연관관계 매핑 기초

목표

- **객체와 테이블 연관관계의 차이를 이해**
- **객체의 참조와 테이블의 외래 키를 매핑**
- 용어 이해
  - **방향**(Direction): 단방향, 양방향
  - **다중성**(Multiplicity): 다대일(N:1), 일대다(1:N), 일대일(1:1), 다대다(N:M) 이해
  - **연관관계의 주인**(Owner): 객체 양방향 연관관계는 관리 주인이 필요

목차

연관관계가 필요한 이유

‘객체지향 설계의 목표는 자율적인 객체들의
협력 공동체를 만드는 것이다.’

–조영호(객체지향의 사실과 오해)

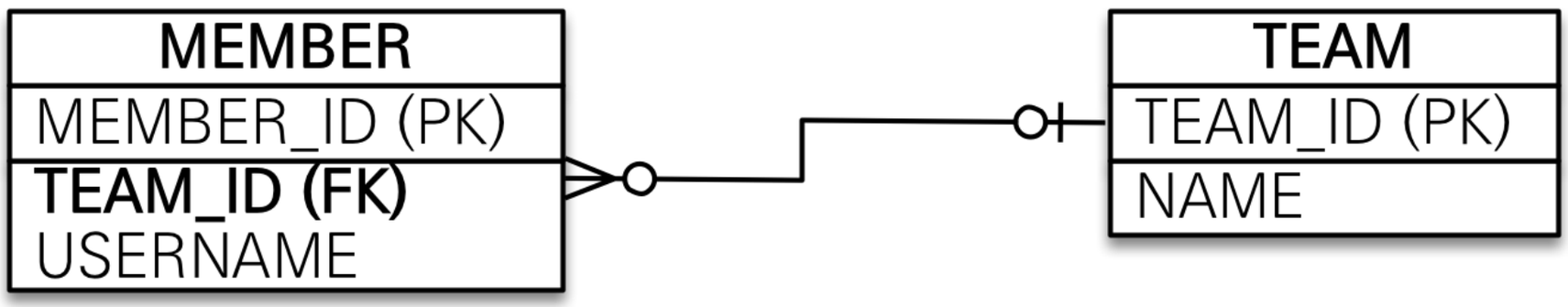
예제 시나리오

- 회원과 팀이 있다.
- 회원은 하나의 팀에만 소속될 수 있다.
- 회원과 팀은 다대일 관계다.

객체를 테이블에 맞추어 모델링
(연관관계가 없는 객체)

[객체 연관관계]

[테이블 연관관계]



객체를 테이블에 맞추어 모델링

(참조 대신에 외래 키를 그대로 사용)

```
@Entity
public class Member {

  @Id @GeneratedValue
  private Long id;

  @Column(name = "USERNAME")
  private String name;

  @Column(name = "TEAM_ID")
  private Long teamId;
  …
}
@Entity
public class Team {

  @Id @GeneratedValue
  private Long id;
  private String name;
  …
}
```

객체를 테이블에 맞추어 모델링
(외래 키 식별자를 직접 다룸)

```
//팀 저장
Team team = new Team();
team.setName("TeamA");
em.persist(team);

//회원 저장
Member member = new Member();
member.setName("member1");
member.setTeamId(team.getId());
em.persist(member);
```

객체를 테이블에 맞추어 모델링

(식별자로 다시 조회, 객체 지향적인 방법은 아니다.)

```
//조회
Member findMember = em.find(Member.class, member.getId());

//연관관계가 없음
Team findTeam = em.find(Team.class, team.getId());
```

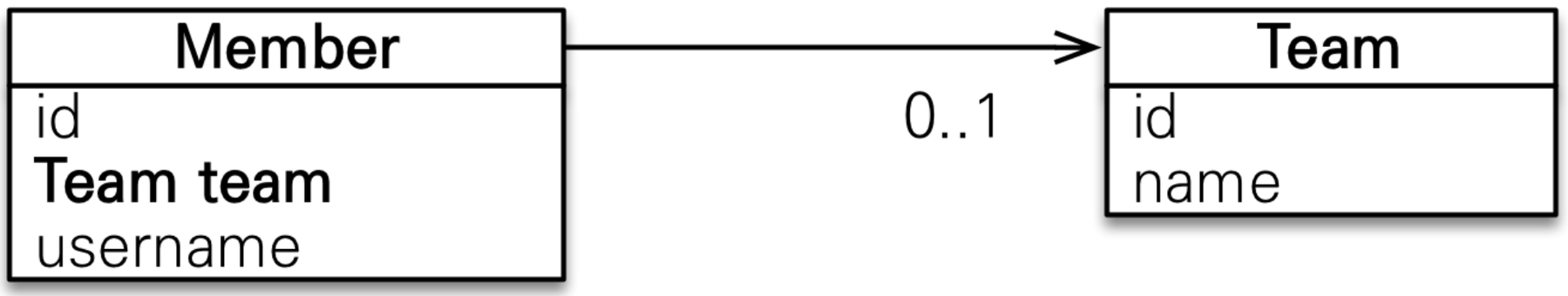
객체를 테이블에 맞추어 데이터 중심으로 모델링하면,
협력 관계를 만들 수 없다.

