 볼 수 있는 것들이 많다. 다음 코드를 보자.

```
String memberId = "helloId";
Member member1 = jpa.find(memberId);
Member member2 = jpa.find(memberId);
```

이것은 같은 트랜잭션 안에서 같은 회원을 두 번 조회하는 코드의 일부분이다. JDBC API를 사용해서 해당 코드를 직접 작성했다면 회원을 조회할 때 마다 SELECT SQL을 사용해서 데이터베이스와 두 번 통신했을 것이다. JPA를 사용하면 회원을 조회하는 SELECT SQL을 한 번만 데이터베이스에 전달하고 두 번째는 조회한 회원 객체를 재사용한다.

참고로 하이버네이트는 SQL 힌트를 넣을 수 있는 기능도 제공한다.

데이터 접근 추상화와 벤더 독립성

관계형 데이터베이스는 같은 기능도 벤더마다 사용법이 다른 경우가 많다. 단적인 예로 페이징 처리는 데이터베이스마다 달라서 사용법을 각각 배워야 한다. 결국, 애플리케이션은 처음 선택한 데이터베이스 기술에 종속되고 다른 데이터베이스로 변경하기는 매우 어렵다.

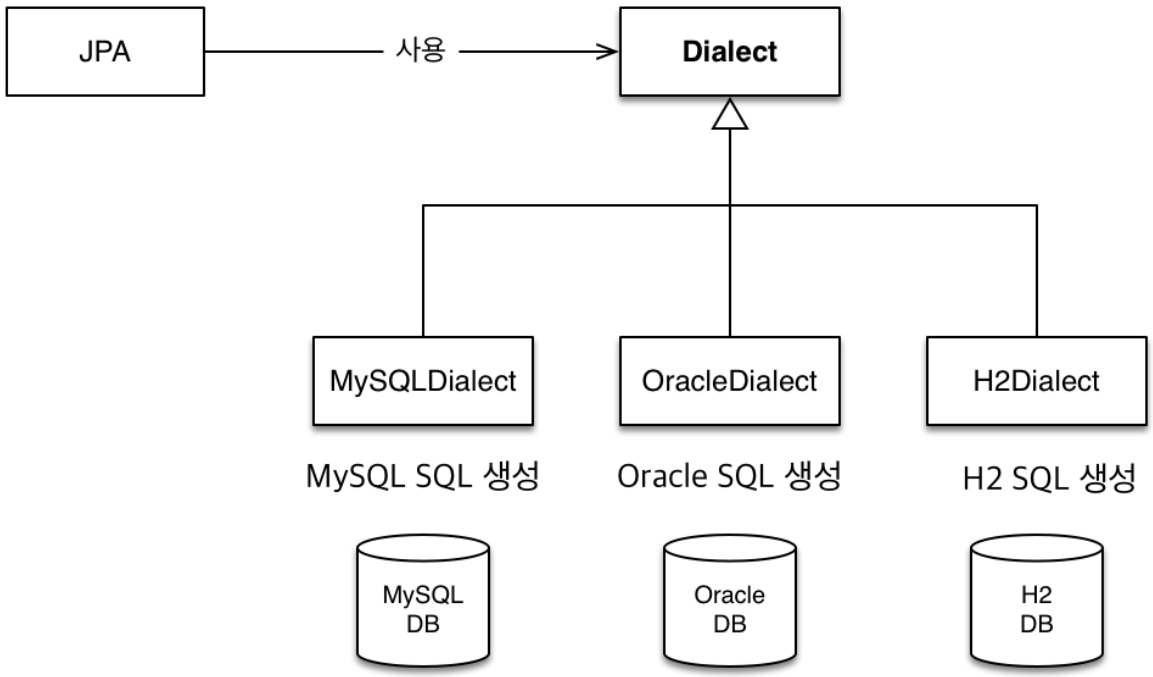

그림 - 벤더 독립성

JPA는 애플리케이션과 데이터베이스 사이에 추상화된 데이터 접근 계층을 제공해서 애플리케이션이 특정 데이터베이스 기술에 종속되지 않도록 한다. 만약 데이터베이스를 변경하면 JPA에게 다른 데이터베이스를 사용한다고 알려주기만 하면 된다. 예를 들어 JPA를 사용하면 로컬 개발 환경은 H2 데이터베이스를 사용하고 개발이나 상용 환경은 Oracle이나 MySQL 데이터베이스를 사용할 수 있다.

