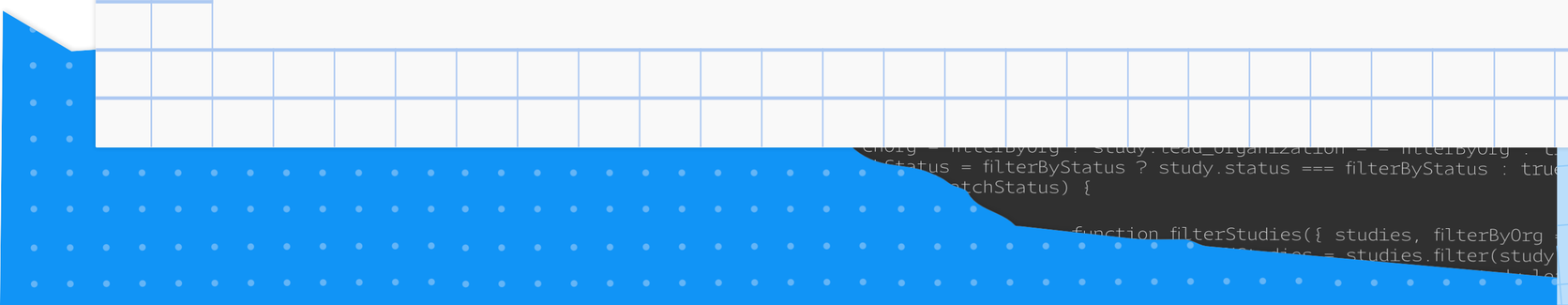
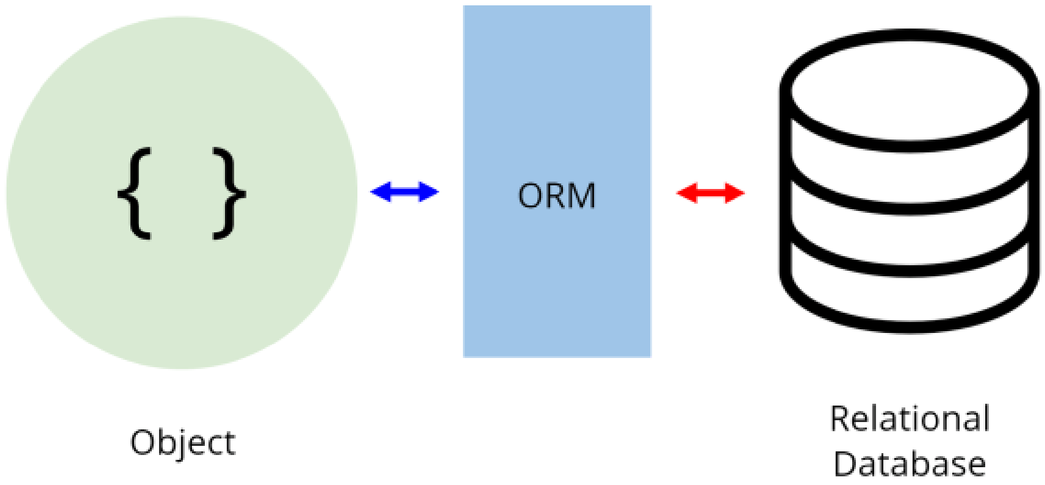
Google Developer Student Clubs

Backend 5th Study



Contents

ORM



Object-relational mapping(객체 관계 매핑)
객체는 객체대로 설계
관계형 데이터베이스는 관계형 데이터베이스대로 설계
ORM 프레임워크가 중간에서 매핑
대중적인 언어에는 대부분 ORM 기술이 존재