- **테이블은 외래 키로 조인**을 사용해서 연관된 테이블을 찾는다.
- **객체는 참조**를 사용해서 연관된 객체를 찾는다.
- 테이블과 객체 사이에는 이런 큰 간격이 있다.

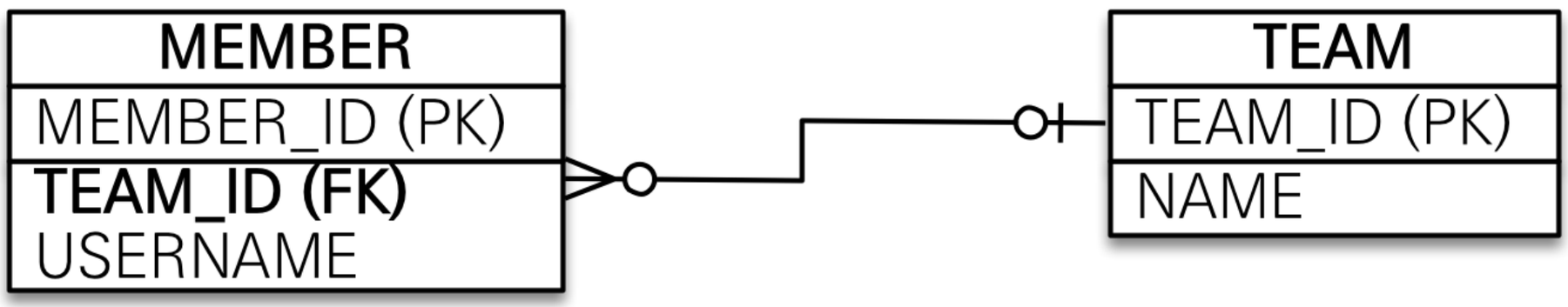
단방향 연관관계

객체 지향 모델링
(객체 연관관계 사용)

[객체 연관관계]



[테이블 연관관계]

