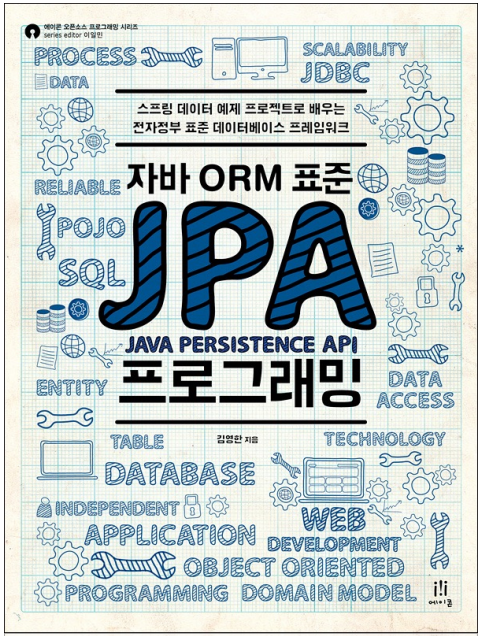
# JPA와 모던 자바 데이터 저장 기술

김영한

# 김영한

SI, J2EE 강사, DAUM,
SK 플래닛, 우아한형제들

저서: 자바 ORM 표준
JPA 프로그래밍



# 목차

- SQL 중심적인 개발의 문제점
- JPA 소개

애플리케이션

**객체 지향 언어** - [Java,Scala, ...]


데이터베이스 세계의 헤게모니
관계형 DB - [Oracle, MySQL, ...]

지금 시대는 **객체**를

**관계형 DB**에 관리

SQL! SQL!! SQL!!!

# SQL 중심적인 개발의 문제점

# 무한 반복, 지루한 코드

## CRUD

INSERT INTO …

UPDATE …

SELECT …

DELETE …

자바 객체를  SQL로 ...

SQL을 자바 객체로 ...

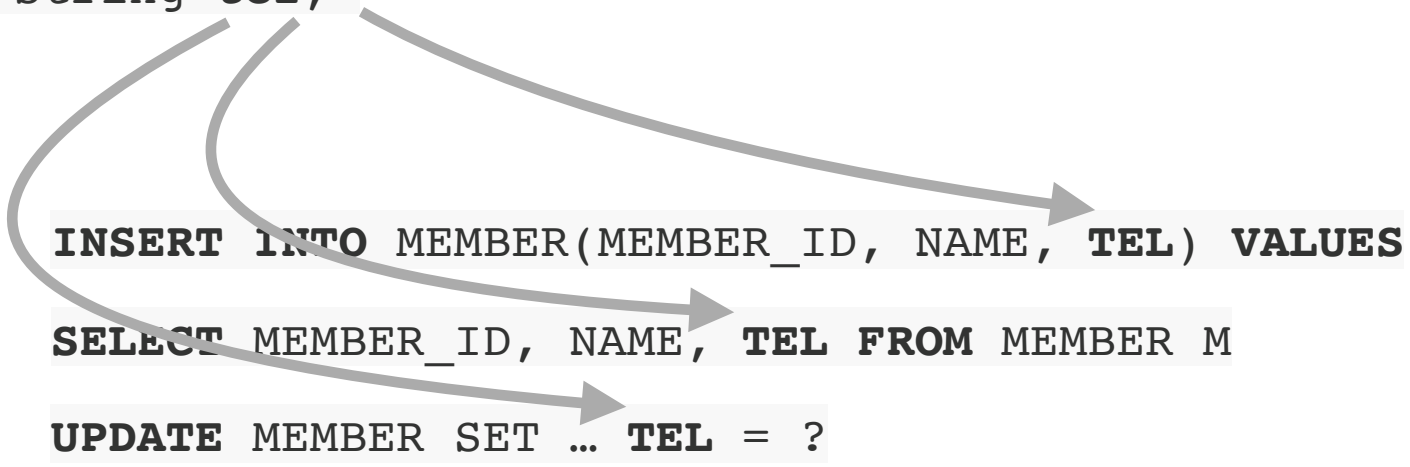
# 객체 CURD

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

  ...
}
```

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

SELECT MEMBER_ID, NAME FROM MEMBER M

UPDATE MEMBER SET …
```