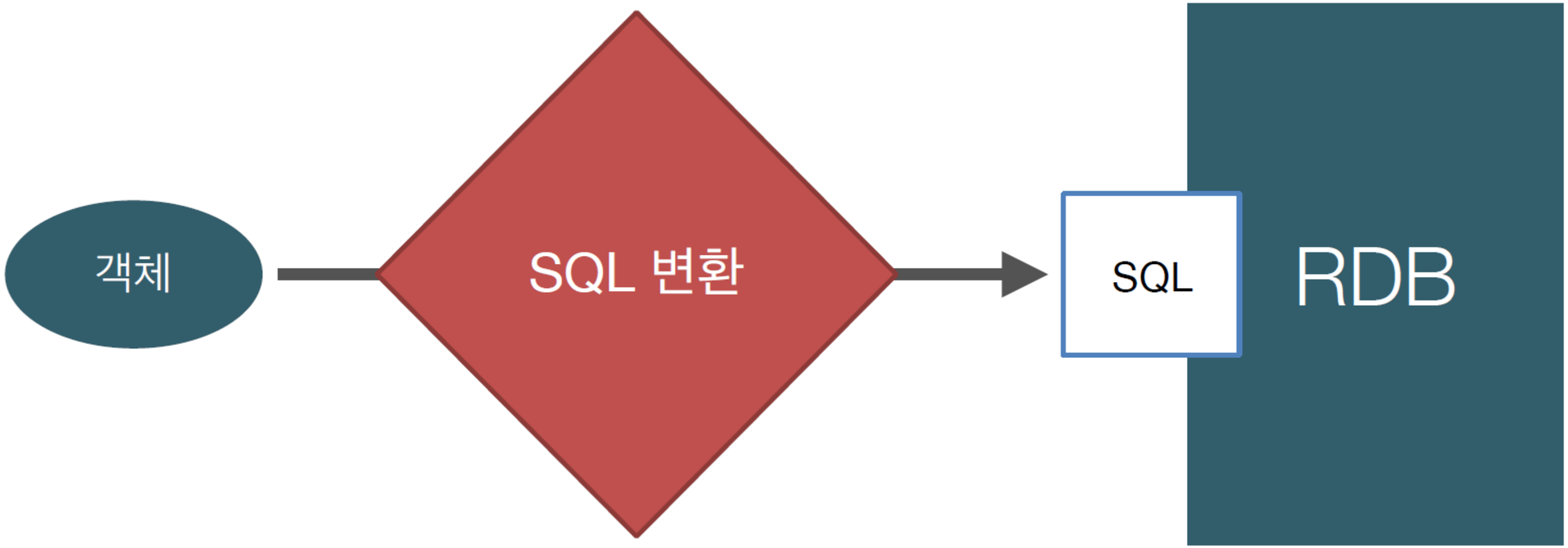
ORM

기본적으로 SQL에 의존적인 개발을 피하기 어렵다.

ORM

객체 지향 프로그래밍은 **추상화**, **캡슐화**, **정보은닉**, **상속**, **다형성** 등 시스템의 복잡성을 제어할 수 있는 다양한 장치들을 제공한다.