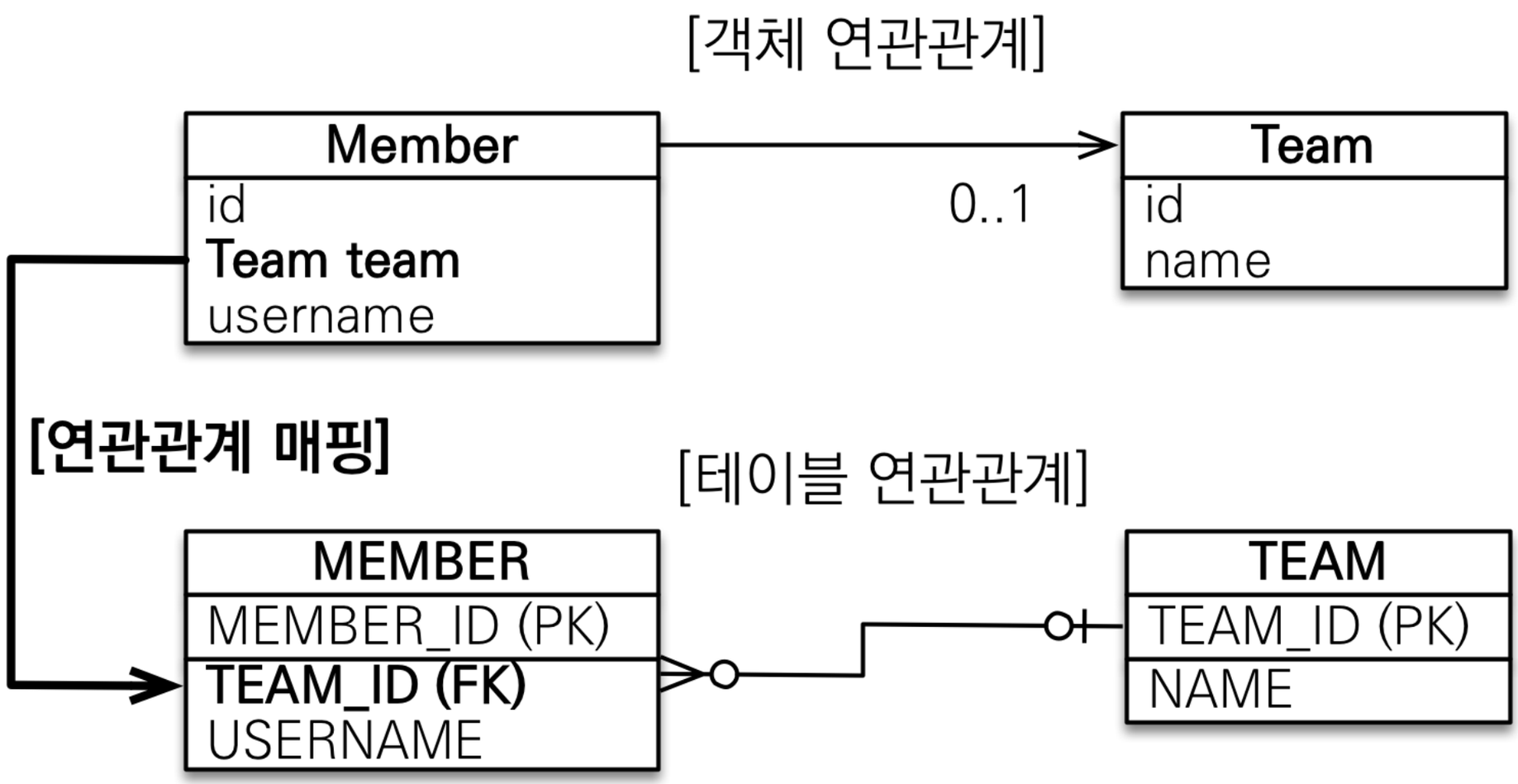


객체 지향 모델링

(객체의 참조와 테이블의 외래 키를 매핑)

```
@Entity
public class Member {

    @Id @GeneratedValue
    private Long id;

    @Column(name = "USERNAME")
    private String name;
    private int age;

//    @Column(name = "TEAM_ID")
//    private Long teamId;

    @ManyToOne
    @JoinColumn(name = "TEAM_ID")
    private Team team;
    …
```

객체 지향 모델링
(ORM 매핑)



객체 지향 모델링
(연관관계 저장)

```
//팀 저장
Team team = new Team();
team.setName("TeamA");
em.persist(team);
```

```
//회원 저장
Member member = new Member();
member.setName("member1");
 member.setTeam(team); //단방향 연관관계 설정, 참조 저장
em.persist(member);
```

객체 지향 모델링
(참조로 연관관계 조회 - 객체 그래프 탐색)

```
//조회
Member findMember = em.find(Member.class, member.getId());

//참조를 사용해서 연관관계 조회
Team findTeam = findMember.getTeam();
```

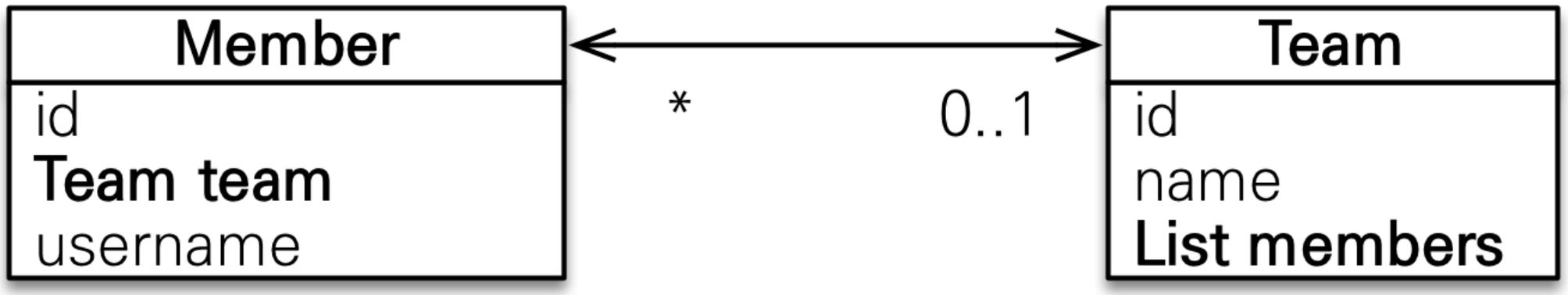
객체 지향 모델링
(연관관계 수정)

```
// 새로운 팀B
Team teamB = new Team();
teamB.setName("TeamB");
em.persist(teamB);

// 회원1에 새로운 팀B 설정
member.setTeam(teamB);
```

