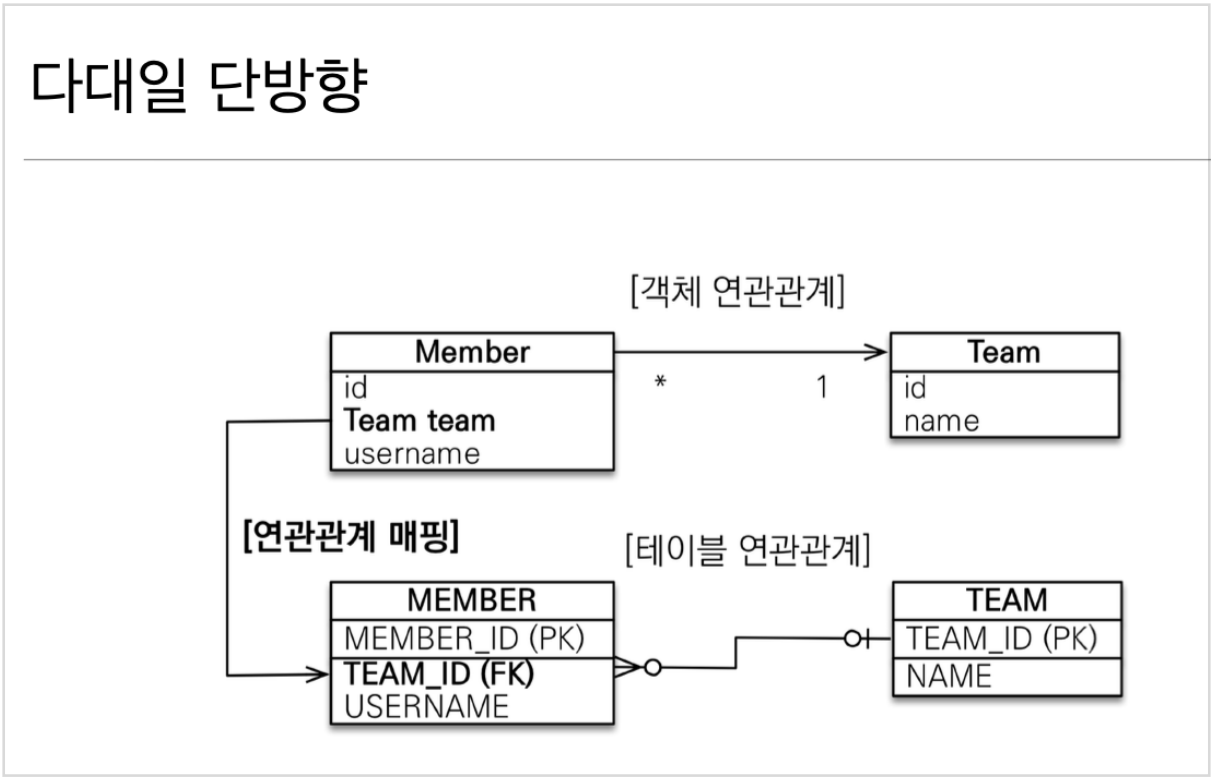
7-0. 다양한 연관관계 매핑

단방향, 양방향

- 테이블
  - 외래 키 하나로 양쪽 조인 가능
  - 사실 방향이라는 개념이 없다
- 객체
  - 참조용 필드가 있는 쪽으로만 참조 가능
    - 예를 들어서 멤버에서 팀을 가려면 팀에 대한 레퍼런스가, 팀에서 멤버를 가려면 멤버에 대한 레퍼런스가 있어야함.
    - 양쪽이 서로 참조하면 양방향인데, 양방향은 사실 없다!!!
      - 참조의 방향은 단 하나밖에 없으며, 단방향이 양쪽으로 들어가니까 양방향으로 보이는 것.
- 연관관계의 주인
  - 테이블은 외래 키 하나로 연관관계 맺기 가능
  - 객체 양방향 관계는 참조가 2군데
    - ➜ 둘 중 테이블의 외래 키를 관리할 곳은 어디로? 일대다라면 '다'쪽에!
    - 주인의 반대편에서는 단순 조회만 가능함.