표준

JPA는 자바 진영의 ORM 기술 표준이다. 앞서 이야기했듯이 표준을 사용하면 다른 구현기술로 손쉽게 변경할 수 있다.

정리

지금까지 JPA를 왜 사용해야 하는지에 대해 설명했다. JPA가 이러한 기능들을 어떻게 지원하는지에 대한 자세한 내용은 책 전반에 걸쳐서 차근차근 살펴보기로 하고, 우선은 다음 장에서 테이블 하나를 등록/수정/삭제/조회하는 간단한 JPA 애플리케이션을 만들어보자.

ORM 궁금증과 오해 (참고)

> **ORM 궁금증과 오해**
>
> 관계형 데이터베이스에는 익숙하지만, 아직 ORM을 접해보지 못한 개발자들이 자주하는 질문과 답을 Q&A 형식으로 정리해보았다.

Q : **ORM 프레임워크를 사용하면 SQL과 데이터베이스는 잘 몰라도 되나요?**
A : 아닙니다. ORM 프레임워크가 애플리케이션을 객체 지향적으로 개발할 수 있도록 도와주긴 하지만 데이터는 결국 관계형 데이터베이스에 저장됩니다. 테이블 설계는 여전히 중요하고 SQL도 잘 알아야 합니다. 그리고 ORM 프레임워크를 사용할 때 가장 중요한 일은 객체와 테이블을 매핑하는 것입니다. 매핑을 올바르게 하려면 객체와 관계형 데이터베이스 양쪽을 모두 이해해야 합니다. 따라서 데이터베이스 테이블 설계나 SQL을 잘 몰라서 ORM 프레임워크를 사용한다는 것은 ORM의 본질을 잘못 이해한 것입니다.

Q : **성능이 느리진 않나요?**
A : 지금 시대에 JAVA가 느리다고 말하는 것과 비슷하다고 생각합니다. JPA는 다양한 성능 최적화 기능을 제공해서 잘 이해하고 사용하면 SQL을 직접 사용할 때보다 더 좋은 성능을 낼 수도 있습니다. 또한 JPA의 네이티브 SQL 기능을 사용해서 SQL을 직접 호출하는 것도 가능합니다. 하지만 JPA를 잘 이해하지 못하고 사용하면 N+1 같은 문제로 인해 심각한 성능 저하가 발생할 수 있습니다. 여기서 말하는 N+1 문제는 예를 들어 SQL 한 번으로 회원 100명을 조회했는데 각 회원마다 주문한 상품을 추가로 조회하기 위해 100번의 SQL을 추가로 실행하는 것을 말합니다. 한 번 SQL을 실행해서 조회한 수만큼 N 번 SQL을 추가로 실행한다고 해서 N+1 문제라 합니다. 이런 문제도 JPA를 약간만 공부하면 어렵지 않게 해결할 수 있습니다.

Q : **통계 쿼리처럼 매우 복잡한 SQL은 어떻게 하나요?**
A : JPA는 통계 쿼리 같이 복잡한 쿼리보다는 실시간 처리용 쿼리에 더 최적화되어 있습니다. 상황에 따라 다르지만 정말 복잡한 통계 쿼리는 SQL을 직접 작성하는 것이 더 쉬운 경우가 많습니다. 따라서 JPA가 제공하는 네이티브 SQL을 사용하거나 MyBatis나 스프링의 JdbcTemplate 같은 SQL Mapper 형태의 프레임워크를 혼용하는 것도 좋은 방법입니다.

