- **연관관계 매핑 - 이론**
  - **단방향 연관관계**
    - - 순수한 객체 연관관계
    - - 테이블 연관관계
    - - 객체 관계 매핑하기
    - - @JoinColumn
    - - @ManyToOne
  - **연관관계 사용하기**
    - - 저장
    - - 조회
    - - 수정
    - - 연관관계 제거
    - - 연관된 엔티티 삭제
  - **양방향 연관관계**
    - - 양방향 연관관계 매핑하기
    - - 일대다 컬렉션 조회
  - **연관관계의 주인(Owner)**
    - - 양방향 매핑의 규칙 - 연관관계의 주인(Owner)
    - - 연관관계의 주인은 외래 키가 있는 곳
  - **양방향 연관관계 저장하기**
  - **양방향 연관관계의 주의점**
    - - 순수한 객체까지 고려한 양방향 연관관계

        - - 연관관계 편의 메서드
        - - 연관관계 편의 메서드 작성시 주의사항
    - **양방향 매핑 정리**
- **[실전 예제] - 2. 연관관계 매핑 시작하기**
    - **일대다, 다대일 연관관계 매핑하기**
    - **객체 그래프 탐색**

연관관계 매핑 - 이론

비즈니스 엔티티들은 대부분 다른 엔티티와 연관관계가 있다. 예를 들어 주문 엔티티는 어떤 상품을 주문했는지 알기 위해 상품 엔티티와 연관관계가 있고 상품 엔티티는 카테고리, 재고 등 또 다른 엔티티와 관계가 있다. 그런데 객체는 참조(주소)를 사용해서 관계를 맺고 테이블은 외래 키를 사용해서 관계를 맺는다. 이 둘은 완전히 다른 특징을 가진다. 객체 관계 매핑(ORM)에서 가장 어려운 부분이 바로 객체 연관관계와 테이블 연관관계를 매핑하는 일이다.

객체의 참조와 테이블의 외래 키를 매핑하는 것이 이 장의 목표다.

시작하기 전에 연관관계 매핑을 이해하기 위한 핵심 키워드를 정리해보았다. 진행하면서 하나씩 이해해보자.

- **방향(Direction)**: [단방향, 양방향]이 있다. 예를 들어 회원과 팀이 관계가 있을 때 `회원 -> 팀` 또는 `팀 -> 회원` 둘 중 한쪽만 참조하는 것을 단방향 관계라 하고, `회원 -> 팀` , `팀 -> 회원` 양쪽 모두 서로 참조하는 것을 양방향 관계라 한다. 방향은 객체관계에만 존재하고 테이블 관계는 항상 양방향이다.
- **다중성(Multiplicity)**: [다대일(N:1), 일대다(1:N), 일대일(1:1), 다대다(N:M)] 다중성이 있다. 예를 들어 회원과 팀이 관계가 있을 때 여러 회원은 한 팀에 속하므로 회원과 팀은 다대일 관계다. 반대로 한 팀에 여러 회원이 소속될 수 있으므로 팀과 회원은 일대다 관계다.
- **연관관계의 주인(owner)**: 객체를 양방향 연관관계로 만들면 연관관계의 주인을 정해야 한다.

단방향 연관관계

연관관계 중에선 다대일(N:1) 단방향 관계를 가장 먼저 이해해야 한다.

지금부터 회원과 팀의 관계를 통해 다대일 단방향 관계를 알아보자.