# 객체 CURD - 필드 추가

```
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 = ?
```

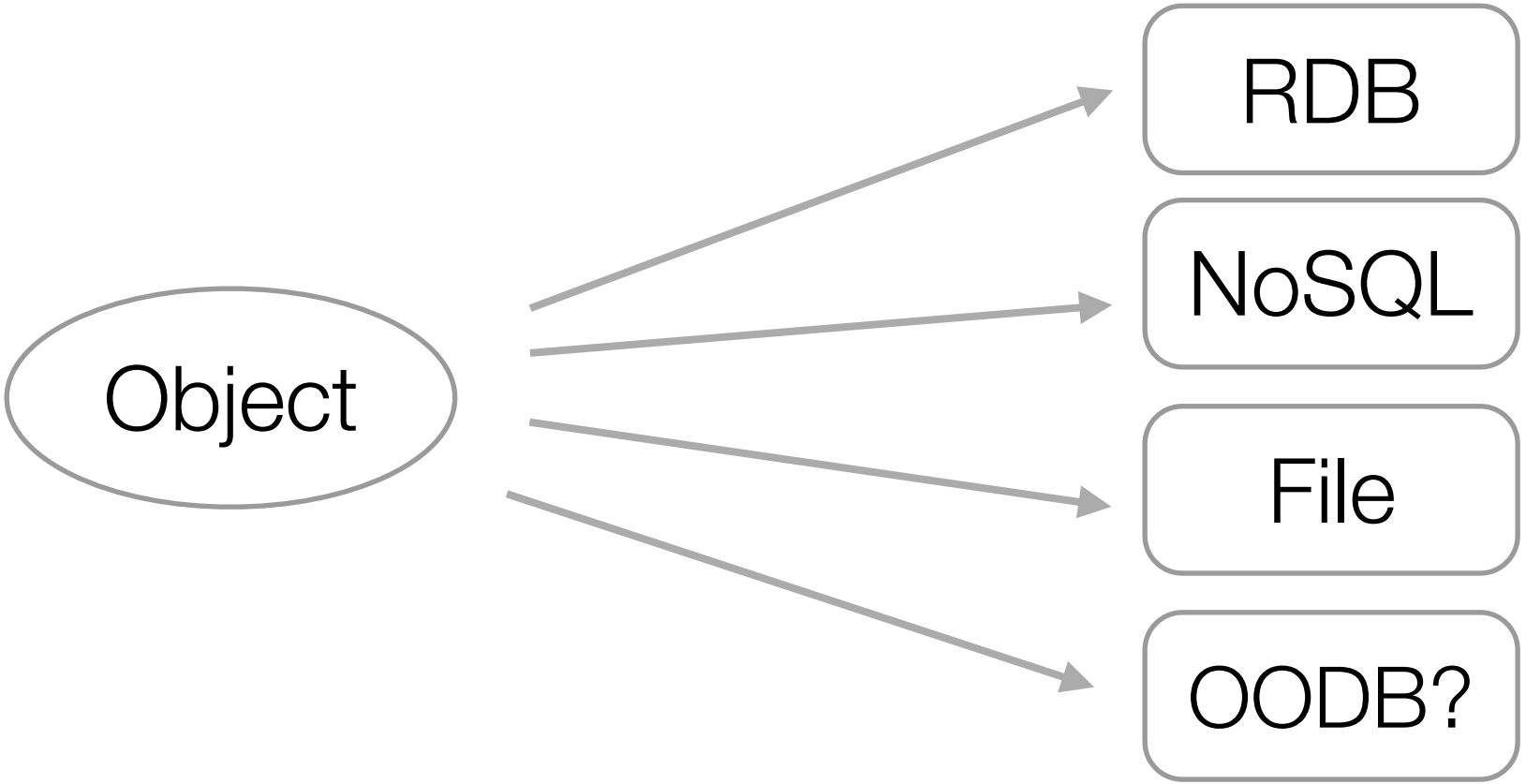
**SQL**에 **의존**적인 개발을
피하기 어렵다.