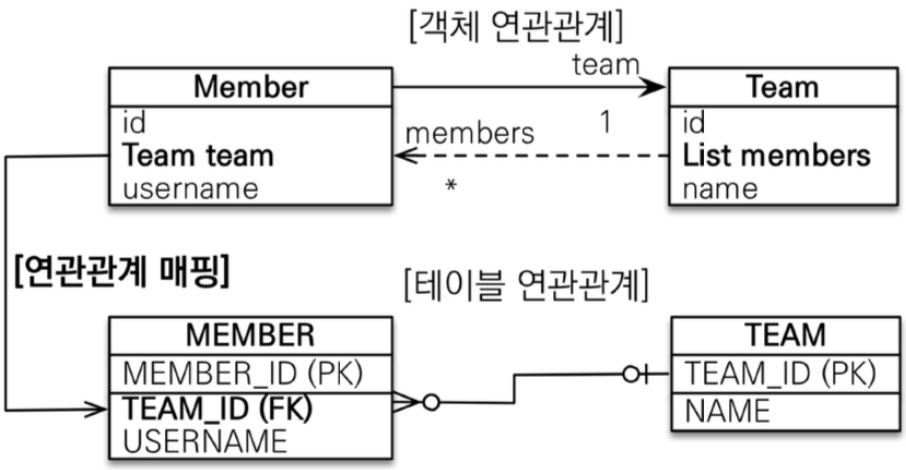
다대일 관계



다대일에서 '다'쪽에 외래키가 있어야함! 즉 연관관계의 주인이 '다'쪽임.

➜ 외래키가 있는 곳에 상대의 참조를 걸어놓고 연관관계 매핑을 하면 됨.

다대일 양방향



단방향인 상황에서 양방향으로 가도 테이블 연관관계는 영향이 없다.

꼭 넣어주어야 하는게 **mappedBy**!!

일대다 관계

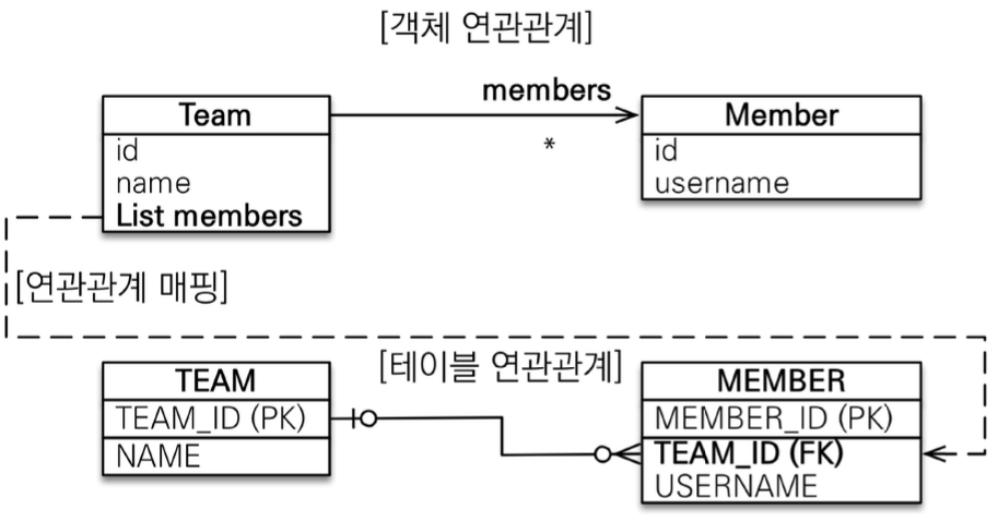
'일(1)'이 주인이 되겠다…

이 모델은 권장하지않음.

객체와 테이블의 차이 때문에 반대편 테이블의 외래 키를 관리하는 특이한 구조

`@JoinColumn` 을 무조건 사용해야함!! 그렇지 않으면 조인에 필요한 중간테이블이 생겨나버림..

일대다 단방향



팀은 멤버를 알고싶은데 멤버는 팀을 알고싶지 않은 상황.
팀을 중심으로 해보겠다. 팀에서 외래키도 관리함.
객체 입장에서 이런 설계가 나올 확률이 높다.
디비 설계 상 팀에는 외래키가 들어갈 수 없다.
디비 입장에서는 '다'쪽에 외래키가 들어가야한다!!!