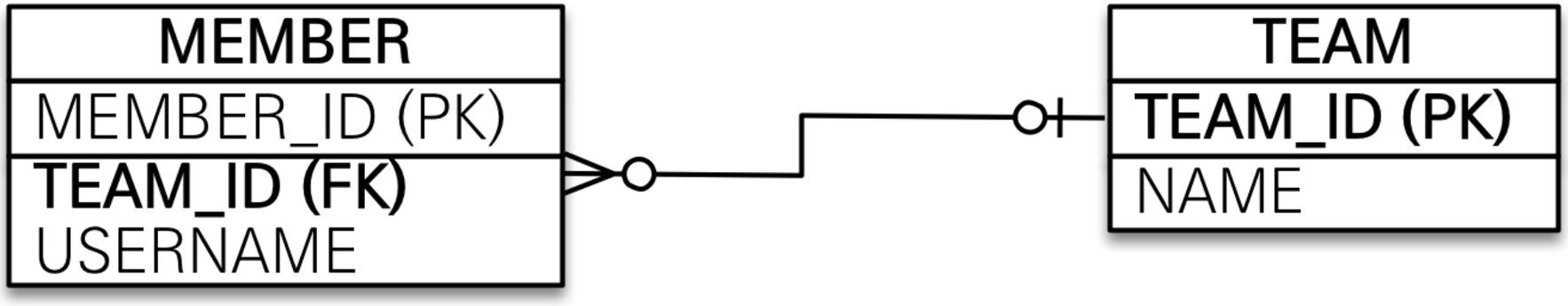
양방향 연관관계와 연관관계의 주인

양방향 매핑

[양방향 객체 연관관계]



[테이블 연관관계]



양방향 매핑

(Member 엔티티는 단방향과 동일)

```
@Entity
public class Member {

  @Id @GeneratedValue
  private Long id;

  @Column(name = "USERNAME")
  private String name;
  private int age;

  @ManyToOne
  @JoinColumn(name = "TEAM_ID")
  private Team team;
  …
```

양방향 매핑

(Team 엔티티는 컬렉션 추가)

```
@Entity
public class Team {

  @Id @GeneratedValue
  private Long id;

  private String name;

  @OneToMany(mappedBy = "team")
  List<Member> members = new ArrayList<Member>();
  …
}
```

양방향 매핑

(반대 방향으로 객체 그래프 탐색)

```
//조회
Team findTeam = em.find(Team.class, team.getId());

int memberSize = findTeam.getMembers().size(); //역방향 조회
```

연관관계의 주인과 mappedBy

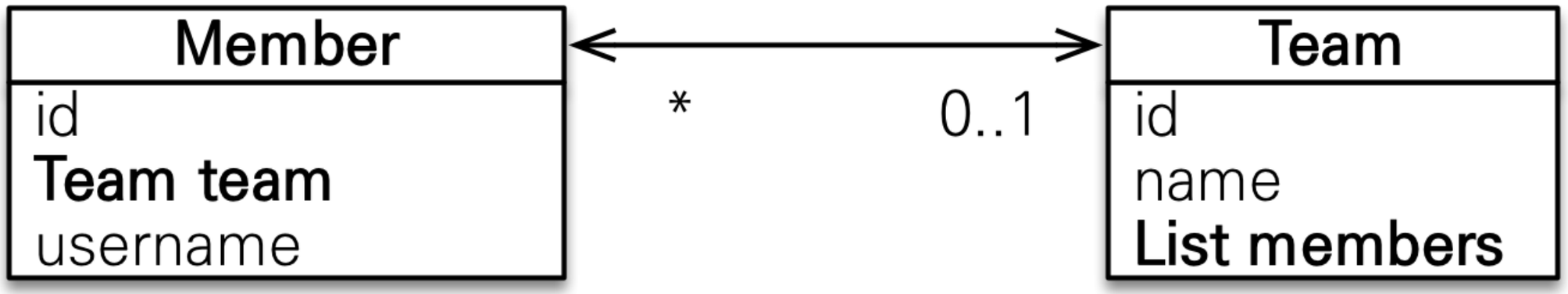
- mappedBy = JPA의 멘탈붕괴 난이도
- mappedBy는 처음에는 이해하기 어렵다.
- 객체와 테이블간에 연관관계를 맺는 차이를 이해해야 한다.