패러다임의 불일치
객체 vs 관계형 데이터베이스

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

–어느 객체지향 개발자가

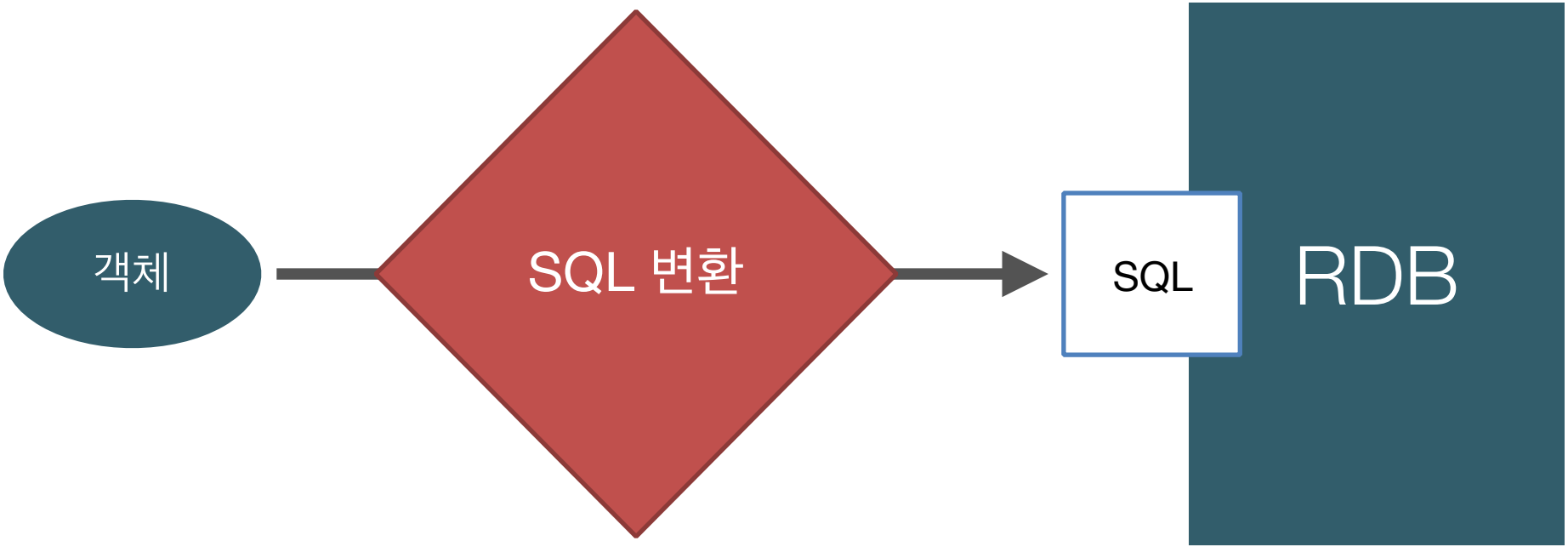
# 객체를 영구 보관하는 다양한 저장소



현실적인 대안은

**관계형 데이터베이스**

# 객체를 관계형 데이터베이스에 저장

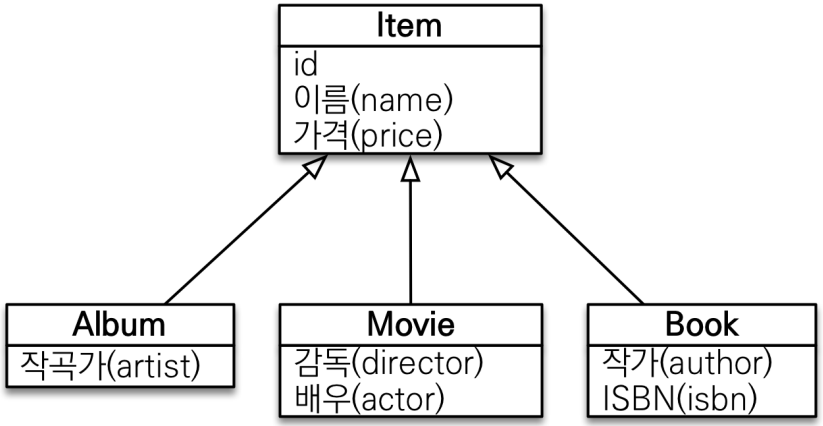



개발자 ≒ SQL 매퍼

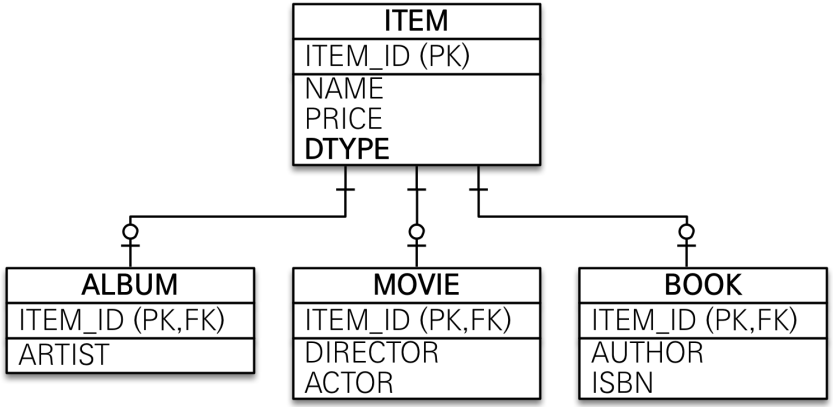
# 객체와 관계형 데이터베이스의 차이

1. 상속
2. 연관관계
3. 데이터 타입
4. 데이터 식별 방법

# 상속




[객체 상속 관계]




[Table 슈퍼타입 서브타입 관계]

# Album 저장

1. 객체 분해
2. INSERT INTO ITEM ...
3. INSERT INTO ALBUM …

# Album 조회

1. 각각의 테이블에 따른 조인 SQL 작성...
2. 각각의 객체 생성...
3. 상상만 해도 복잡
4. 더 이상의 설명은 생략한다.
5. 그래서 **DB에 저장할 객체에는 상속 관계 안쓴다.**