➡ 팀에 외래키로 멤버 아이디가 있다면 팀을 계속 인서트해야함... 팀이 중복이되어버림
➡ 그냥 상식적으로 멤버에 팀 아이디가 있는 것이 매우 자연스러움.
➡ 팀이 중복이 된다는 건 멤버'들'이 팀에 있는건데팀에 외래키로 member_id가 있으면 팀을 계속 인서트 해야돼서 중복된다는거...

```
package hellojpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import java.util.List;

public class JpaMain {
  public static void main(String[] args) {
```

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

    EntityManager em = emf.createEntityManager();

    EntityTransaction tx = em.getTransaction();
    tx.begin();
    try {
      Member member = new Member();
      member.setName("member1");

      em.persist(member);

      Team team = new Team();
      team.setName("teamA");
      // 여기까지는 팀 테이블에 적용

      team.getMembers().add(member); //**
      // 여기가 좀 이상함..
      // 연관관계가 업데이트 된 것이므로 테이블 차원에서는 멤버의 외래키를 업뎃

      em.persist(team);

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

    emf.close();

  }

}
```

```java
package hellojpa;

import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
public class Team {

  @Id @GeneratedValue
  private Long id;
  private String name;
  @OneToMany // 팀의 입장에서는 일대다이기 때문에 OneToMany
  @JoinColumn(name = "TEAM_ID")
  private List<Member> members = new ArrayList<>();

  public void addMember(Member member) { // team->member 방향 편의 메소드
    member.setTeam(this);
    members.add(member);
  }

  public List<Member> getMembers() {
    return members;
  }

  public void setMembers(List<Member> members) {
    this.members = members;
  }

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public String getName() {
```

```
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

}
```

이렇게 하면 결과는 어떻게 나오냐?
결국 멤버 테이블의 외래키를 업데이트 해야하는 경우가 나옴.
어떻게 보면 매우 당연한데, 테이블에서는 외래키를 가지고 양방향을 넘나들 수 있어 '다' 쪽에서만 외래키를 가지며 연관관계를 맺기 때문이고 연관관계의 증거(?)라고 할 수 있는 것이 외래키밖에 없기 때문에 테이블 차원에서는 팀이 아닌 멤버의 외래키를 업데이트하는 것.
즉, 실생활에서 보면 팀 차원에서 멤버가 추가될 때 마다 개인의 정보(외래키)를 바꾸는 것으로 볼 수 있다.

```
        references Team
Hibernate:
    call next value for hibernate_sequence
Hibernate:
    call next value for hibernate_sequence
Hibernate:
    /* insert hellojpa.Member
        */ insert
        into
            Member
            (USERNAME, TEAM_ID, id)
        values
            (?, ?, ?)
Hibernate:
    /* insert hellojpa.Team
        */ insert
        into
            Team
            (name, id)
        values
            (?, ?)
Hibernate:
    /* create one-to-many row hellojpa.Team.members */ update
        Member
    set
        TEAM_ID=?
    where
        id=?
```

업데이트 쿼리가 추가로 나오는 이유??????

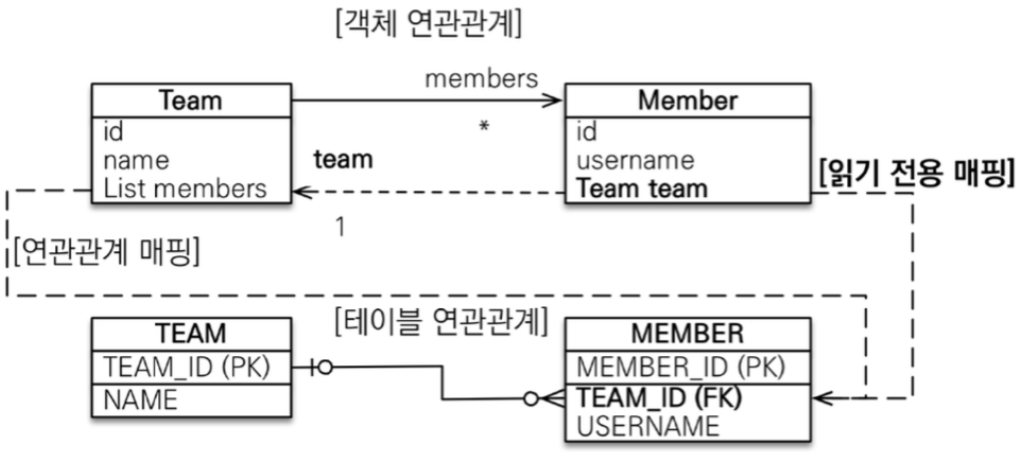
➡ 문제가 되는 부분에서 팀 엔티티를 저장하는데 옆 테이블에 있는 걸 업데이트 칠 수 밖에… 왜냐면 멤버와 연관관계 매핑이 멤버에 있는 외래키로 되어있기 때문.

➡ 진짜 심각한 이유는?

➡ 나는 '일' 쪽의 엔티티를 건드렸는데 쿼리를 살펴보니 '다' 쪽의 엔티티도 업데이트 때림… 이건 좀..

=> 결론은 일단 다대일로 단방향 매핑을 해놓은 다음 그 반대의 로직도 필요하게 된다면 그때 양방향 매핑을 추가하기!!

일대다 양방향




