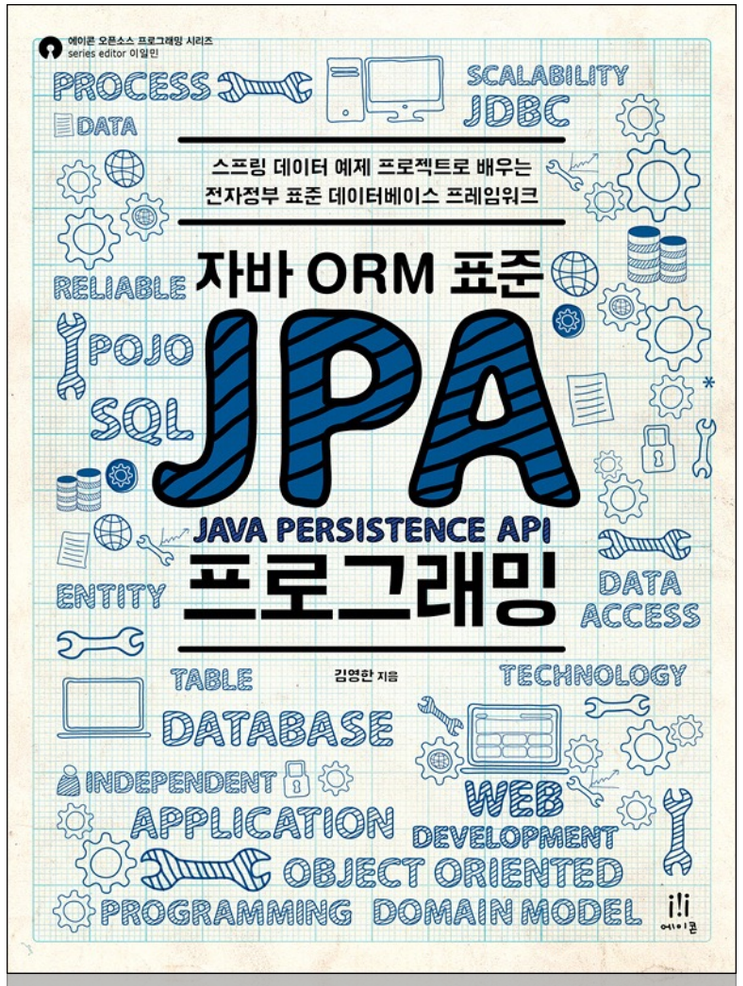
JPA 기초와 매핑

실습

김영한

SI, J2EE 강사, DAUM, SK 플래닛
우아한형제들

저서: 자바 ORM 표준 **JPA** 프로그래밍



목차

Hello JPA

예제 ex01

H2 데이터베이스

- http://www.h2database.com/
- 최고의 실습용 DB
- 가볍다.(1.5M)
- 웹용 쿼리툴 제공
- MySQL, Oracle 데이터베이스 시뮬레이션 기능
- 시퀀스, AUTO INCREMENT 기능 지원

메이븐 설정

- https://maven.apache.org/
- 자바 라이브러리, 빌드 관리
- 라이브러리 자동 다운로드 및 의존성 관리

객체 매핑하기

- **@Entity:** JPA가 관리할 객체
  - 엔티티라 한다.
- **@Id:** DB PK와 매핑 할 필드

```java
@Entity
public class Member {

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