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

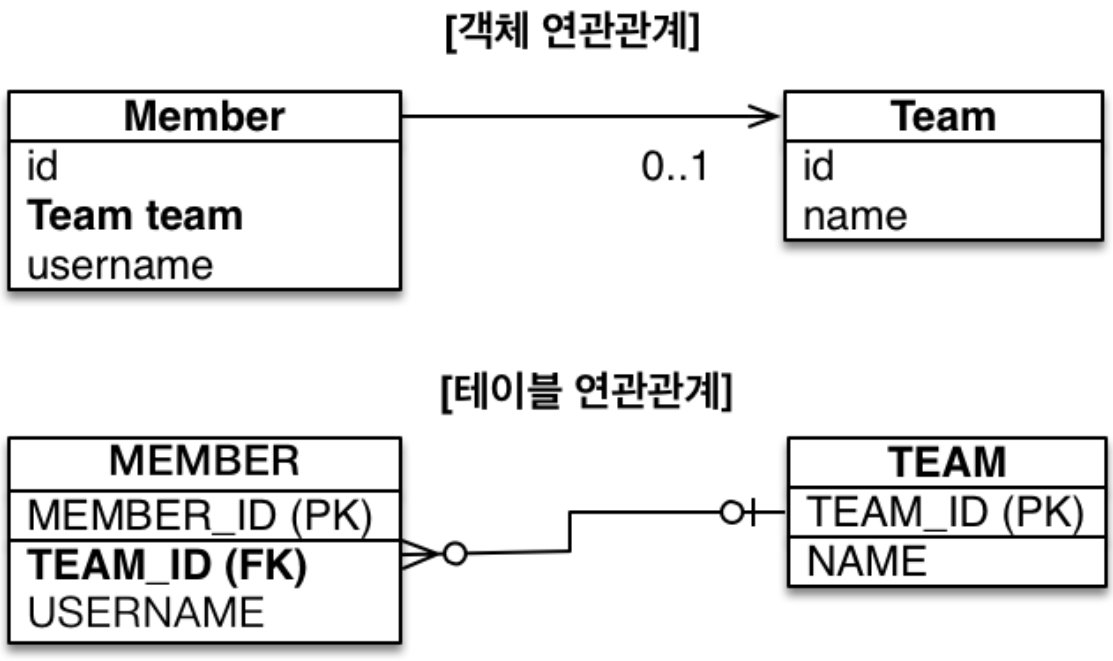

그림 - 다대일 연관관계 I 다대일(N:1), 단방향

그림을 분석해 보자.

객체 연관관계

- 회원 객체는 `Member.team` 필드(멤버변수)로 팀 객체와 연관관계를 맺는다.
- 회원 객체와 팀 객체는 **단방향 관계**다. 회원은 `Member.team` 필드를 통해서 팀을 알 수 있지만 반대로 팀은 회원을 알 수 없다. 예를 들어 `member -> team`의 조회는 `member.getTeam()`으로 가능하지만, 반대 방향인 `team -> member`를 접근하는 필드는 없다.

테이블 연관관계

- 회원 테이블은 `TEAM_ID` 외래 키로 팀 테이블과 연관관계를 맺는다.
- 회원 테이블과 팀 테이블은 **양방향 관계**다. 회원 테이블의 `TEAM_ID` 외래 키를 통해서 회원과 팀을 조인할 수 있고 반대로 팀과 회원도 조인할 수 있다. 예를 들어 `MEMBER` 테이블의 `TEAM_ID` 외래 키 하나로 `MEMBER JOIN TEAM` 과 `TEAM JOIN MEMBER`가 모두 가능하다.

====== 회원과 팀 조인 ======

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

====== 팀과 회원 조인 ======

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

객체 연관관계와 테이블 연관관계의 가장 큰 차이

참조를 통한 연관관계는 언제나 단방향이다. 객체간에 연관관계를 양방향으로 만들고 싶으면 반대쪽에도 필드를 추가해서 참조를 보관해야 한다. 결국 연관관계를 하나 더 만들어야 한다. 이렇게 양쪽에서 서로 참조하는 것을 양방향 연관관계라 한다. 하지만 정확히 이야기하면 이것은 **양방향 관계가 아니라 서로 다른 단뱡향 관계 2개다.** 반면에 테이블은 외래 키 하나로 양방향으로 조인할 수 있다.

====== 단방향 연관관계 ======

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

====== 양방향 연관관계 ======

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

객체 연관관계 vs 테이블 연관관계

- 객체는 참조(주소)로 연관관계를 맺는다.
- 테이블은 외래 키로 연관관계를 맺는다.

이 둘은 비슷해 보이지만 매우 다른 특징을 가진다.

- 연관된 데이터를 조회할 때 객체는 참조(`a.getB().getC()`)를 사용하지만, 테이블은 조인(JOIN)을 사용한다.
- 참조를 사용하는 객체의 연관관계는 단방향이다.
  - A -> B (a.b)

- 외래 키를 사용하는 테이블의 연관관계는 양방향이다.
  - A JOIN B 가 가능하면 반대로 B JOIN A 도 가능하다.

- 객체를 양방향으로 참조하려면 단방향 연관관계를 2개 만들어야 한다.
  - A -> B (a.b)
  - B -> A (b.a)

지금까지 객체 연관관계와 테이블 연관관계의 차이점을 알아보았다. 이제 순수한 객체 연관관계 예제와 순수한 테이블 연관관계 예제를 보고 둘을 매핑해보자.

- 순수한 객체 연관관계

순수하게 객체만 사용한 연관관계를 살펴보자.