```
package hellojpa;

import org.hibernate.annotations.ManyToAny;

import javax.persistence.*;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

@Entity
public class Member {

  @Id @GeneratedValue
  private Long id;

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

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

  @ManyToOne // member 입장에서 team과는 다대일이므로 many to one
```

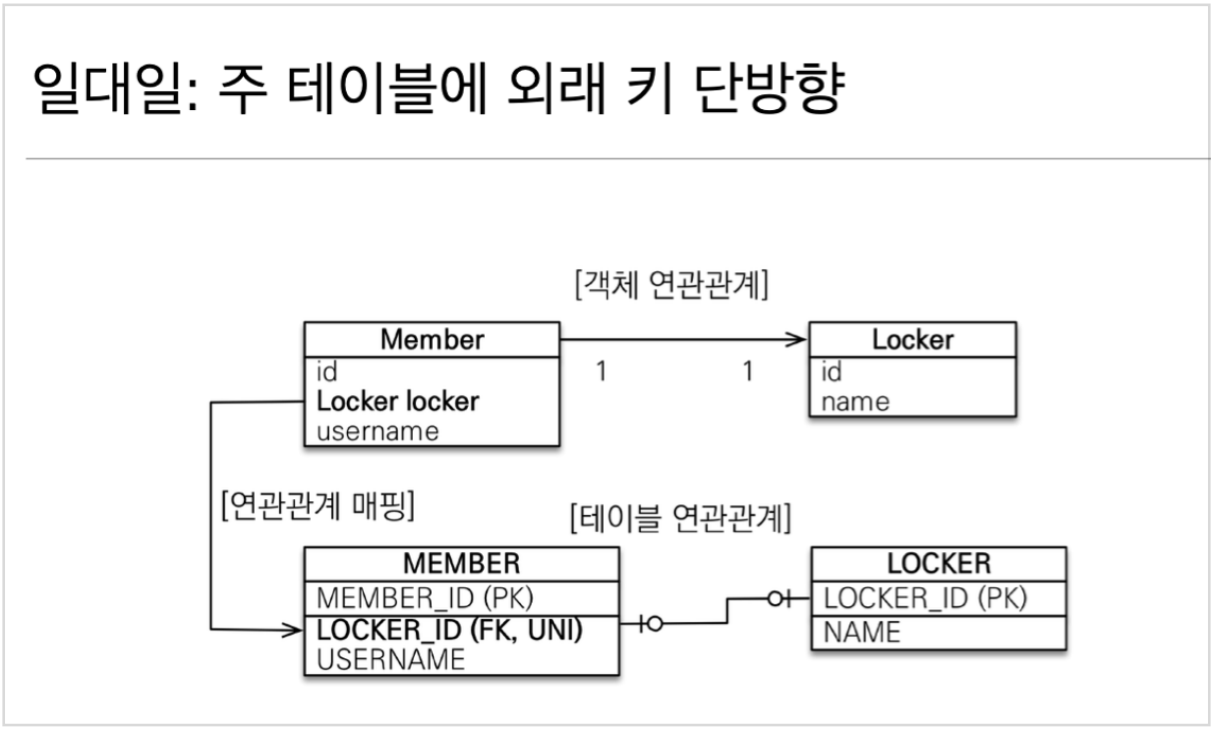
```
    @JoinColumn(name = "TEAM_ID", insertable = false, updatable = false ) // 관계
할 때 조인하는 외래키 컬럼의 이름 넣어주기
    private Team team;