객체와 테이블이 관계를 맺는 차이

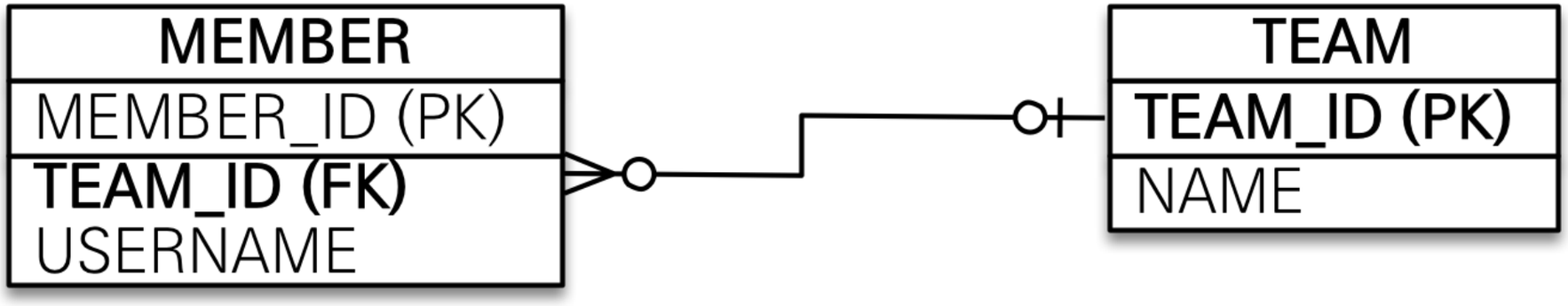
- **객체 연관관계 = 2개**
    - 회원 -> 팀 연관관계 1개(단방향)
    - 팀 -> 회원 연관관계 1개(단방향)
- **테이블 연관관계 = 1개**
    - 회원 <-> 팀의 연관관계 1개(양방향)

객체와 테이블이 관계를 맺는 차이

[양방향 객체 연관관계]



[테이블 연관관계]



객체의 양방향 관계

- 객체의 **양방향 관계는 사실 양방향 관계가 아니라 서로 다른 단방향 관계 2개다.**

- 객체를 양방향으로 참조하려면 **단방향 연관관계를 2개** 만들어야 한다.

- A -> B (a.getB())
- B -> A (b.getA())

```
class A {
    B b;
}
```

```
class B {
    A a;
}
```