# 자바 컬렉션에 저장하면?

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

# 자바 컬렉션에서 조회하면?

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

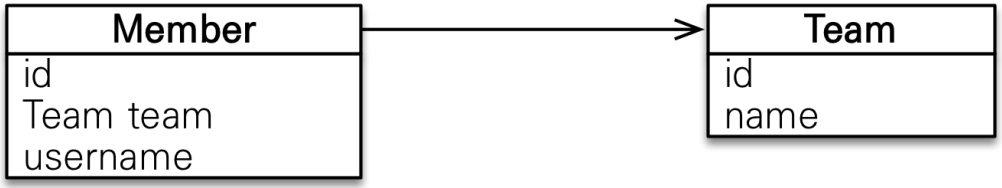
부모 타입으로 조회 후 다형성 활용

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

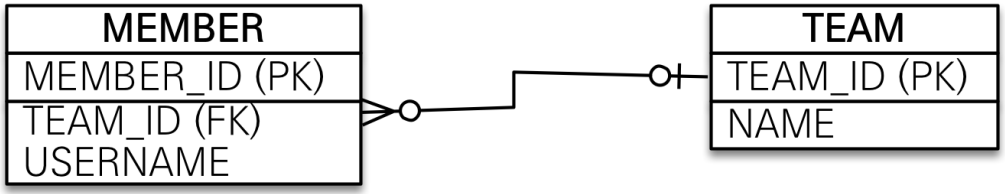
# 연관관계

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

**[객체 연관관계]**




**[테이블 연관관계]**

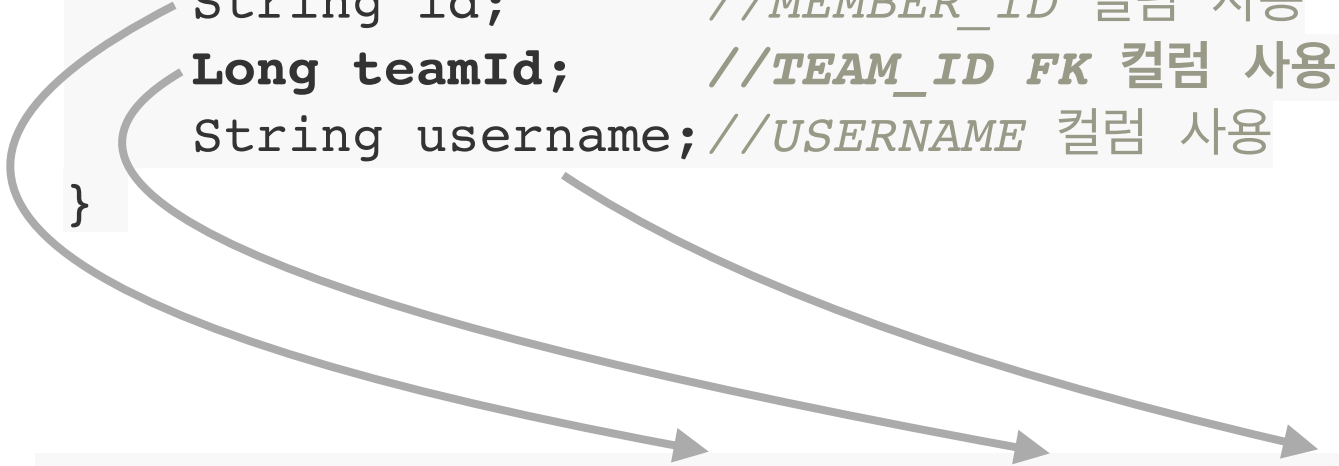


# 객체를 테이블에 맞추어 모델링

```
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
}
```

# 테이블에 맞춘 객체 저장

```
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 …
```

# 객체다운 모델링

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

    Team getTeam() {
        return team;
    }
}

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

# 객체 모델링 저장

```
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 …
```

# 객체 모델링 조회

```
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;
}
```

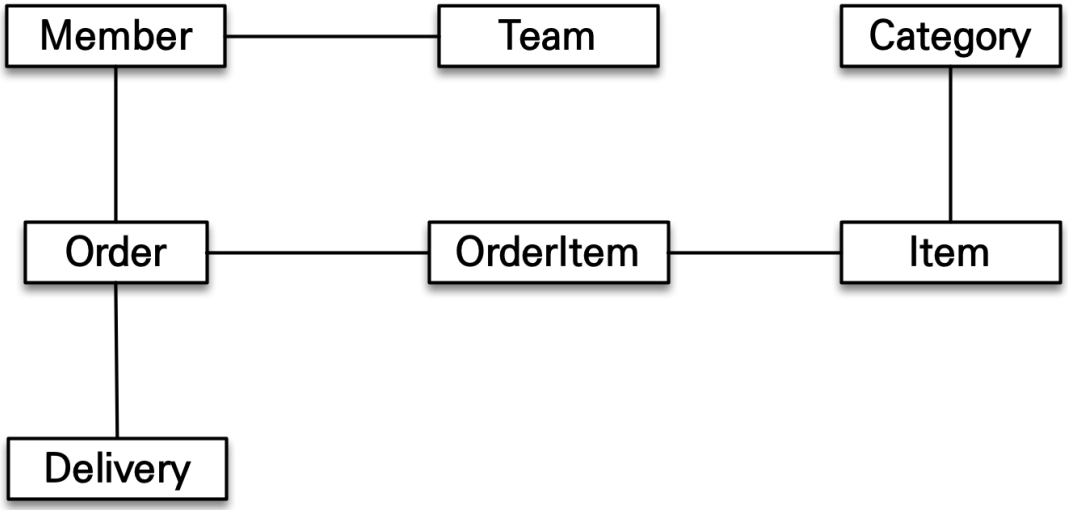
# 객체 모델링, 자바 컬렉션에 관리

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

```
Member member = list.get(memberId);
Team team = member.getTeam();
```

# 객체 그래프 탐색

객체는 자유롭게 객체 그래프를 탐색할 수 있어야 한다.



# 처음 실행하는 SQL에 따라 탐색 범위 결정

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

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

# 엔티티 신뢰 문제

```
class MemberService {
    ...
    public void process() {
        Member member = memberDAO.find(memberId);
        member.getTeam(); //???
        member.getOrder().getDelivery(); // ???
    }
}
```

# 모든 객체를 미리 로딩할 수는 없다.

상황에 따라 동일한 회원 조회 메서드를 여러벌 생성

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

memberDAO.getMemberWithTeam();//Member와 Team 조회

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

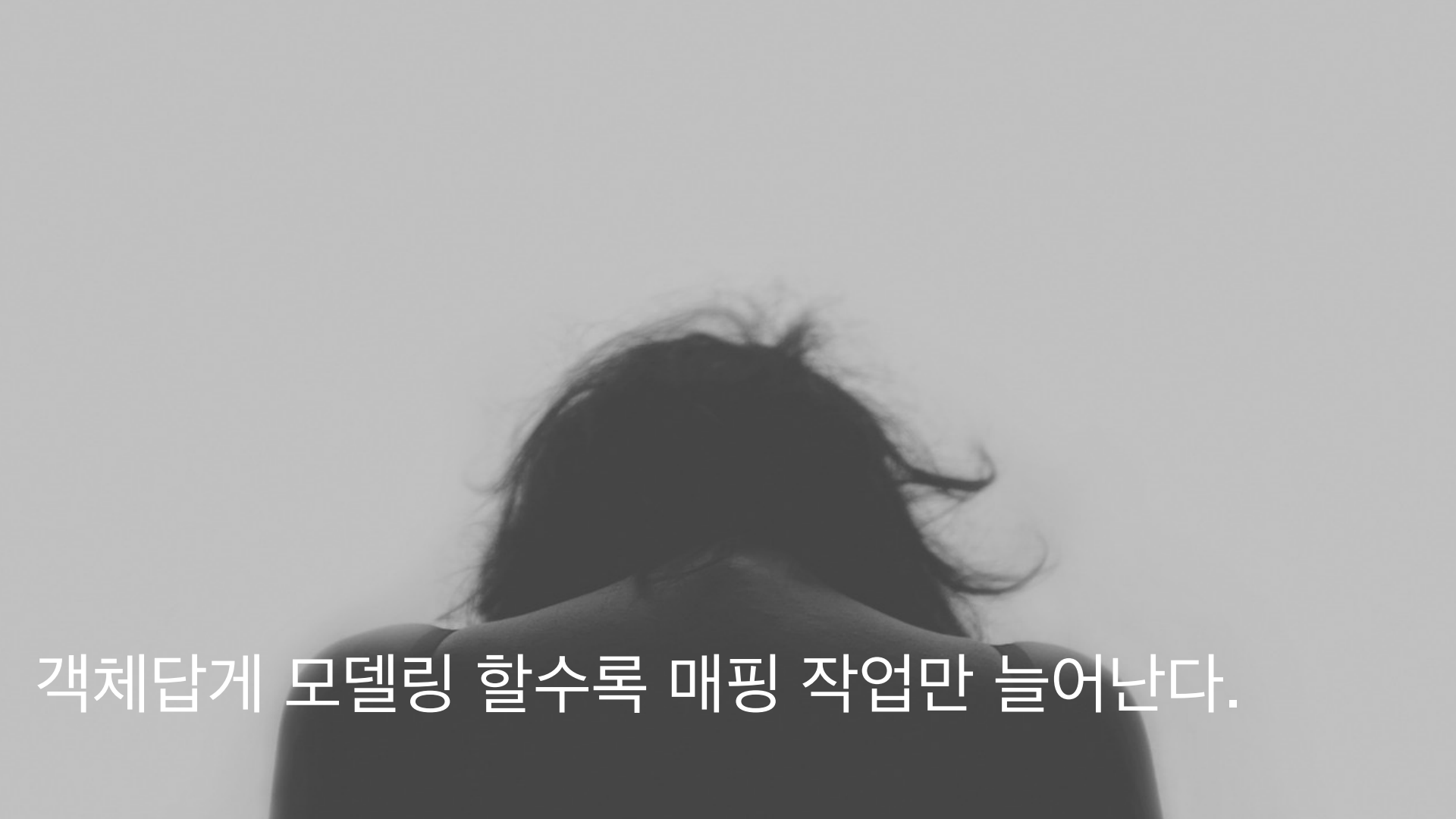
계층형 아키텍처
진정한 의미의 **계층 분할**이 어렵다.