    public Team getTeam() {
        return team;
    }

    public void setTeam(Team team) {
        this.team = team;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {

        this.name = name;
    }
}
```

insertable, updatable 옵션을 넣어서 insert, update를 막음!!
완전히 망해버리는걸 막을 수 있음
팀에 있는 List members가 매핑 주인 역할을 하고
멤버의 Team team도 주인 역할을 하다가 insertable, updatable 옵션을 넣어주면 읽기 전용으로 가면서 양방향으로 간다!

어쨌든 결론은 다대일 양방향을 사용하자. 설계라는게 테이블이 수십개씩 있으면 단순해야함 위 방향은 너무 복잡..

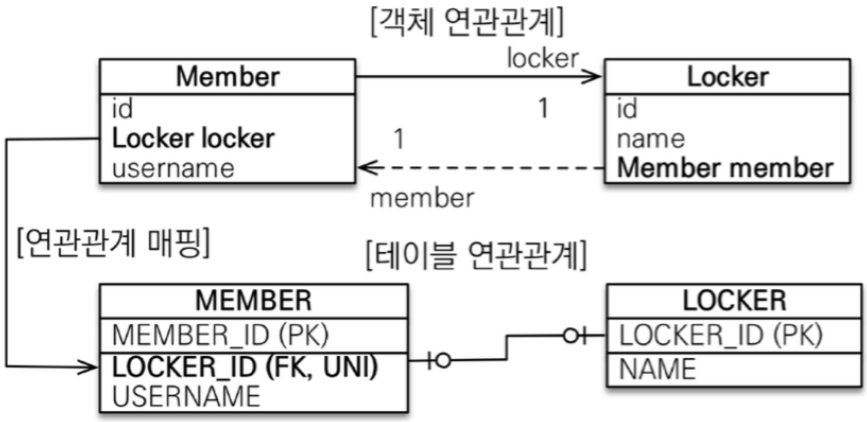
일대일 관계

일대일이라 외래 키를 아무데나 넣을 수 있음.
외래 키에 데이터베이스 유니크 제약조건을 추가해야함.



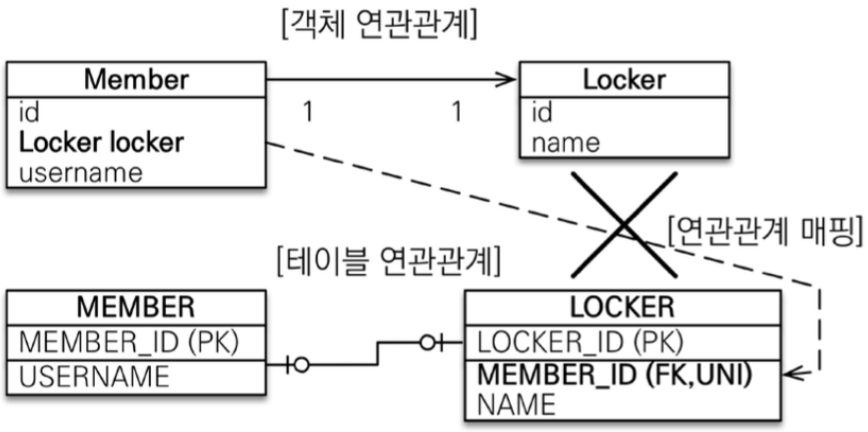
위 그림처럼 member에 락커아이디를 넣어줘도 되지만 반대로 locker에 memberid를 넣어줘도 된다.
마치 다대일 단방향 매핑과 비슷함.

일대일: 주 테이블에 외래 키 양방향



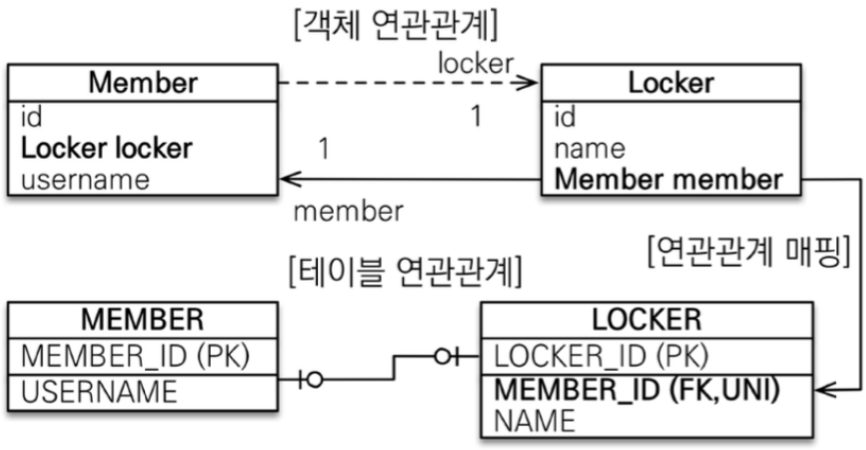
간단한 일대일 양방향 매핑. 어느 한 쪽에는 무조건 mappedBy를 넣어줘야한다.

일대일: 대상 테이블에 외래 키 단방향



일대다 단방향처럼 외래키가 연관관계의 주인이 아닌 곳에 있음..

일대일: 대상 테이블에 외래 키 양방향



결론은 내가 내것만 관리할 수 있음. 내 안에 있는 외래키만 관리할 수 있다!
내 엔티티의 외래키는 내가 관리함. locker를 양방향의 주인으로 잡고 member는 조회 기능만…

주 테이블에 외래 키 vs 대상 테이블에 외래 키

만약에 나중에 하나의 회원이 여러개의 라커를 가질 수 있으면 대상 테이블에 있는 유니크 조건만 빼면 일대다로 바꿀 수있다.