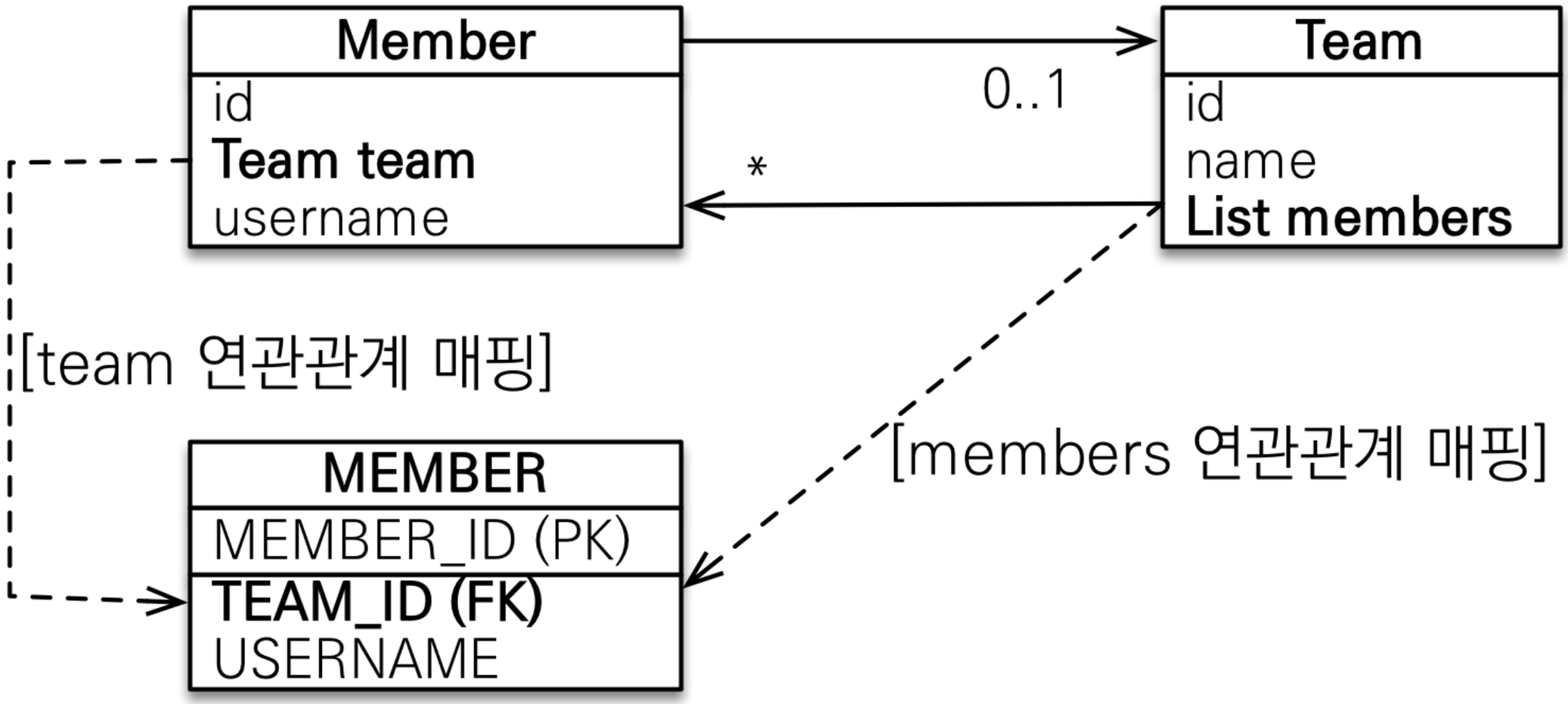
테이블의 양방향 연관관계

- 테이블은 **외래 키 하나**로 두 테이블의 연관관계를 관리
- MEMBER.TEAM_ID 외래 키 하나로 양방향 연관관계 가짐
  (양쪽으로 조인할 수 있다.)

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

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

둘 중 하나로 외래 키를 관리해야 한다.