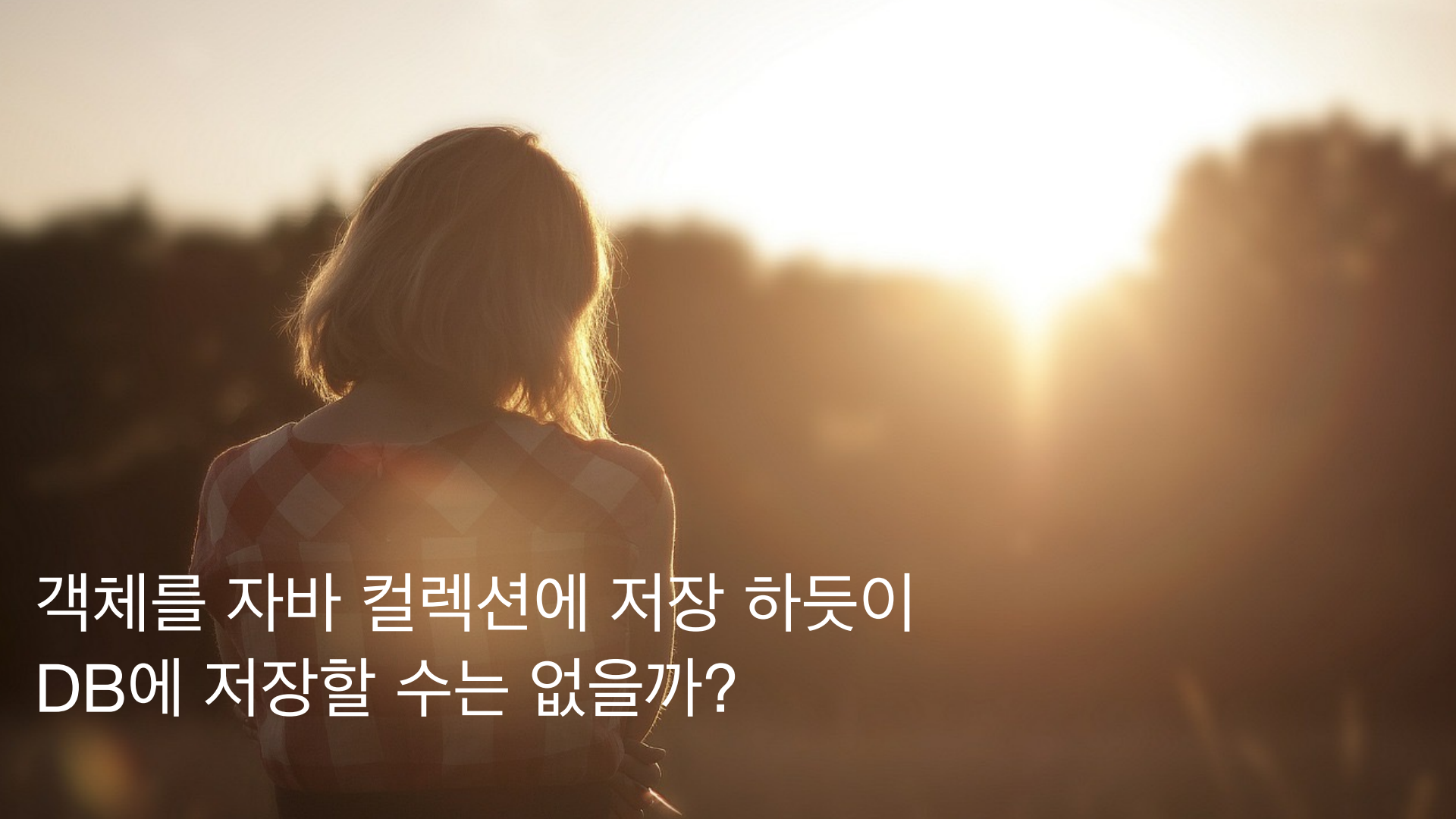
# 비교하기

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

member1 == member2; //다르다.

class MemberDAO {

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

# 비교하기 - 자바 컬렉션에서 조회

```
String memberId = "100";
Member member1 = list.get(memberId);
Member member2 = list.get(memberId);

member1 == member2; //같다.
```


객체답게 모델링 할수록 매핑 작업만 늘어난다.


객체를 자바 컬렉션에 저장 하듯이
DB에 저장할 수는 없을까?