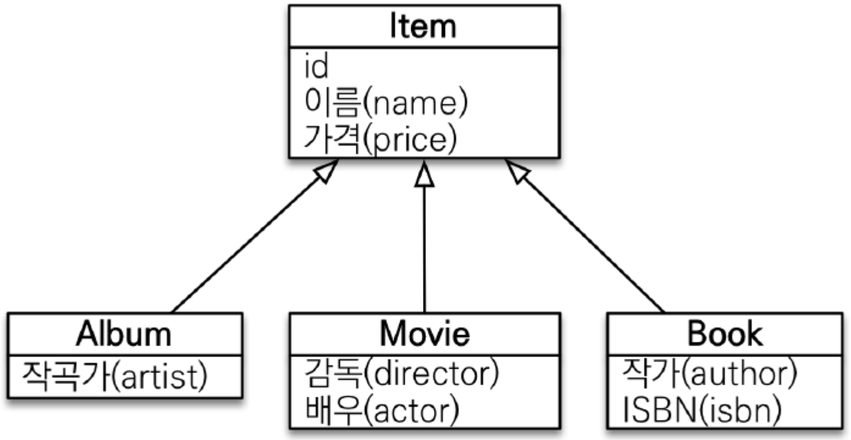
ORM



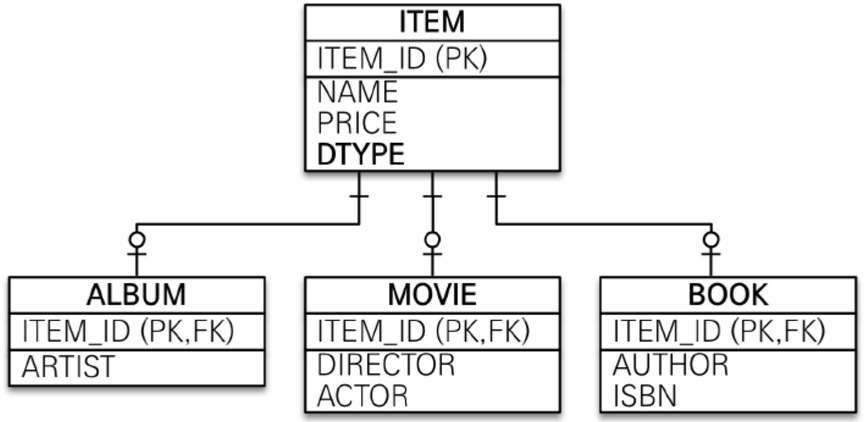
ORM

사실상 개발자 == SQL 매퍼

상속?



[객체 상속 관계]



[Table 슈퍼타입 서브타입 관계]

ORM

1. 객체 분해
2. INSERT INTO ITEM ...
3. INSERT INTO ALBUM …
4. 각각의 테이블에 따른 조인 SQL 작성...
5. 각각의 객체 생성...
6. 상상만 해도 복잡
7. 더 이상의 설명은 생략한다.
8. 그래서 **DB에 저장할 객체에는 상속 관계 안쓴다.**

자바를 활용한다면?

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

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

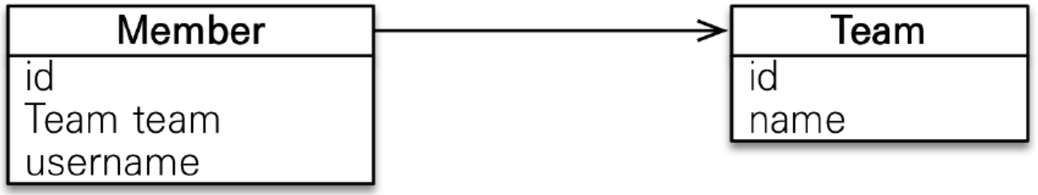
부모 타입으로 조회 후 다형성 활용

```
Item item = list.get(albumId);
```

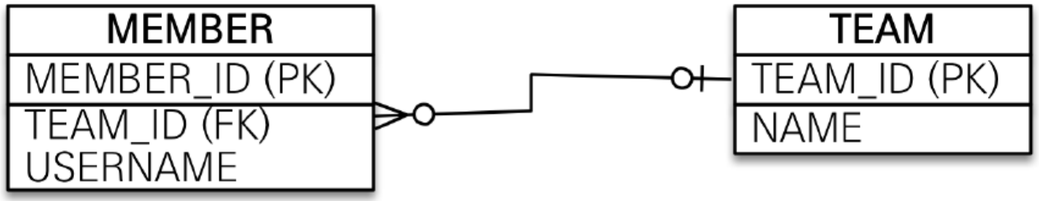
연관 관계

- 객체는 **참조**를 사용: member.getTeam()
- 테이블은 **외래 키**를 사용: JOIN ON M.TEAM_ID = T.TEAM_ID

[객체 연관관계]



[테이블 연관관계]