```sql
create table Member (
    id bigint not null,
    name varchar(255),
    primary key (id)
)
```

persistence.xml

- JPA 설정 파일
- /META-INF/persistence.xml 위치
- javax.persistence로 시작: JPA 표준 속성
- hibernate로 시작: 하이버네이트 전용 속성

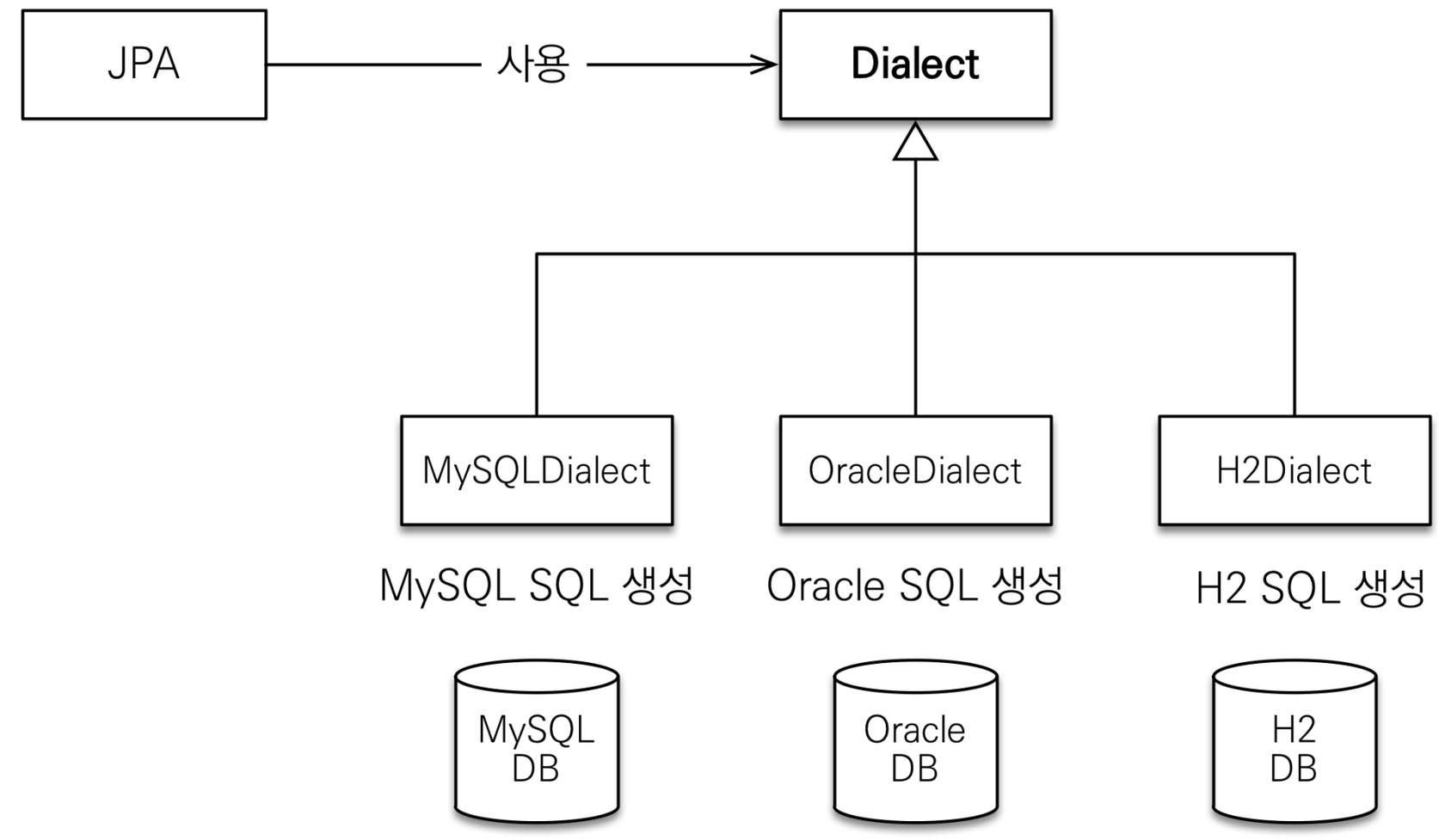
persistence.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="“2.2">

    <persistence-unit name="hello">
        <properties>

            <!-- 필수 속성 -->
            <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
            <property name="javax.persistence.jdbc.user" value="sa"/>
            <property name="javax.persistence.jdbc.password" value=""/>
            <property name="javax.persistence.jdbc.url" value="jdbc:h2:tcp://localhost/~/test"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />

            <!-- 옵션 -->
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.use_sql_comments" value="true" />
            <property name="hibernate.id.new_generator_mappings" value="true" />

            <!--<property name="hibernate.hbm2ddl.auto" value="create" />-->
        </properties>
    </persistence-unit>
</persistence>
```

데이터베이스 방언

- JPA는 특정 데이터베이스에 종속적이지 않은 기술
- 각각의 데이터베이스가 제공하는 SQL 문법과 함수는 조금씩 다르다
    - 가변 문자: MySQL은 VARCHAR, Oracle은 VARCHAR2
    - 문자열을 자르는 함수: SQL 표준은 SUBSTRING(), Oracle은 SUBSTR()
    - 페이징: MySQL은 LIMIT , Oracle은 ROWNUM
- 방언: SQL 표준을 지키지 않거나 특정 데이터베이스만의 고유한 기능

데이터베이스 방언


JPA
사용
Dialect
MySQLDialect
OracleDialect
H2Dialect
MySQL SQL 생성
Oracle SQL 생성
H2 SQL 생성
MySQL DB
Oracle DB
H2 DB