JPA - Java Persistence API

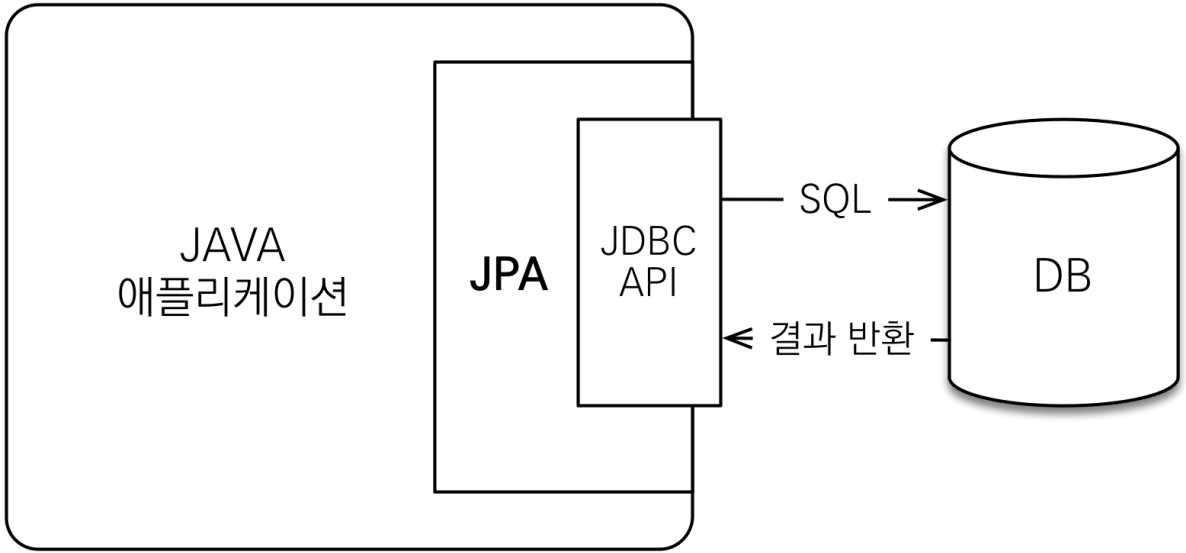
# JPA?

- Java Persistence API
- 자바 진영의 **ORM** 기술 표준

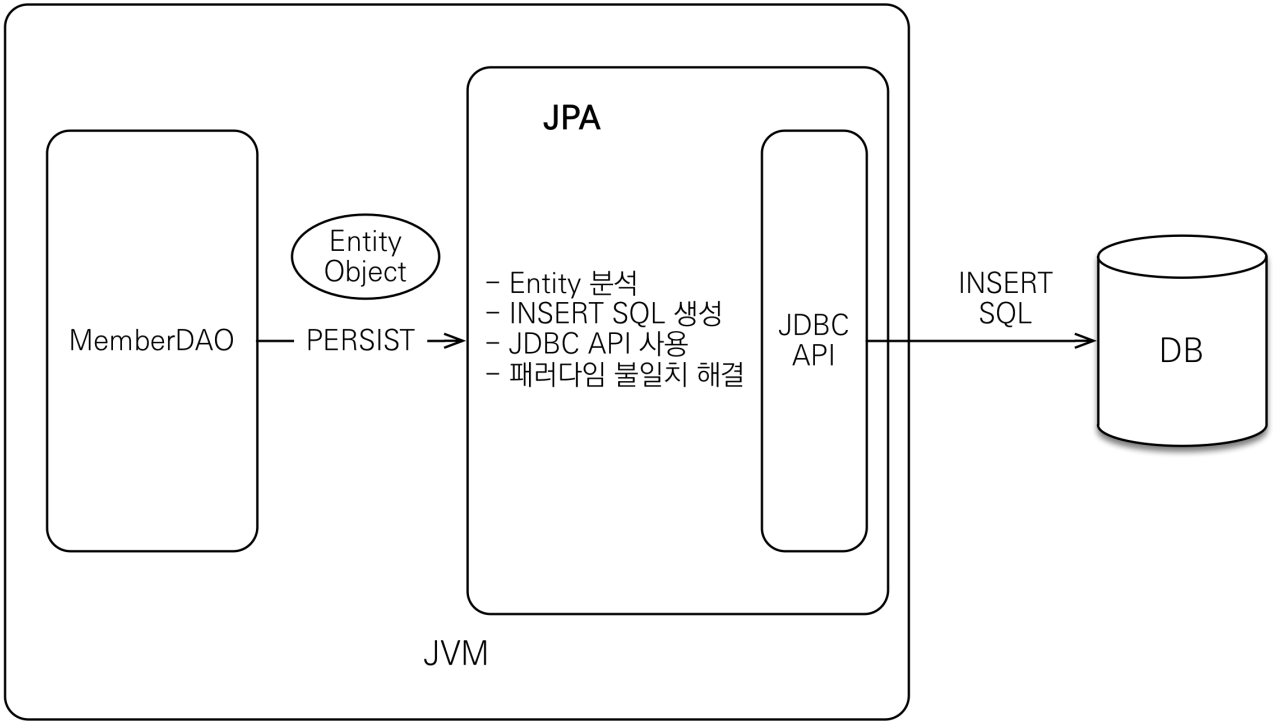
# ORM?