ORM

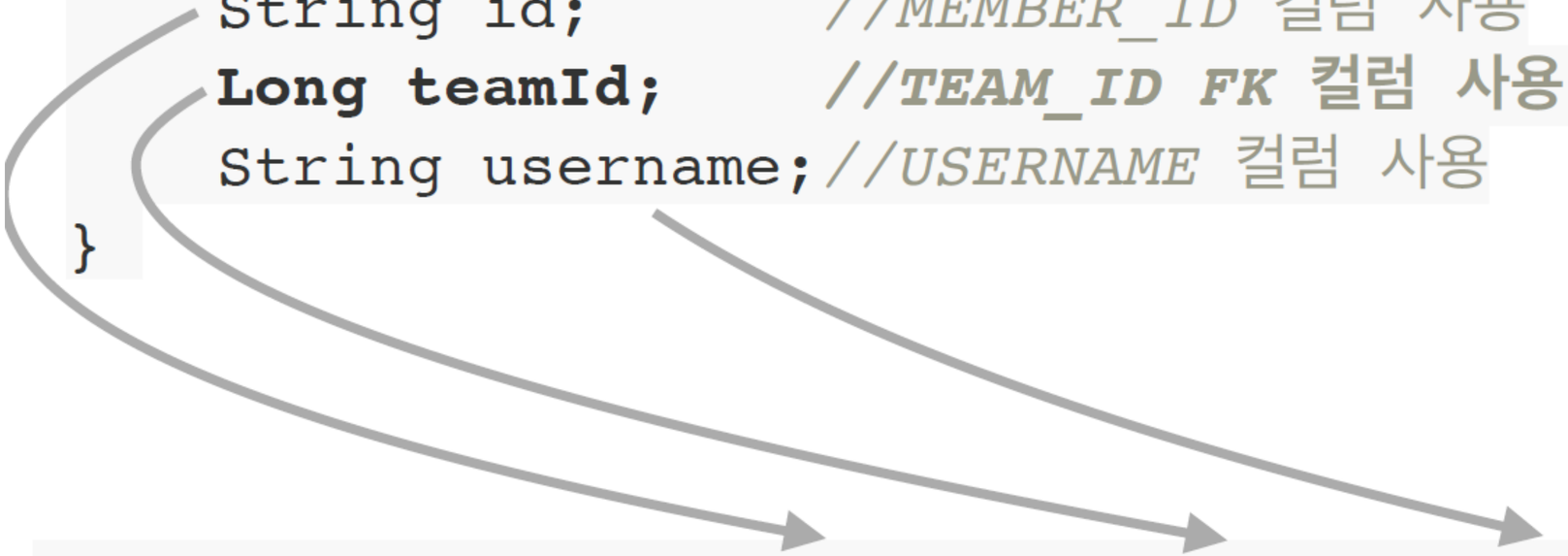
객체에 맞추어 테이블 모델링

```
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
}
```

ORM

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



```
INSERT INTO MEMBER(MEMBER_ID, TEAM_ID, USERNAME) VALUES …
```

ORM

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

```
member.getTeam().getId();
```

```
INSERT INTO MEMBER(MEMBER_ID, TEAM_ID, USERNAME) VALUES …
```

ORM

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

```
public Member find(String memberId) {
    //SQL 실행 ...
    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;
}
```

ORM

상황에 따라 동일한 회원 조회 메서드를 여러벌 생성

```
memberDAO.getMember();   //Member만 조회

memberDAO.getMemberWithTeam();//Member와 Team 조회

//Member,Order,Delivery
memberDAO.getMemberWithOrderWithDelivery();
```

ORM

객체답게 모델링 할수록 매핑 작업만 늘어난다.
객체를 자바 컬렉션에 저장 하듯이 DB에 저장할 수는 없을까?