하지만 주 테이블에 외래키가있다면 대상 테이블에 대해 많이 바꿔야함.
=> 비즈니스 로직을 보고 하는게 좋음.

하지만 멤버가 라커를 가지고 있는게 훨씬 개발자한텐 유리함.
대부분의 비즈니스에서는 멤버를 조회하는게 훨씬 많은데, 이 때 쿼리 한방으로 멤버가 갖고있는 라커를 다 볼 수 있다. 명확하게 일대일이면 그냥 개발자한테 유리하게 하자..

만약에 외래키가 대상 테이블에 있으면 그냥 양방향 걸자!!

두 개의 비교는 강의록!

일대일 정리

- **주 테이블에 외래 키**
  - 주 객체가 대상 객체의 참조를 가지는 것 처럼
    주 테이블에 외래 키를 두고 대상 테이블을 찾음
  - 객체지향 개발자 선호
  - JPA 매핑 편리
  - 장점: 주 테이블만 조회해도 대상 테이블에 데이터가 있는지 확인 가능
  - 단점: 값이 없으면 외래 키에 null 허용
- **대상 테이블에 외래 키**
  - 대상 테이블에 외래 키가 존재
  - 전통적인 데이터베이스 개발자 선호
  - 장점: 주 테이블과 대상 테이블을 일대일에서 일대다 관계로 변경할 때 테이블 구조 유지
  - 단점: 프록시 기능의 한계로 **지연 로딩으로 설정해도 항상 즉시 로딩됨**(프록시는 뒤에서 설명)

대상 테이블에 외래 키가 있을 때 예시)
프록시 객체를 만들려면 어떤 객체가 null인지를 확인해야하는데, 대상 테이블에 외래 키가 있으면 락커의 값이 null인지 판단하려면 락커에 내 멤버 아이디가 있는지 뒤져야함. 어쨌든 락커를 뒤져야함!!
=> 어차피 쿼리를 던져야 하니까 굳이 프록시에 넣지 않음..! 그래서 지연 로딩 설정이 의미가 없음…
=> 실제로는 일대일 정리에서는 주 테이블에 외래 키를 쓴다.

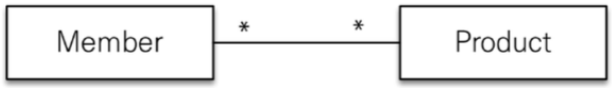
다대다 관계

실무에선 안쓰는게 낫다.. 편안하게 듣자!
RDB에서는 정규화된 테이블 2개로 다대다 관계를 표현할 수 없다….
그래서 orderitem마냥 관계 테이블을 만들어야 한다. 디비 시간에 배웠던거같은 관계테이블,…
=> 다대다 관계 ➡ 다대일 + 일대다

- 객체는 컬렉션을 사용해서 객체 2개로 다대다 관계 가능.

다대다

- 객체는 컬렉션을 사용해서 객체 2개로 다대다 관계 가능