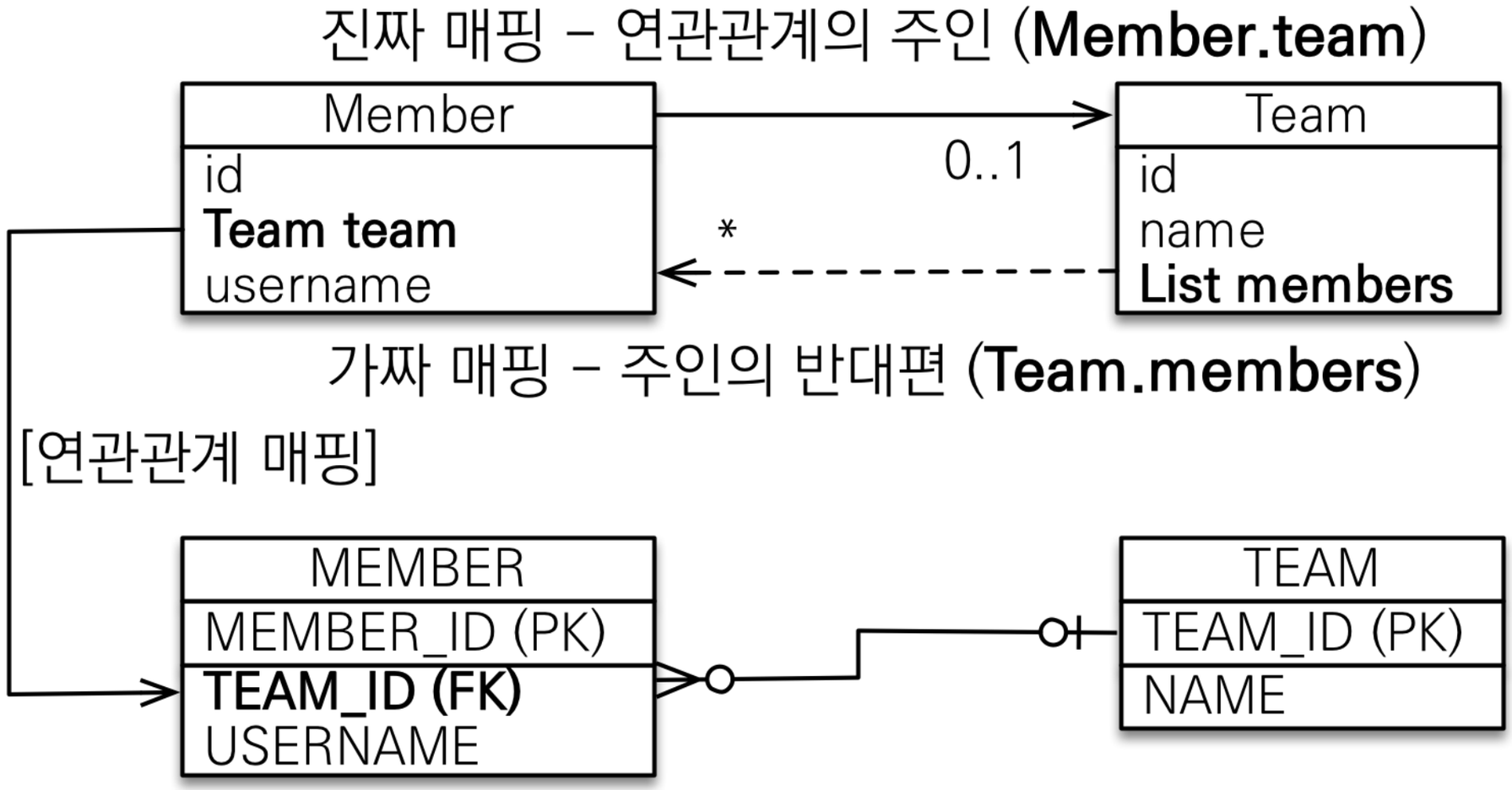
연관관계의 주인(Owner)

양방향 매핑 규칙

- 객체의 두 관계중 하나를 연관관계의 주인으로 지정
- **연관관계의 주인만이 외래 키를 관리(등록, 수정)**
- **주인이 아닌쪽은 읽기만 가능**
- 주인은 mappedBy 속성 사용X
- 주인이 아니면 mappedBy 속성으로 주인 지정

누구를 주인으로?

- 외래 키가 있는 있는 곳을 주인으로 정해라
- 여기서는 **Member.team**이 연관관계의 주인



양방향 매핑시 가장 많이 하는 실수

(연관관계의 주인에 값을 입력하지 않음)

```
Team team = new Team();
team.setName("TeamA");
em.persist(team);

Member member = new Member();
member.setName("member1");

//역방향(주인이 아닌 방향)만 연관관계 설정
team.getMembers().add(member);

em.persist(member);
```

ID	USERNAME	TEAM_ID
1	member1	**null**

양방향 매핑시 연관관계의 주인에 값을 입력해야 한다.

(순수한 객체 관계를 고려하면 항상 양쪽다 값을 입력해야 한다.)

```
Team team = new Team();
team.setName("TeamA");
em.persist(team);

Member member = new Member();
member.setName("member1");

team.getMembers().add(member);
//연관관계의 주인에 값 설정
member.setTeam(team); //**

em.persist(member);
```

ID	USERNAME	TEAM_ID
1	member1	**2**

양방향 연관관계 주의 - 실습

- **순수 객체 상태를 고려해서 항상 양쪽에 값을 설정하자**
- 연관관계 편의 메소드를 생성하자
- 양방향 매핑시에 무한 루프를 조심하자
  - 예: toString(), lombok, JSON 생성 라이브러리

양방향 매핑 정리

- **단방향 매핑만으로도 이미 연관관계 매핑은 완료**

- 양방향 매핑은 반대 방향으로 조회(객체 그래프 탐색) 기능이 추가된 것 뿐

- JPQL에서 역방향으로 탐색할 일이 많음

- 단방향 매핑을 잘 하고 양방향은 필요할 때 추가해도 됨
(테이블에 영향을 주지 않음)