Google Developer Student Clubs

JPA

JPA

Java Persistence API

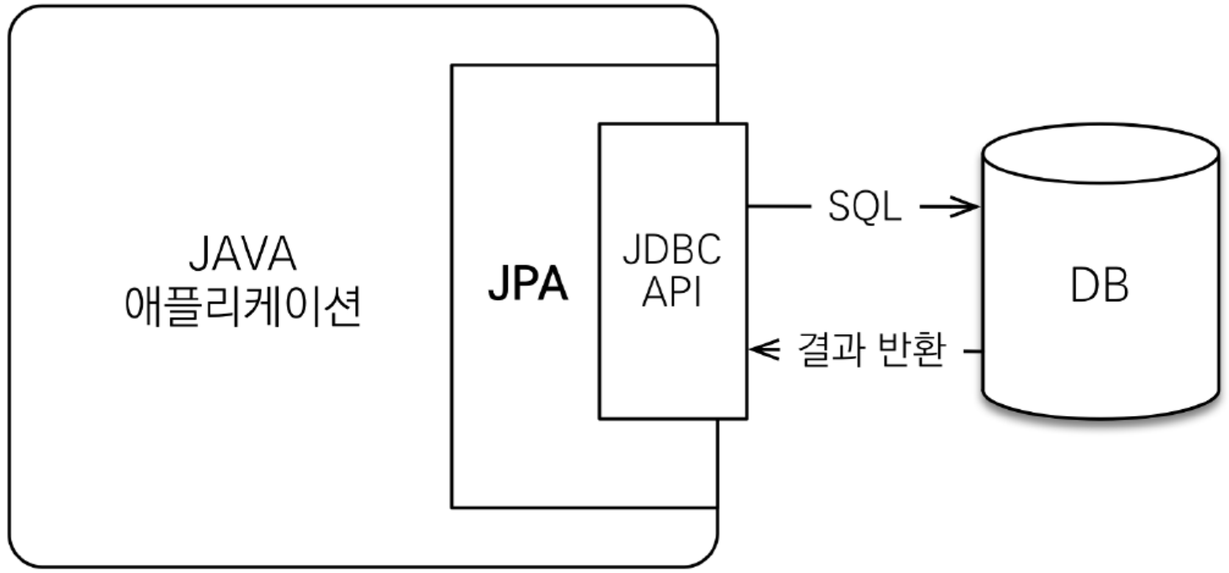
자바 진영의 ORM 기술 표준

JPA

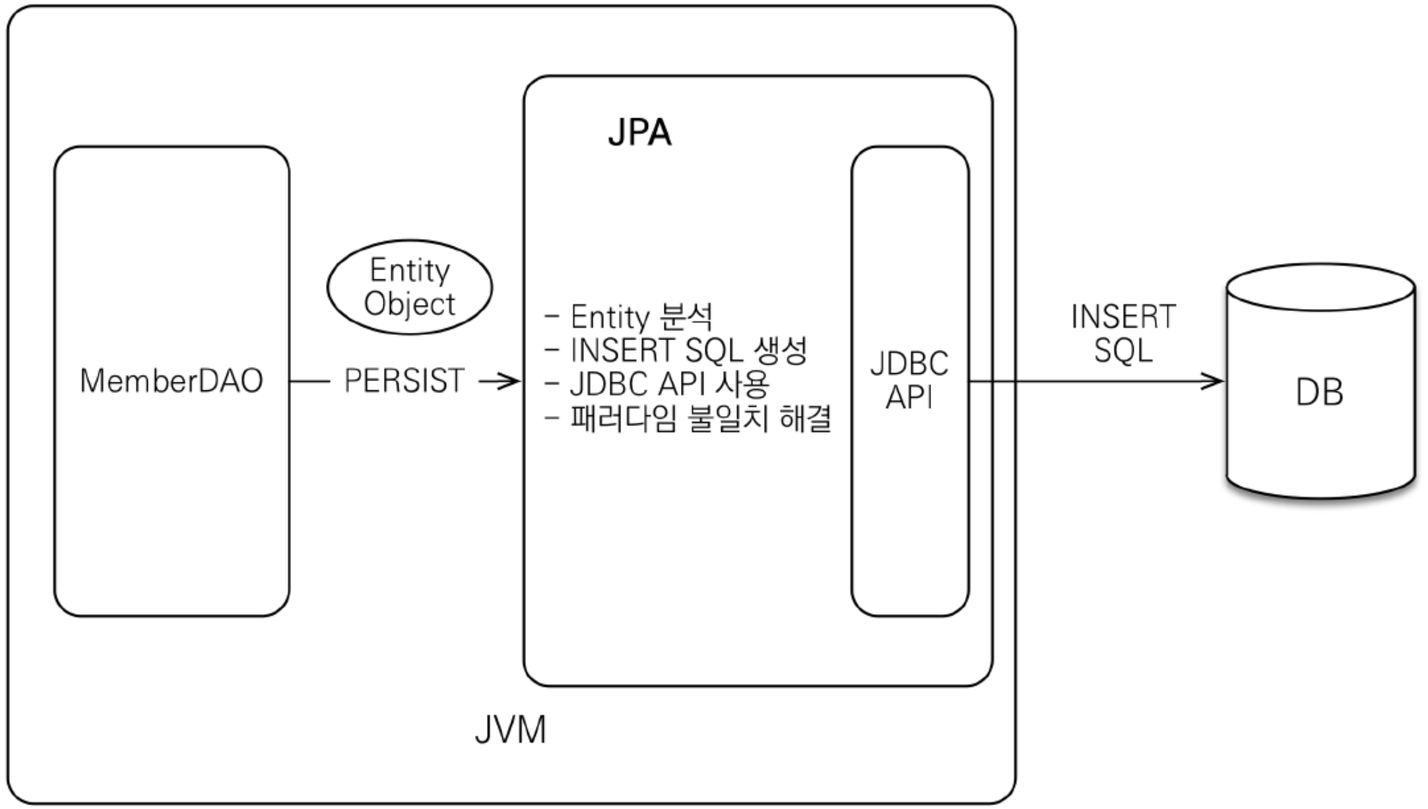
Java Persistence API

자바 진영의 ORM 기술 표준

JPA



JPA


JPA
MemberDAO
Entity
Object
PERSIST
- Entity 분석
- INSERT SQL 생성
- JDBC API 사용
- 패러다임 불일치 해결
JDBC
API
INSERT
SQL
DB
JVM

Why JPA?