```java
package hellojpa;

import org.hibernate.annotations.ManyToAny;

import javax.persistence.*;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

@Entity
public class Member {

  @Id @GeneratedValue
  private Long id;

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

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

```
@ManyToOne// member 입장에서 team과는 다대일이므로 many to one
@JoinColumn(name = "TEAM_ID")
private Team team;

@OneToOne
@JoinColumn(name = "LOCKER_ID")
private Locker locker;

@ManyToMany
@JoinTable(name = "MEMBER_PRODUCT")
private List<Product> products = new ArrayList<>();

public Team getTeam() {
  return team;
}

public void setTeam(Team team) {
  this.team = team;
}

public Long getId() {
  return id;
}

public void setId(Long id) {
  this.id = id;
}

public String getName() {
  return name;
}

public void setName(String name) {
  this.name = name;
}

public void changeTeam(Team team) {
  this.team = team;
```

```
    team.getMembers().add(this);
  }
}
```

```
package hellojpa;

import org.w3c.dom.ls.LSInput;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import java.util.ArrayList;
import java.util.List;

@Entity
public class Product {

  @Id @GeneratedValue
  private Long id;

  private String name;

//  @ManyToMany(mappedBy = "products")
//  private List<Member> members = new ArrayList<>();

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public String getName() {
```

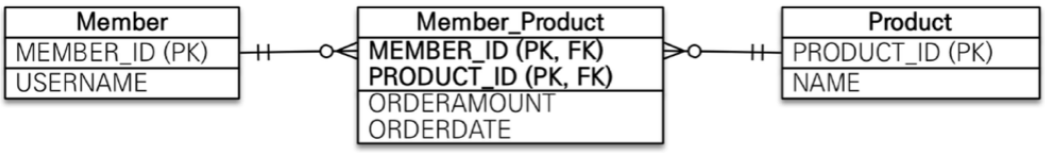
```
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

이건 단방향. product의 주석을 풀면 양방향.

근데 이건 실무에서는 사용하지 않음.

다대다 매핑의 한계

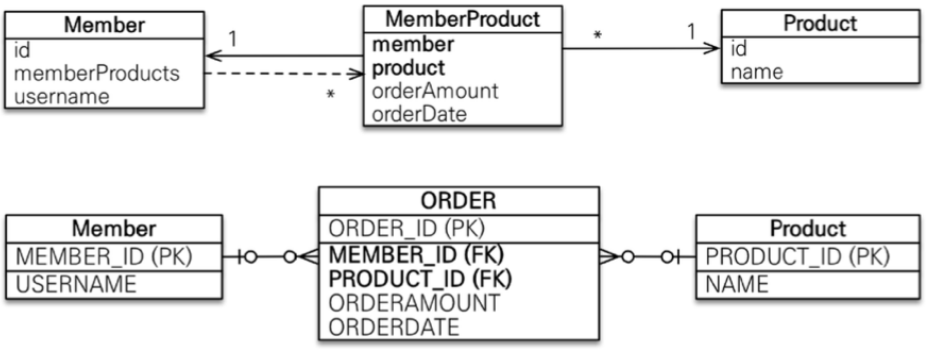
- **편리해 보이지만 실무에서 사용X**
- 연결 테이블이 단순히 연결만 하고 끝나지 않음
- 주문시간, 수량 같은 데이터가 들어올 수 있음



위 그림에서처럼 멤버와 프로덕트의 pk를 묶어서 composite id로 쓸지
아래 그림에서처럼 의미없는 값(order_id)을 주고 각자의 pk를 fk로 쓸지
이건 고민해보자. 영한이형은 후자를 쓴다. 일반적으로는 실무에서 ID라는게 어딘가에 종속적으로 걸리게되면 유연하게 변경이 쉽지 않다.

다대다 한계 극복

- **연결 테이블용 엔티티 추가(연결 테이블을 엔티티로 승격)**
- **@ManyToMany -> @OneToMany, @ManyToOne**



결론은 단순하게 ManyToMany를 써서 풀 수 있는건 없다.

그래서 연결 테이블용 **엔티티 추가**를 써라!!

#JPA