# ORM(JPA)의 필요성 - 1

> ORM, JPA 둘 다 자바에서 데이터베이스를 다뤄 본 분들은 한 번쯤은 들어봤거나, 사용해 보셨을 거라고 생각합니다. 그러나 해당 기술이 어떤 장점이 있냐고 물어봤을 때 이에 답하기는 쉽지 않습니다. 이번 시간에는 ORM(JPA)의 필요성에 대해 알아봅시다.

## 개념

### ORM

- 객체와 관계형 데이터베이스 간의 간극을 좁혀주는 기술입니다.

### JPA

- JPA는 자바 진형에서 고안한 ORM 기술 표준입니다.
- 스프링 진영에서도 스프링 데이터 JPA라는 기술로 JPA를 지원합니다.

> 객체와 관계형 데이터베이스 간의 간극이란 뭘 말하는 것일까요? 먼저 순수 SQL을 직접 다룰 때 발생하는 문제점을 살펴봅시다

## SQL을 직접 다룰 때 발생하는 문제점

- RDBMS(관계형 데이터베이스)
  - 가장 대중적이고 신뢰성 높은 데이터 저장소

> 자바로 개발하는 애플리케이션은 RDBMS를 데이터 저장소로 사용하는 경우가 많습니다.

데이터베이스의 데이터를 관리하기 위해서는 SQL문을 사용해야 하는데, 자바로 작성한 애플리케이션은 이를 위해 JDBC API를 사용해서 SQL문을 데이터베이스에 전달합니다.

### 번거로운 변환 작업

첫 번째 문제점은 변환 작업 과정이 번거롭다는 것입니다.

#### 회원 관리 기능 CRUD

Java에서 SQL을 직접 다루는 경우를 먼저 살펴봅시다. 이를 위해 회원 관리 예제를 생각해보죠. DAO는 Data Access Object의 약자로, 말 그대로 데이터 접근 객체임을 의미합니다.

> Member.class

```
public class Member {
    private Long memberId;
    private String name;
}
```

> MemberDAO.class

```
public class MemberDAO {
    public Member find(Long memberId) {
        // ...
    }
    public Member save(Member member) {
        // ...
    }
}
```

**find()**

1. 회원 조회 SQL

```
String sql = "SELECT member_id, name
            FROM member m
            WHERE member_id = ?"
```

2. JDBC API

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

3. 조회 결과 매핑

```
String memberId = rs.getLong("member_id");
String name = rs.getString("name")

Member member = new Member(memberId, name)
```

**save()**

1. 회원 등록 SQL

```
String sql = "INSERT INTO member(member_id, name) VALUES (?, ?)"
```

2. JDBC API

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

pstmt.execute(sql);
```

단순 회원 조회 및 등록인데도 여러 단계를 거쳐야 한다는 것을 알 수 있습니다. 이처럼 데이터 접근 계층을 설계하는 것은 매우 번거롭습니다.

우리가 흔히 Collection이나 배열 형태로 객체를 다룰 때와는 느낌이 많이 다릅니다. 예를 들어 List에서 요소를 추가하기 위해서는 단순히 다음과 같이 한 줄로 처리할 수 있습니다.

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

데이터베이스 연산을 자바로 수행할 때는 왜 이렇게 복잡하게 진행해야 할까요?

데이터베이스는 객체 구조와는 다른 데이터 중심의 구조를 가지기 때문에 객체를 그대로 데이터베이스에 저장하거나 조회하는 데 사용할 수 없습니다.

## SQL에 의존적인 개발

**테이블 컬럼 변경**

위의 예시에서, 만약 Member 테이블에 전화번호 컬럼이 추가된다면 어떻게 될까요?

저장 시 사용하는 INSERT 문, 조회 시 사용하는 SELECT 문에 대한 수정 뿐만 아니라 이를 다시 매핑하는 자바 코드까지 전부 수정해야 합니다. 즉, SQL에 의존적인 개발은 유지보수가 힘들다는 문제점이 있습니다.

**연관된 객체 관리**

```
class Member {
    private Long memberId;
    private String name;
    private String tel;
    private Team team; // 추가
}

class Team {
    private Long teamId;
    private String teamName;
}
```

만약 '팀'이라는 객체가 존재하고, 각 회원 객체가 특정한 팀에 소속되어 있다고 할 때, DB에서 특정 회원의 팀을 조회하려면 어떻게 해야 할까요?

이를 위해선 위의 find()함수만으로는 구현이 불가능합니다. 왜냐하면, find()함수에서의 sql을 보면 team에 관한 정보를 가져오는 내용이 없기 때문이죠.

이를 위해서는 다음과 같은 team에 대한 JOIN 절이 포함된 별도의 SQL문이 필요합니다.

```
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
```

연관된 객체가 정상적으로 조회되는지 알기 위해서는 사용하는 SQL문을 확인해야 합니다.

> 즉, DAO를 통해 애플리케이션 데이터 접근을 추상화하더라도 결국 DAO 로직 내부에 존재하는 SQL문을 확인해야 정확한 동작을 알 수 있습니다.

## 요약

SQL문을 직접 다뤄서 개발하는 경우, 요약하면 다음과 같은 문제가 발생합니다.

- 완전한 계층 분리가 어렵다
- SQL에 의존적이고 유지보수가 어렵다.

다음 시간에는 이런 일이 발생하게 되는 원인을 알기 위해 조금 더 구체적으로 객체와 RDBMS의 패러다임 불일치 문제에 대해 알아봅시다