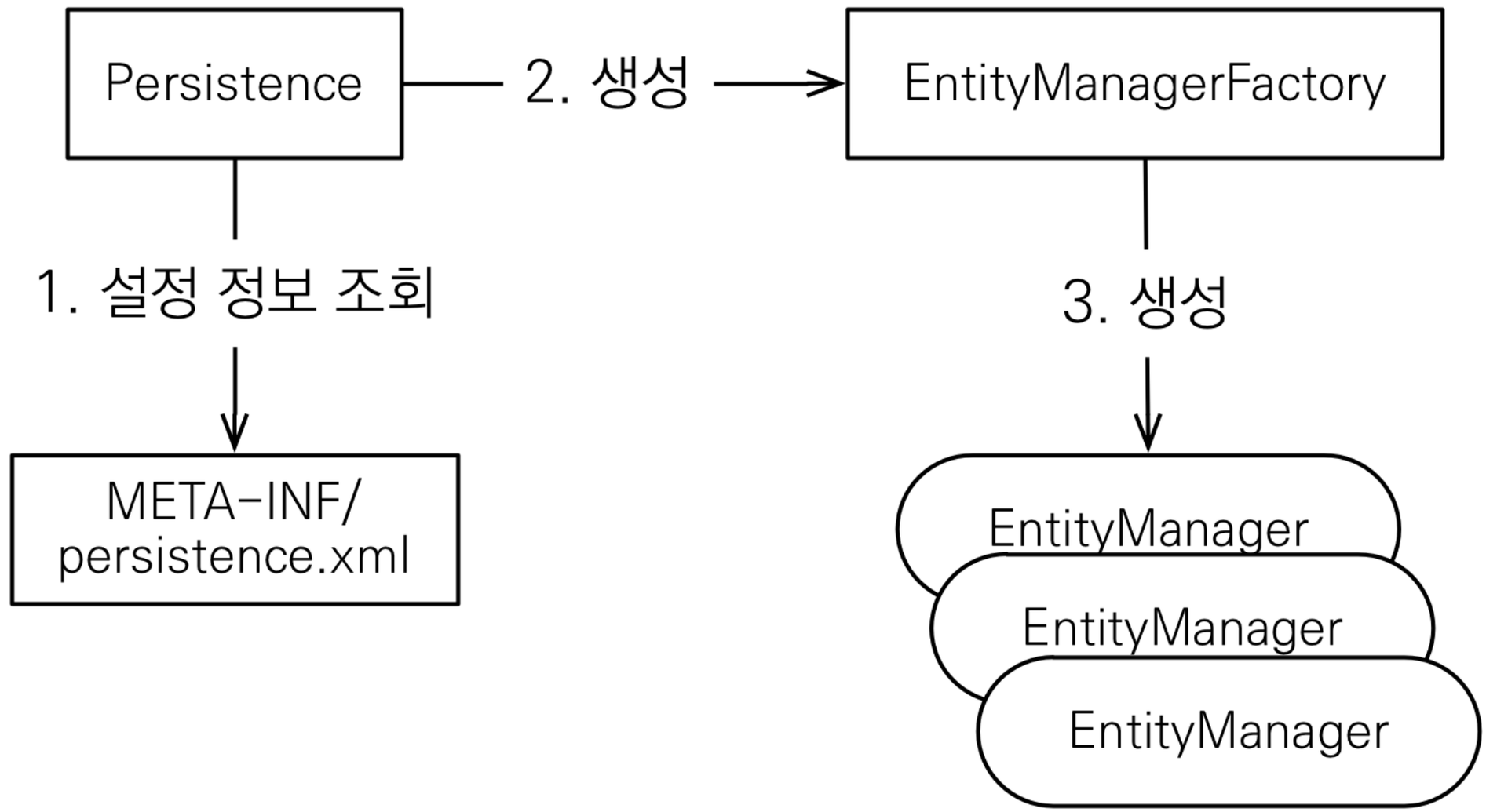
데이터베이스 방언

- hibernate.dialect 속성에 지정
  - H2 : org.hibernate.dialect.H2Dialect
  - Oracle 10g : org.hibernate.dialect.Oracle10gDialect
  - MySQL : org.hibernate.dialect.MySQL5InnoDBDialect
- 하이버네이트는 45가지 방언 지원

애플리케이션 개발

- 엔티티 매니저 팩토리 설정
- 엔티티 매니저 설정
- 트랜잭션
- 비즈니스 로직 (CRUD)

엔티티 매니저 설정


Persistence
2. 생성
EntityManagerFactory
1. 설정 정보 조회
3. 생성
META-INF/
persistence.xml
EntityManager
EntityManager
EntityManager

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory(“hello");

EntityManager em = emf.createEntityManager();

EntityTransaction tx = em.getTransaction();
tx.begin();

try {

    Member member = new Member();
    member.setId(1L);
    member.setName("hello");

    em.persist(member);

    tx.commit();
} catch (Exception e) {
    tx.rollback();
} finally {
    em.close();
}

emf.close();
```

주의

- 엔티티 매니저 팩토리는 하나만 생성해서 애플리케이션 전체에서 공유
- 엔티티 매니저는 쓰레드간에 공유하면 안된다(사용하고 버려야 한다).
- JPA의 모든 데이터 변경은 트랜잭션 안에서 실행

필드와 컬럼 매핑

예제 ex02

데이터베이스 스키마 자동 생성하기

- DDL을 애플리케이션 실행 시점에 자동 생성
- 테이블 중심 -> 객체 중심
- 데이터베이스 방언을 활용해서 데이터베이스에 맞는 적절한 DDL 생성
- 이렇게 **생성된 DDL은 개발 장비에서만 사용**
- 생성된 DDL은 운영서버에서는 사용하지 않거나, 적절히 다듬은 후 사용

데이터베이스 스키마 자동 생성하기

- hibernate.hbm2ddl.auto
  - **create**: 기존테이블 삭제 후 다시 생성 (DROP + CREATE)
  - **create-drop**: create와 같으나 종료시점에 테이블 DROP
  - **update**: 변경분만 반영(운영DB에는 사용하면 안됨)
  - **validate**: 엔티티와 테이블이 정상 매핑되었는지만 확인
  - **none**: 사용하지 않음

데이터베이스 스키마 자동 생성하기 주의

- **운영 장비에는 절대 create, create-drop, update 사용하면 안된다.**
- 개발 초기 단계는 create 또는 update
- 테스트 서버는 update 또는 validate
- 스테이징과 운영 서버는 validate 또는 none

실습2

- 스키마 자동 생성하기 설정
- 스키마 자동생성하기 실행, 옵션별 확인

매핑 어노테이션

- @Column
- @Temporal
- @Enumerated
- @Lob
- @Transient

```java
@Entity
public class Member {

  @Id
  private Long id;

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

  private int age;

  @Temporal(TemporalType.TIMESTAMP)
  private Date regDate;