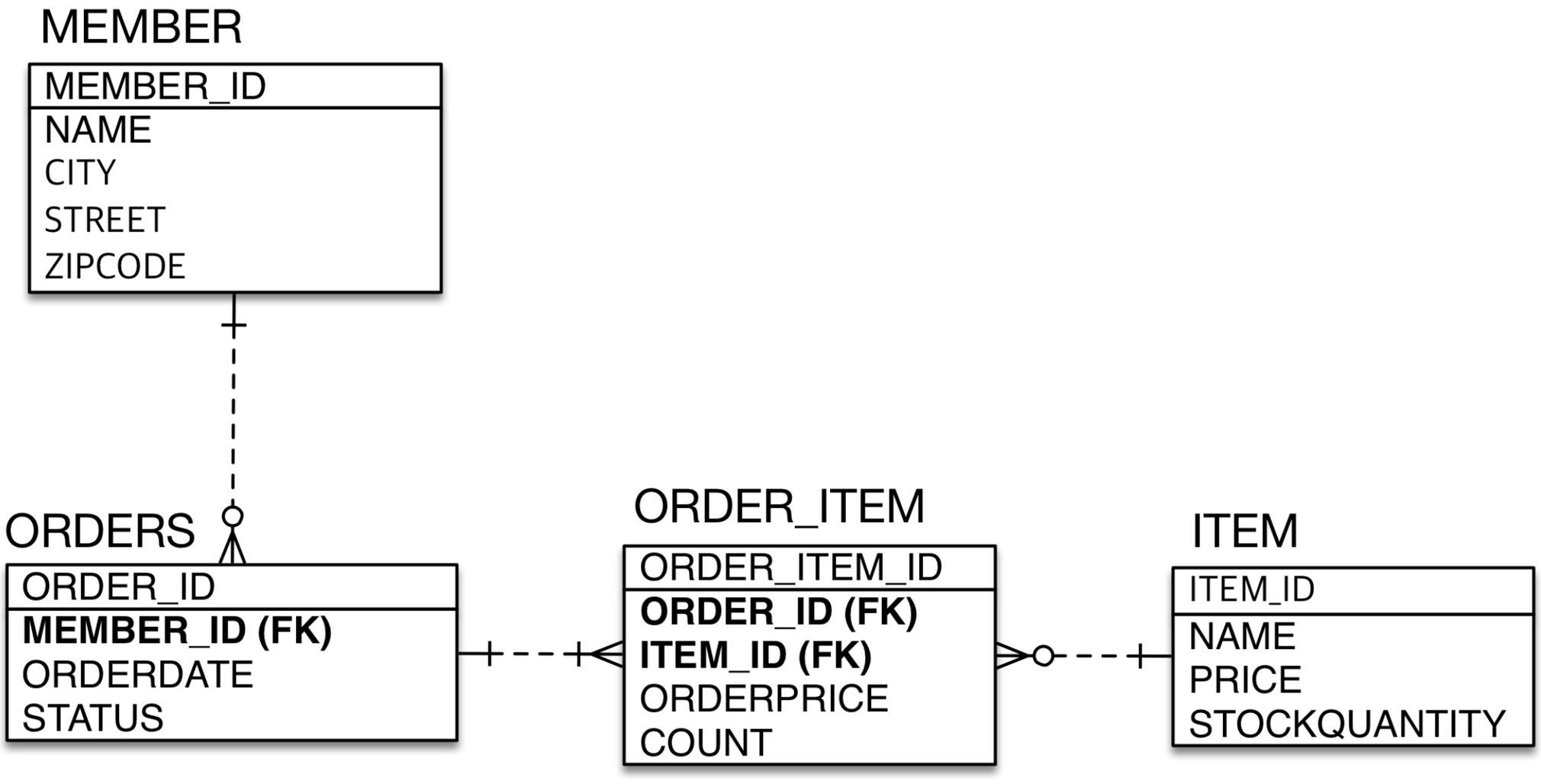
연관관계의 주인을 정하는 기준

- 비즈니스 로직을 기준으로 연관관계의 주인을 선택하면 안됨
- **연관관계의 주인은 외래 키의 위치를 기준으로 정해야함**

실전 예제 - 2. 연관관계 매핑 시작

테이블 구조

- 테이블 구조는 이전과 같다.



객체 구조

- 참조를 사용하도록 변경