SQL 중심적인 개발에서 객체 중심으로 개발

생산성

유지보수

패러다임의 불일치 해결

성능

표준

JPA

- 저장: **jpa.persist**(member)
- 조회: Member member = **jpa.find**(memberId)
- 수정: **member.setName**(“변경할 이름”)
- 삭제: **jpa.remove**(member)

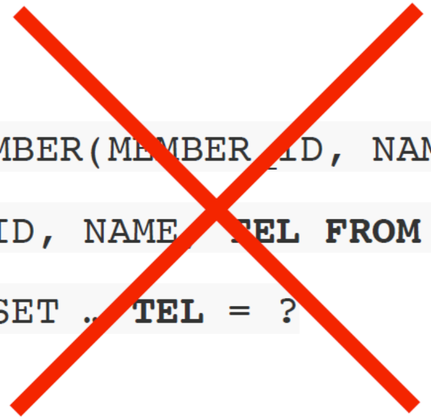
유지보수

```java
public class Member {
  private String memberId;
  private String name;
  private String tel;
   ...
}
```

```
INSERT INTO MEMBER(MEMBER_ID, NAME, TEL) VALUES

SELECT MEMBER_ID, NAME, TEL FROM MEMBER M

UPDATE MEMBER SET … TEL = ?
```



필드가 추가 되면 그냥 객체에 쓰기만 하면 됨

JPA

개발자가 할일

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

나머진 JPA가 처리

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

Google Developer Student Clubs

쉬는 시간

성능 개선

1. 1차 캐시와 동일성(identity) 보장
2. 트랜잭션을 지원하는 쓰기 지연(transactional write-behind)
3. 지연 로딩(Lazy Loading)

1차 캐시와 동일성 보장

1. 같은 트랜잭션 안에서는 같은 엔티티를 반환 - 약간의 조회 성능 향상
2. DB Isolation Level이 Read Commit이어도 애플리케이션에서 Repeatable Read 보장

```
String memberId = "100";
Member m1 = jpa.find(Member.class, memberId); //SQL
Member m2 = jpa.find(Member.class, memberId); //캐시

println(m1 == m2) //true
```

SQL 1번만 실행

트랜잭션을 지원하는 쓰기 지연 - INSERT

1. 트랜잭션을 커밋할 때까지 INSERT SQL을 모음
2. JDBC BATCH SQL 기능을 사용해서 한번에 SQL 전송

```
transaction.begin();  // [트랜잭션] 시작

em.persist(memberA);
em.persist(memberB);
em.persist(memberC);
//여기까지 INSERT SQL을 데이터베이스에 보내지 않는다.

//커밋하는 순간 데이터베이스에 INSERT SQL을 모아서 보낸다.
transaction.commit(); // [트랜잭션] 커밋
```

트랜잭션을 지원하는 쓰기 지연 - UPDATE

1. UPDATE, DELETE로 인한 로우(ROW)락 시간 최소화
2. 트랜잭션 커밋 시 UPDATE, DELETE SQL 실행하고, 바로 커밋

```
transaction.begin();  // [트랜잭션] 시작

changeMember(memberA);
deleteMember(memberB);
비즈니스_로직_수행();      //비즈니스 로직 수행 동안 DB 로우 락이 걸리지 않는다.

//커밋하는 순간 데이터베이스에 UPDATE, DELETE SQL을 보낸다.
transaction.commit(); // [트랜잭션] 커밋
```

지연 로딩과 즉시 로딩

- 지연 로딩: 객체가 실제 사용될 때 로딩
- 즉시 로딩: JOIN SQL로 한번에 연관된 객체까지 미리 조회

지연 로딩

```
Member member = memberDAO.find(memberId);
Team team = member.getTeam();
String teamName = team.getName();
```

```
SELECT * FROM MEMBER

SELECT * FROM TEAM
```

즉시 로딩

```
Member member = memberDAO.find(memberId);
Team team = member.getTeam();
String teamName = team.getName();
```

```
SELECT M.*, T.*
FROM MEMBER
JOIN TEAM …
```

Google Developer Student Clubs

끝!
사실상 진짜 내용은 다음시간