  @Enumerated(EnumType.STRING)
  private MemberType memberType;
  …
```

@Column

- 가장 많이 사용됨
- **name: 필드와 매핑할 테이블의 컬럼 이름**
- insertable, updatable: 읽기 전용
- nullable: null 허용여부 결정, DDL 생성시 사용
- unique: 유니크 제약 조건, DDL 생성시 사용
- columnDefinition, length, precision, scale (DDL)

@Temporal

- 날짜 타입 매핑

```
@Temporal(TemporalType.DATE)
private Date date; //날짜
```

```
@Temporal(TemporalType.TIME)
private Date time; //시간
```

```
@Temporal(TemporalType.TIMESTAMP)
private Date timestamp; //날짜와 시간
```

@Enumerated

- 열거형 매핑
- EnumType.ORDINAL: 순서를 저장(기본값)
- **EnumType.STRING: 열거형 이름을 그대로 저장, 가급적 이것을 사용**

```
@Enumerated(EnumType.STRING)
private RoleType roleType;
```

@Lob

- CLOB, BLOB 매핑
- CLOB : String, char[], java.sql.CLOB
- BLOB : byte[], java.sql.BLOB

```
@Lob
private String lobString;

@Lob
private byte[] lobByte;
```

@Transient

- 이 필드는 매핑하지 않는다.
- 애플리케이션에서 DB에 저장하지 않는 필드

실습2

- 다양한 매핑 어노테이션 추가
- 생성된 DDL 확인하기

식별자 매핑

예제 ex02

식별자 매핑 어노테이션

- @Id
- @GeneratedValue

```
@Id @GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

식별자 매핑 방법

- @Id(직접 매핑)
- **IDENTITY**: 데이터베이스에 위임, MYSQL
- **SEQUENCE**: 데이터베이스 시퀀스 오브젝트 사용, ORACLE
  - @SequenceGenerator 필요
- **TABLE**: 키 생성용 테이블 사용, 모든 DB에서 사용
  - @TableGenerator 필요
- **AUTO**: 방언에 따라 자동 지정, 기본값

권장하는 식별자 전략

- 기본 키 제약 조건: null 아님, 유일, **변하면 안된다.**
- 미래까지 이 조건을 만족하는 자연키는 찾기 어렵다. 대리키(대체키)를 사용하자.
- 예를 들어 주민등록번호도 기본 키로 적절하기 않다.
- **권장: Long + 대체키 + 키 생성전략 사용**

실습 예제2

- 다양한 식별자 매핑

연관관계 매핑

예제 ex03, ex04, ex05

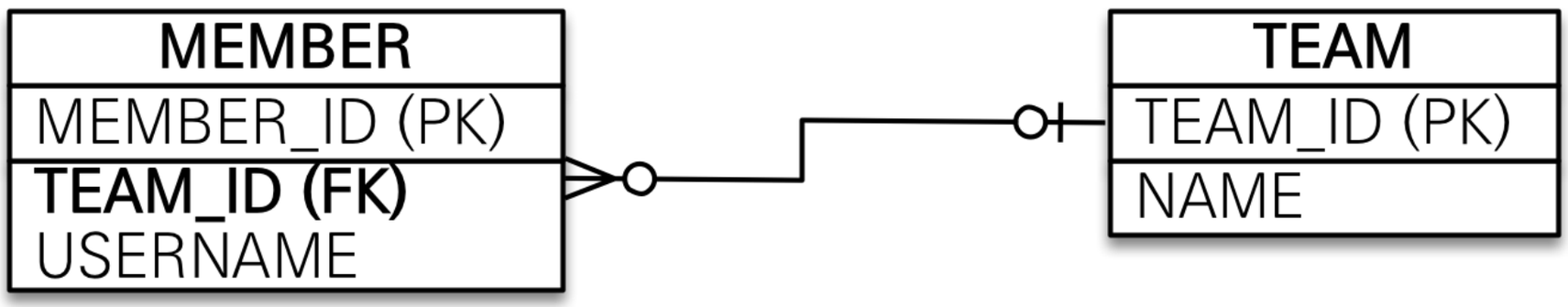
‘객체지향 설계의 목표는 자율적인 객체들의
협력 공동체를 만드는 것이다.’

–조영호(객체지향의 사실과 오해)

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;
  private int age;

  @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.