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

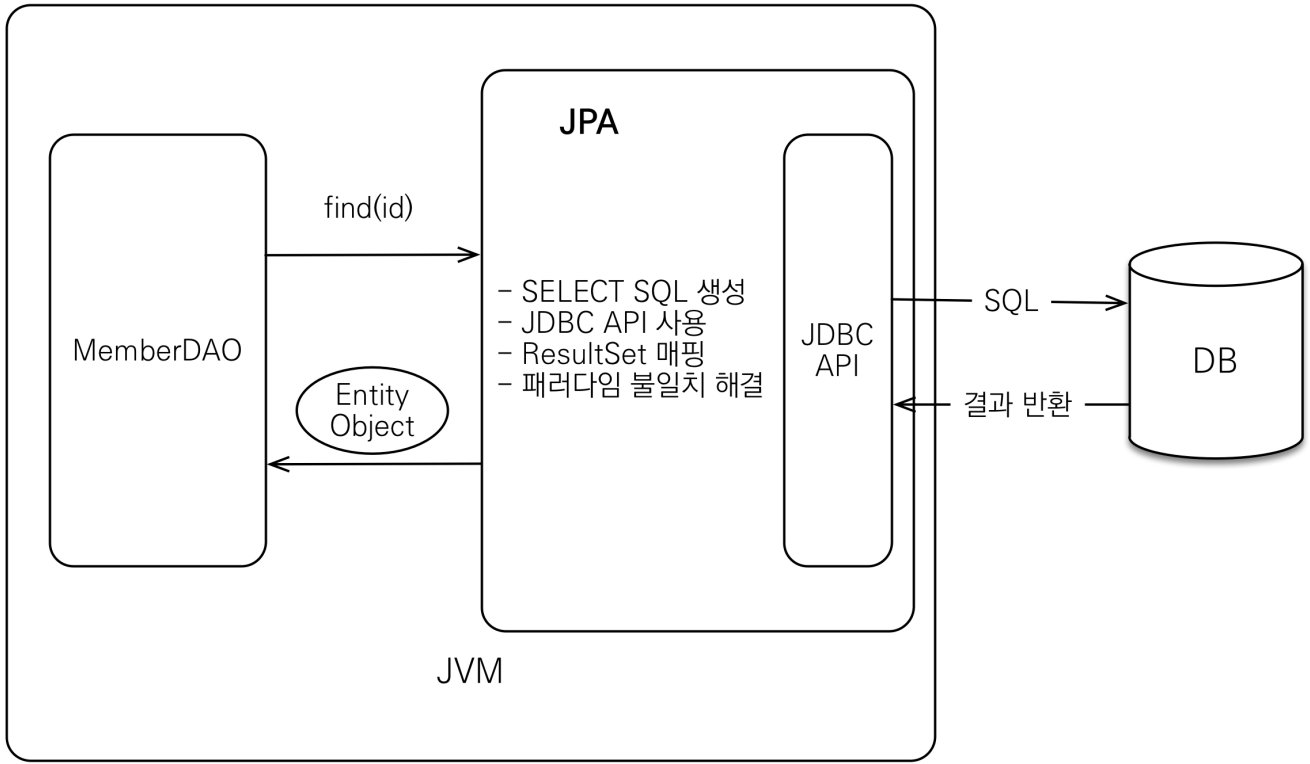
# JPA는 애플리케이션과 JDBC 사이에서 동작



# JPA 동작 - 저장


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

# JPA 동작 - 조회


JPA
find(id)
MemberDAO
Entity
Object
- SELECT SQL 생성
- JDBC API 사용
- ResultSet 매핑
- 패러다임 불일치 해결
JDBC
API
SQL
결과 반환
DB
JVM

# JPA 소개

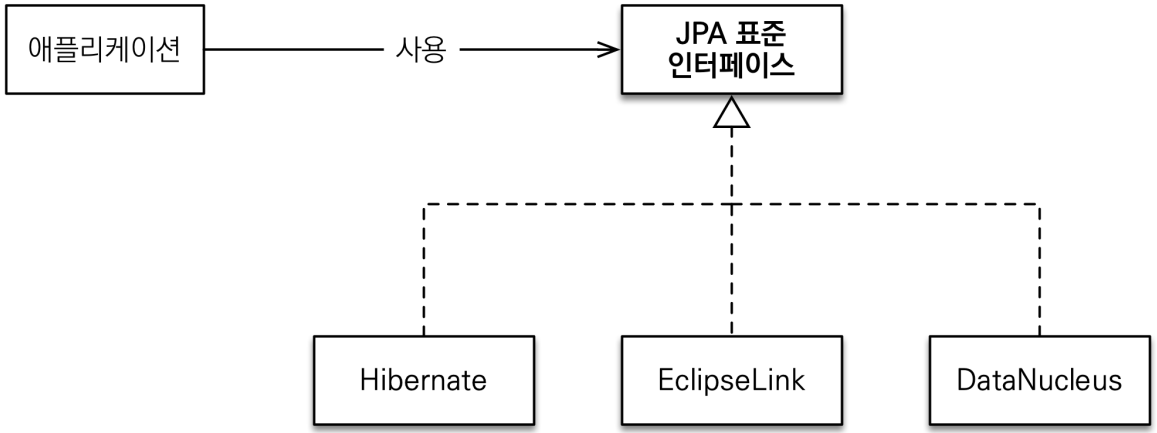
하이버네이트
(오픈 소스)

EJB - 엔티티 빈(자바 표준)

JPA(자바 표준)

# JPA는 표준 명세

- JPA는 인터페이스의 모음
- JPA 2.1 표준 명세를 구현한 3가지 구현체
- 하이버네이트, EclipseLink, DataNucleus



# JPA 버전

- JPA 1.0(JSR 220) 2006년 : 초기 버전. 복합 키와 연관관계 기능이 부족
- JPA 2.0(JSR 317) 2009년 : 대부분의 ORM 기능을 포함, JPA Criteria 추가
- JPA 2.1(JSR 338) 2013년 : 스토어드 프로시저 접근, 컨버터(Converter), 엔티티 그래프 기능이 추가

# JPA를 왜 사용해야 하는가?

- SQL 중심적인 개발에서 객체 중심으로 개발
- 생산성
- 유지보수
- 패러다임의 불일치 해결
- 성능
- 데이터 접근 추상화와 벤더 독립성
- 표준