Q : **MyBatis와 어떤 차이가 있나요?**
A : MyBatis나 스프링 JdbcTemplate을 보통 SQL Mapper라 합니다. 이것은 이름 그대로 객체와 SQL을 매핑합니다. 따라서 SQL과 매핑할 객체만 지정하면 지루하게 반복되는 JDBC API 사용과 응답 결과를 객체로 매핑하는 일은 SQL Mapper가 대신 처리해줍니다. 이런 SQL Mapper가 편리하긴 하지만 결국 개발자가 SQL을 직접 작성해야 하므로 SQL에 의존하는 개발을 피할 수 없습니다. 반면에 ORM은 객체와 테이블을 매핑만하면 ORM 프레임워크가 SQL을 만들어서 데이터베이스와 관련된 처리를 해주므로 SQL에 의존하는 개발을 피할 수 있습니다.

Q : **하이버네이트 프레임워크를 신뢰할 수 있나요?**
A : 하이버네이트는 2001년에 공개된 후 지금도 발전하고 있는 성숙한 ORM 프레임워크입니다. 핵심 모듈들의 테스트 케이스만 4000개가 넘고 코어 코드는 10만 라인에 육박합니

다. 글로벌에서 수 만개 이상의 자바 프로젝트에서 사용중이고 매일 3천번 이상의 내려받기가 일어납니다.[4] 그리고 Atlassian, AT&T, Cisco 같은 업체들[5]이 사용합니다.

Q : **제 주위에는 MyBatis(iBatis, myBatis)만 사용하는데요?**
A : 국내에서 유독 MyBatis를 많이 사용하는 것은 사실입니다. 하지만 전 세계를 대상으로 조사하면 하이버네이트 ORM 프레임워크를 사용하는 비중이 절대적으로 많습니다.

다음 표는 2014년에 ZeroTurnaround라는 회사에서 전 세계 자바 개발자를 대상으로 한 설문 조사 결과입니다.[6]

사용 프레임워크	투표율
하이버네이트	67.5%
순수 JDBC	22%
SpringJdbcTemplate	19.5%
EclipseLink	13%
MyBatis	6.5%
JOOQ	1.5%
Other	7%

구글 트랜드 검색을 사용해서 하이버네이트와 MyBatis를 비교해도 비슷한 결과가 나옵니다.

2014년 6월 전 세계를 대상으로 구글 트랜드 검색

- JPA + Hibernate : 94%
- iBatis + myBatis: 6%

Q : **학습곡선이 높다고 하던데요?**
A : 네 JPA는 학습곡선이 높은 편입니다. JPA를 사용하려면 객체와 관계형 데이터베이스를 어떻게 매핑하는지 학습한 후에 JPA의 핵심 개념들을 이해해야 합니다. 기초 없이 단순히 인터넷에 돌아다니는 예제들을 복사해서 사용하기만 하면 금방 한계에 부딪힙니다. 실무에서는 수많은 테이블과 객체를 매핑해야 하므로 매핑하는 방법을 정확히 이해해야 합니다. 또한, JPA의 핵심 개념인 영속성 컨텍스트에 대한 이해가 부족하면 SQL을 직접 사용해서 개

> 발하는 것보다 못한 상황이 벌어질 수 있습니다. **사실 JPA가 어려운 근본적인 이유는 ORM이 객체 지향과 관계형 데이터베이스라는 두 기둥 위에 있기 때문입니다.** 이 둘의 기초가 부족하면 어려울 수밖에 없습니다.

---

1. http://hibernate.org ↩
2. http://www.eclipse.org/eclipselink ↩
3. http://www.datanucleus.org/ ↩
4. https://community.jboss.org/wiki/HibernateFAQ-ProductEvaluationFAQ ↩
5. 하이버네이트를 사용하는 해외 주요 업체
   https://community.jboss.org/wiki/WhoUsesHibernate ↩
6. 중복 투표 가능, http://www.slideshare.net/ZeroTurnaround/java-tools-and-technologies-landscape-for–2014-image-gallery ↩