실습

- ex03, 테이블에 맞춘 엔티티 설계 및 실행

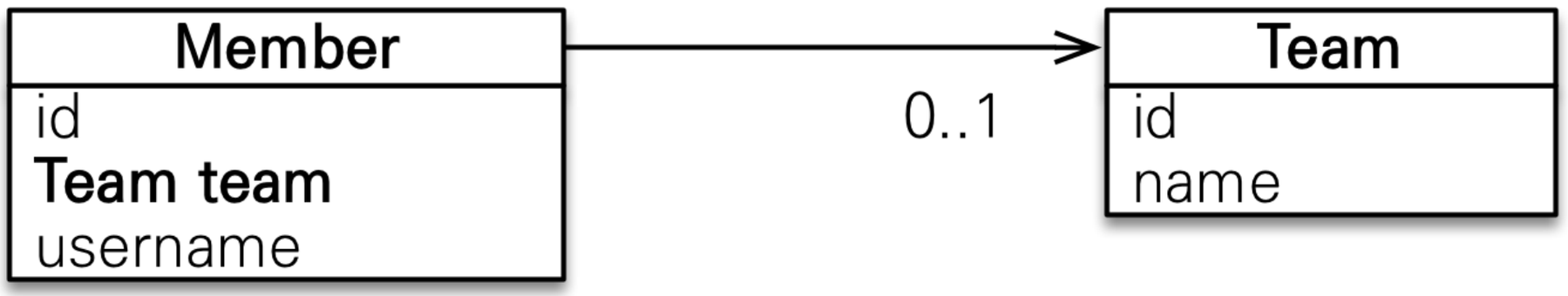
연관관계 매핑 이론

단방향 매핑

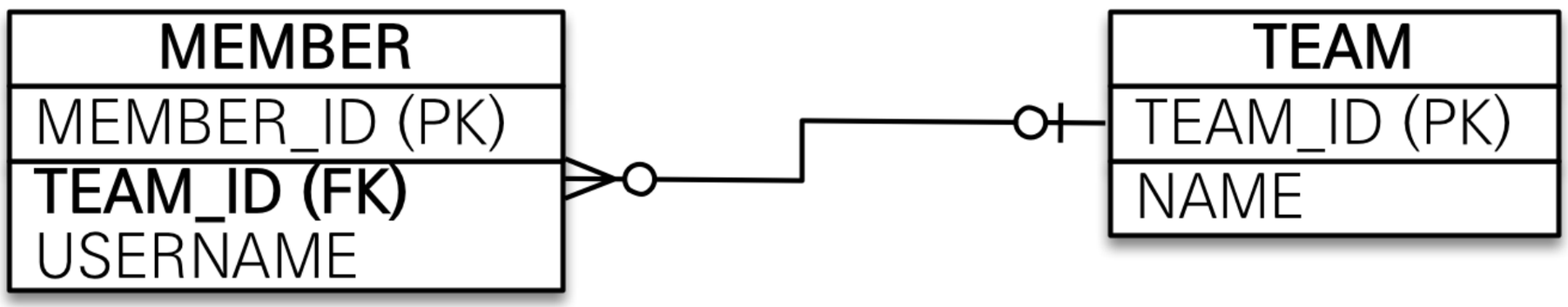
예제 ex04

객체 지향 모델링
(객체 연관관계 사용)

[객체 연관관계]



[테이블 연관관계]