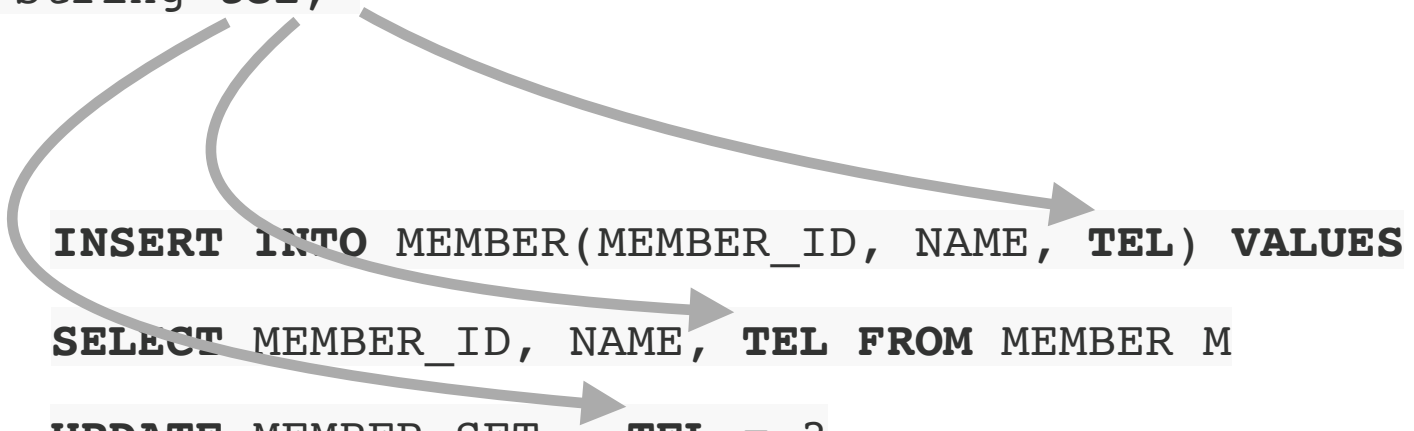
# 생산성 - JPA와 CRUD

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

# 유지보수 - 기존: 필드 변경시 모든 SQL 수정

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

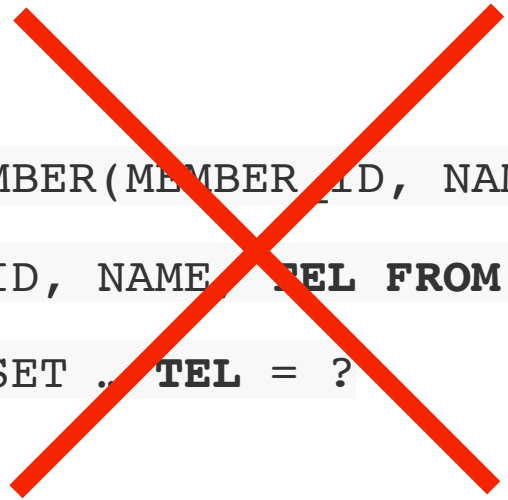
```
INSERT INTO MEMBER(MEMBER_ID, NAME, TEL) VALUES

SELECT MEMBER_ID, NAME, TEL FROM MEMBER M

UPDATE MEMBER SET … TEL = ?
```



# 유지보수 - JPA: 필드만 추가하면 됨, SQL은 JPA가 처리

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

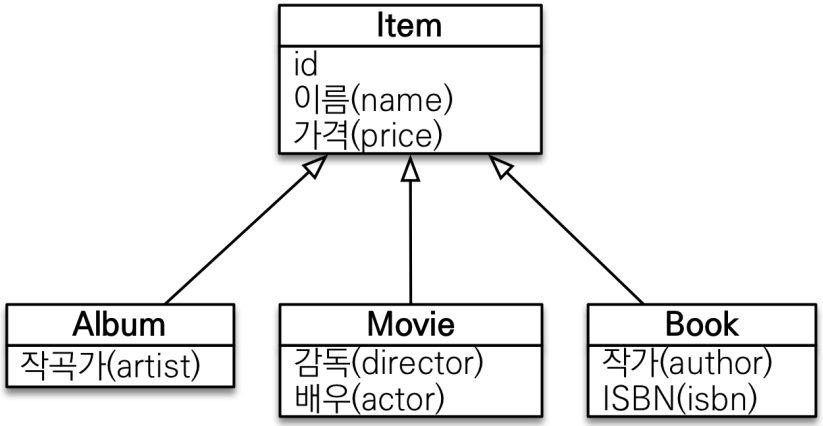
```
INSERT INTO MEMBER(MEMBER_ID, NAME, TEL) VALUES

SELECT MEMBER_ID, NAME, TEL FROM MEMBER M

UPDATE MEMBER SET ... TEL = ?
```



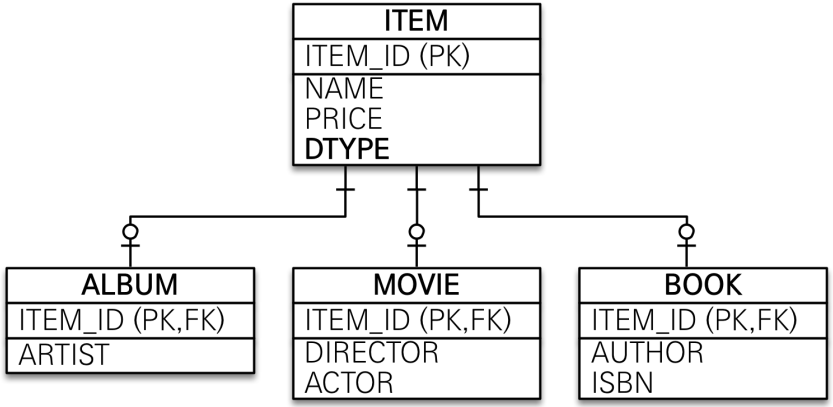
# JPA와 패러다임의 불일치 해결

1. JPA와 상속
2. JPA와 연관관계
3. JPA와 객체 그래프 탐색
4. JPA와 비교하기

# JPA와 상속




[객체 상속 관계]




[Table 슈퍼타입 서브타입 관계]

# JPA와 상속 - 저장

개발자가 할일

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

나머진 JPA가 처리

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

# JPA와 상속 - 조회

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
```

# JPA와 연관관계, 객체 그래프 탐색

연관관계 저장

```
member.setTeam(team);
jpa.persist(member);
```

객체 그래프 탐색

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

# 신뢰할 수 있는 엔티티, 계층

```
class MemberService {
    ...
    public void process() {
        Member member = memberDAO.find(memberId);
        member.getTeam(); //자유로운 객체 그래프 탐색
        member.getOrder().getDelivery();
    }
}
```

# JPA와 비교하기

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

member1 == member2; //같다.
```

동일한 트랜잭션에서 조회한 엔티티는 같음을 보장

# JPA의 성능 최적화 기능

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 …
```


ORM은 객체와 RDB
두 기둥위에 있는 기술