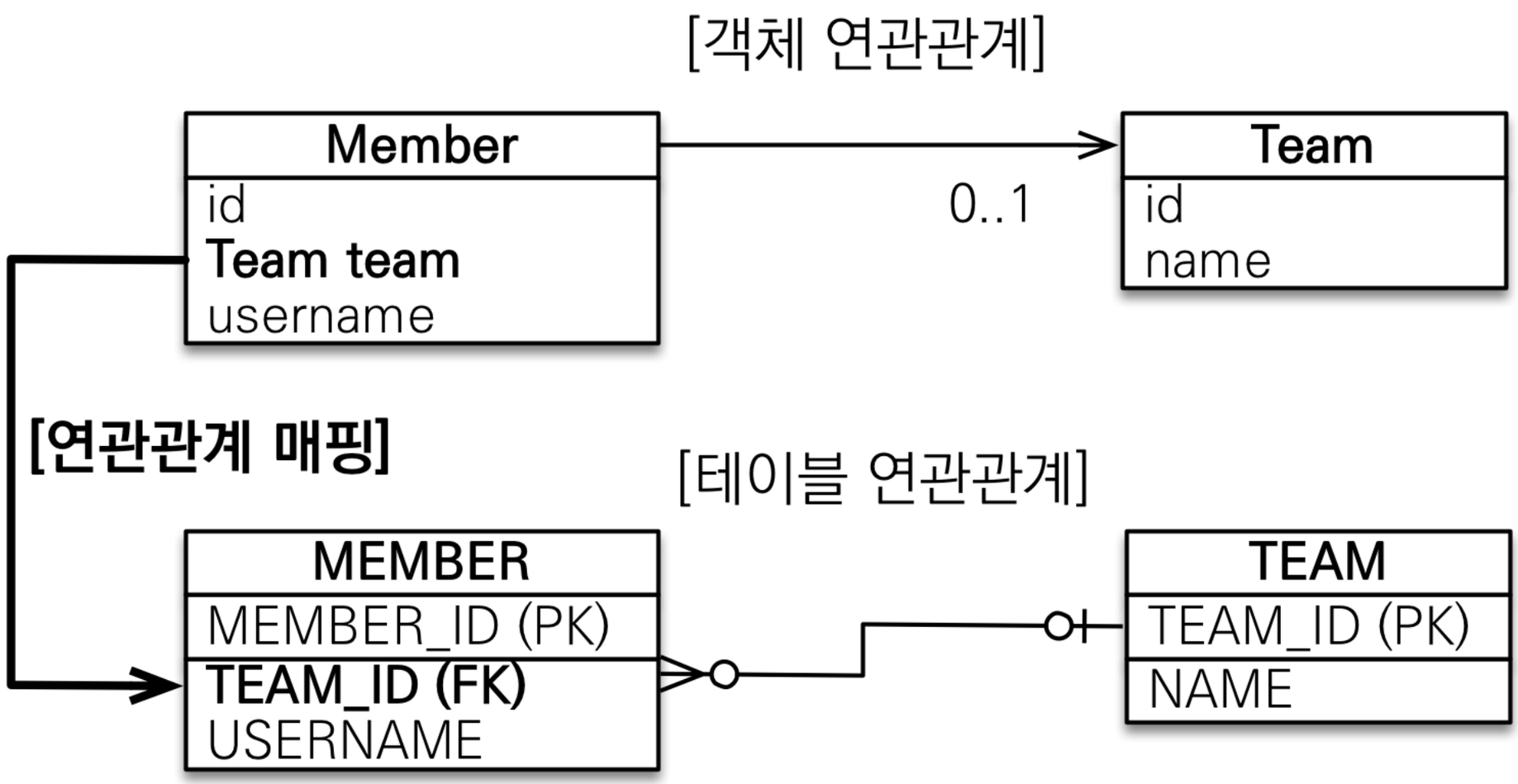


객체 지향 모델링

(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);
```

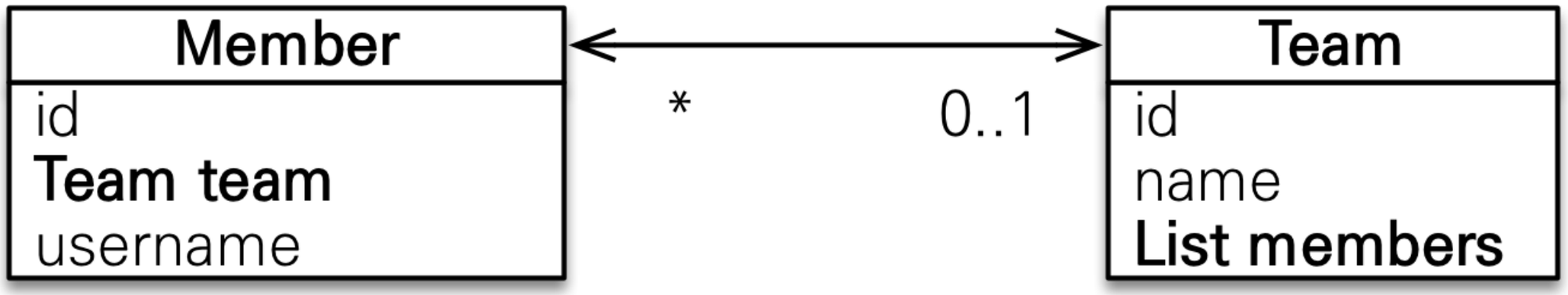
실습

- ex03의 외래 키 기반을 ex04와 같이 참조를 사용하도록 변경

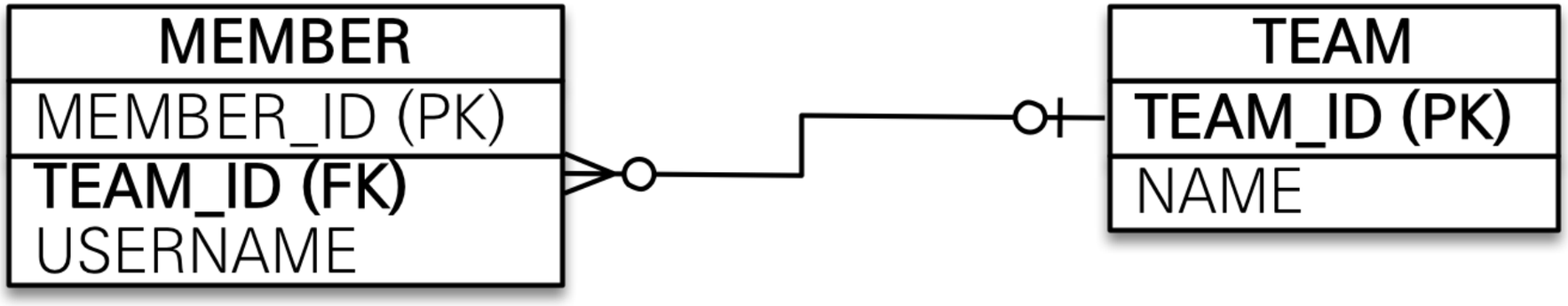
양방향 매핑

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

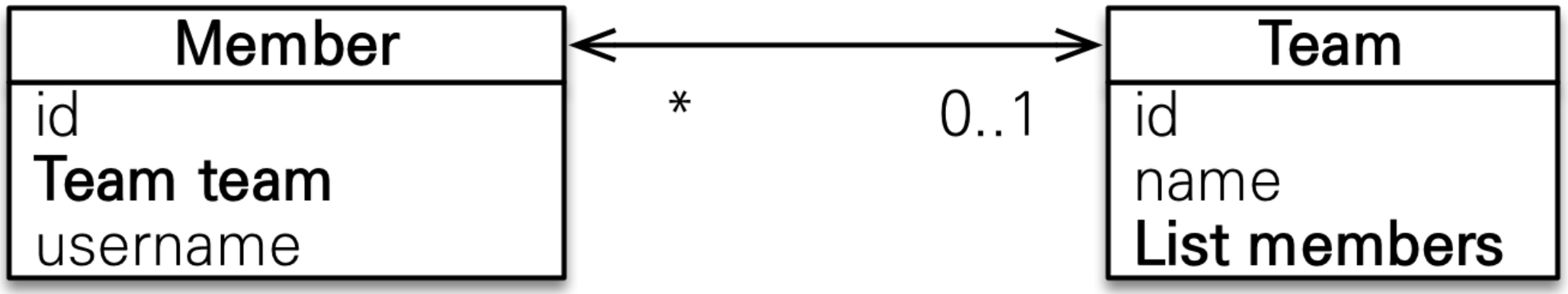
- mappedBy = JPA의 멘붕 클래스1
- mappedBy는 처음에는 이해하기 어렵다.
- 객체와 테이블간에 연관관계를 맺는 차이를 이해해야 한다.

객체와 테이블이 관계를 맺는 차이

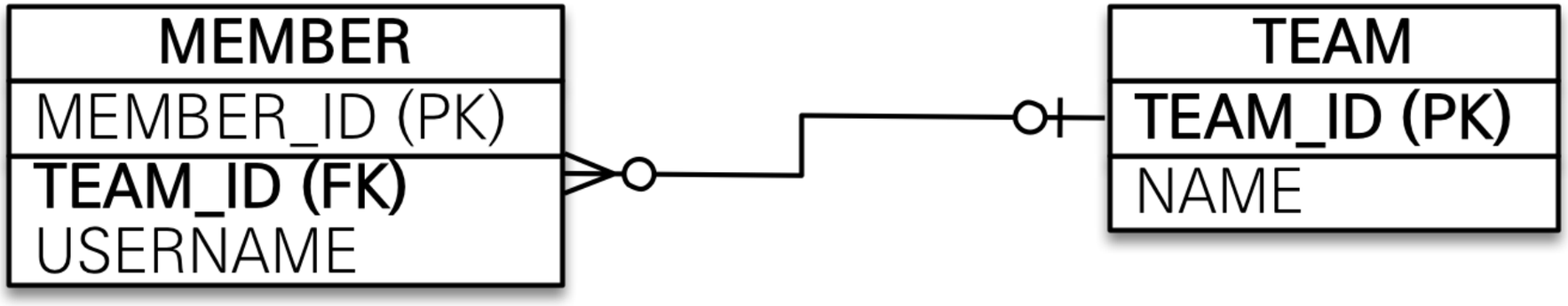
- **객체 연관관계**
    - 회원 -> 팀 연관관계 1개(단방향)
    - 팀 -> 회원 연관관계 1개(단방향)
- **테이블 연관관계**
    - 회원 <-> 팀의 연관관계 1개(양방향)

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;
}
```

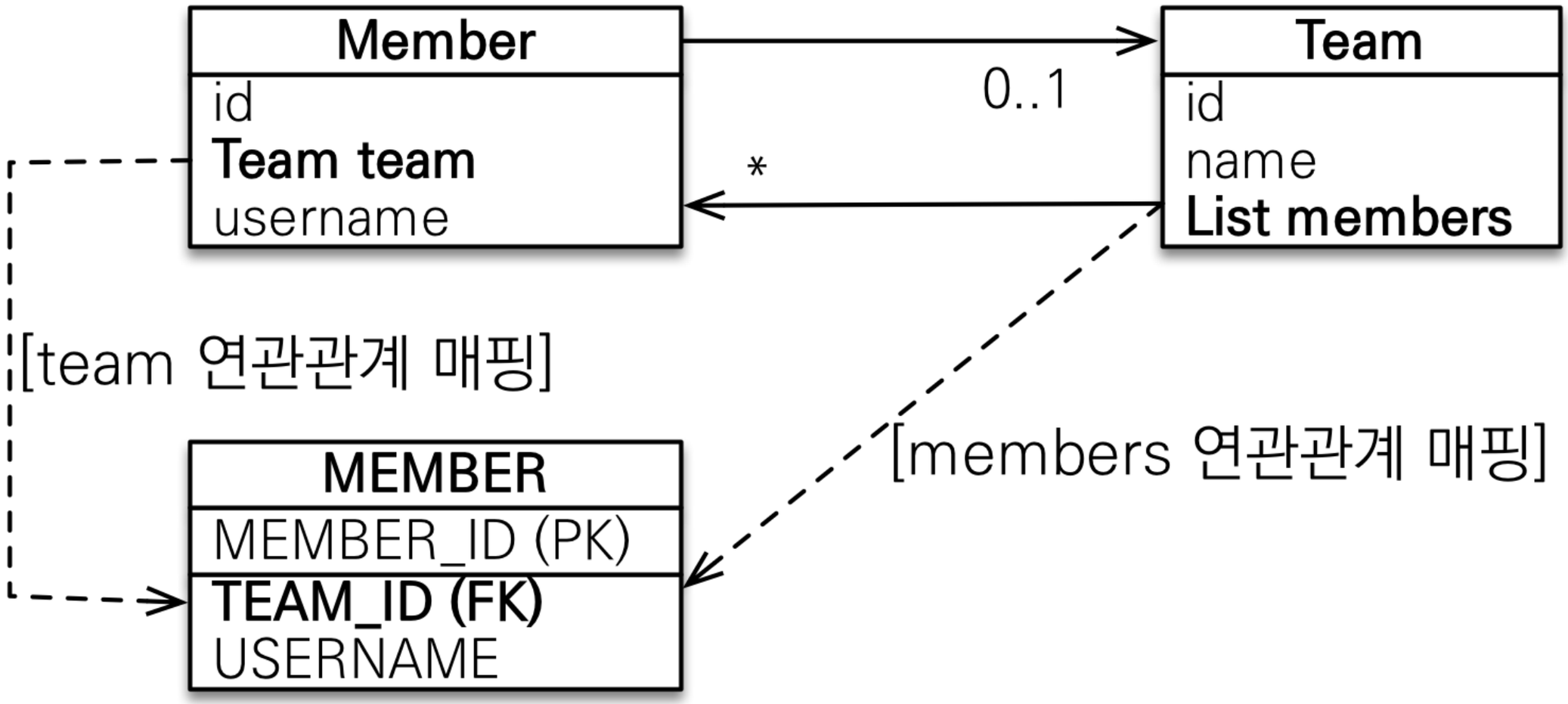
테이블의 양방향 연관관계

- 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.



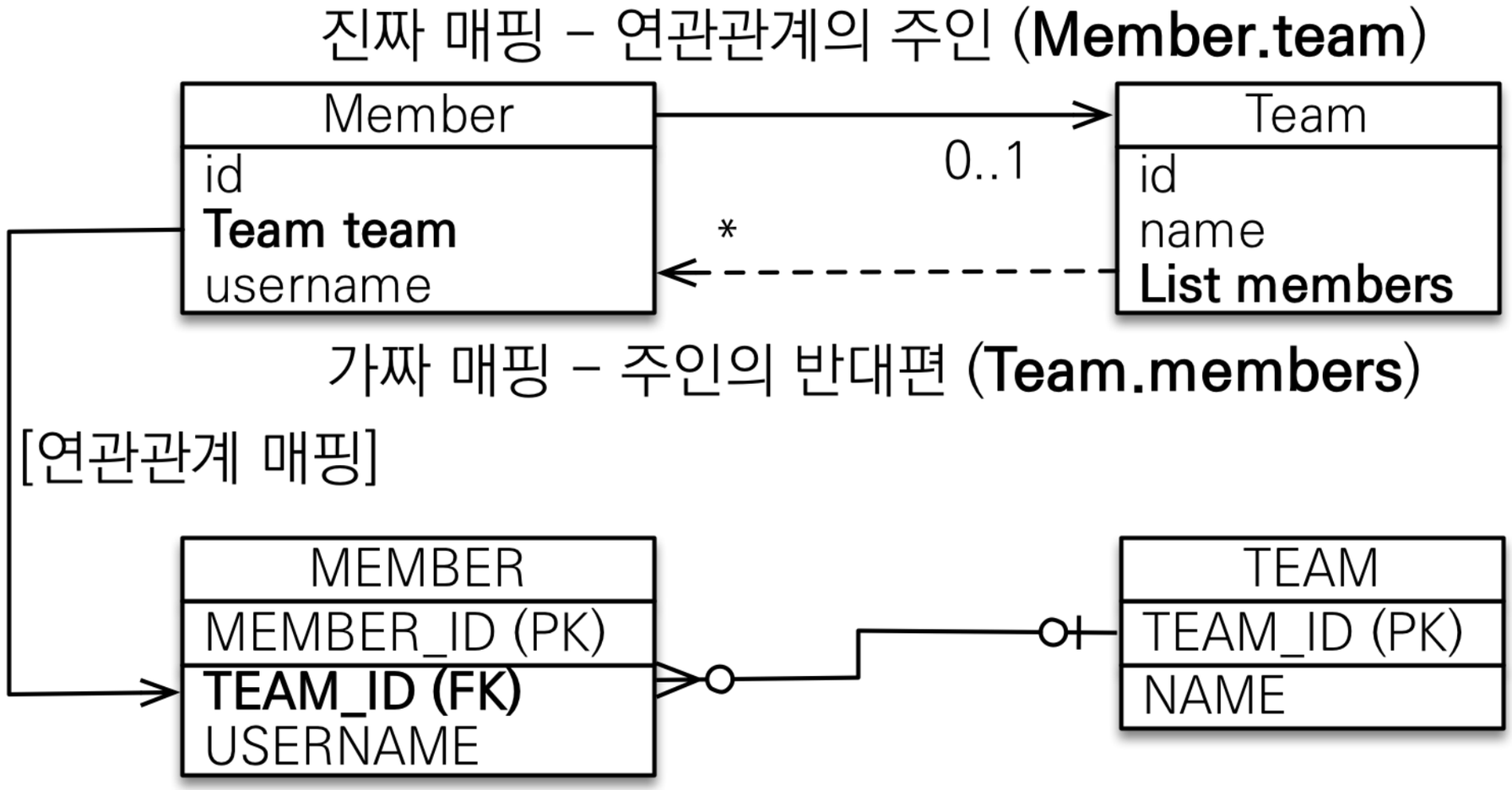
연관관계의 주인(Owner)

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 매핑의 장점

- **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)

실습5 - 양방향 매핑

- 반대방향 추가
- mappedBy

다양한 매핑 어노테이션 소개

연관관계 매핑 어노테이션

- 다대일 (@ManyToOne)
- 일대다 (@OneToMany)
- 일대일 (@OneToOne)
- 다대다 (@ManyToMany)
- @JoinColumn, @JoinTable

상속 관계 매핑 어노테이션

- @Inheritance
- @DiscriminatorColumn
- @DiscriminatorValue
- @MappedSuperclass(매핑 속성만 상속)

복합키 어노테이션

- @IdClass
- @EmbeddedId
- @Embeddable
- @MapsId