====== 회원 클래스 ======

```
public class Member {

    private String id;
    private String username;

    private Team team; //팀의 참조를 보관

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

    //Getter, Setter ...
}
```

====== 팀 클래스 ======

```
public class Team {

    private String id;
    private String name;

    //Getter, Setter ...
}
```

회원1과 회원2를 팀1에 소속시키자.

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

    //생성자(id, 이름)
    Member member1 = new Member("member1","회원1");
    Member member2 = new Member("member2","회원2");
    Team team1 = new Team("team1","팀1");

    member1.setTeam(team1);
    member2.setTeam(team1);

    Team findTeam = member1.getTeam();
}
```

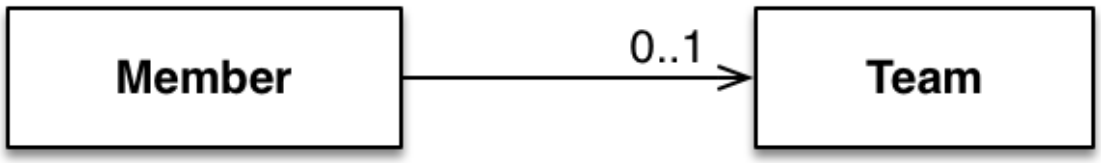

그림 - 순수객체단방향1 I 단방향 다대일(N:1) 클래스

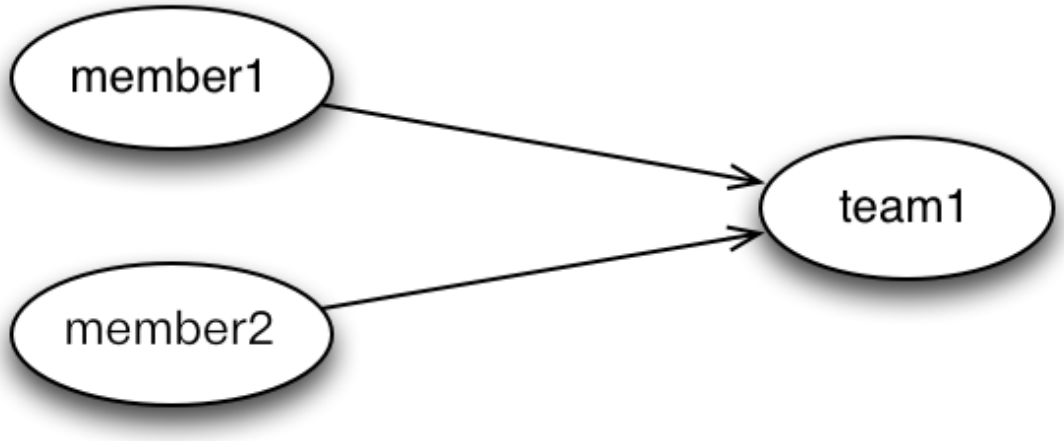

그림 - 순수객체단방향2 I 단방향 다대일(N:1) 인스턴스

회원1과 회원2는 팀1에 소속했다. 그리고 다음 코드로 회원1이 속한 팀1을 조회했다.

```
Team findTeam = member1.getTeam();
```

이처럼 객체는 참조를 사용해서 연관관계를 탐색할 수 있는데 이것을 **객체 그래프 탐색**이라 한다.

- 테이블 연관관계

이번에는 데이터베이스 테이블의 회원과 팀의 관계를 살펴보자.

다음은 회원 테이블과 팀 테이블의 DDL이다. 추가로 회원 테이블의 `TEAM_ID` 에 외래 키 제약조건을 설정했다.

====== 회원 테이블 ======

```
CREATE TABLE MEMBER (
    MEMBER_ID VARCHAR(255) NOT NULL,
    TEAM_ID VARCHAR(255),
    USERNAME VARCHAR(255),
    PRIMARY KEY (MEMBER_ID)
)
```

====== 팀 테이블 ======

```
CREATE TABLE TEAM (
```

```
    TEAM_ID VARCHAR(255) NOT NULL,
    NAME VARCHAR(255),
    PRIMARY KEY (TEAM_ID)
)
```

====== 외래 키 제약조건 ======

```
ALTER TABLE MEMBER ADD CONSTRAINT FK_MEMBER_TEAM
    FOREIGN KEY (TEAM_ID)
    REFERENCES TEAM
```

회원1과 회원2를 팀1에 소속시키자.

```
INSERT INTO TEAM(TEAM_ID, NAME) VALUES('team1','팀1');
INSERT INTO MEMBER(MEMBER_ID, TEAM_ID, USERNAME) VALUES('member1','team1','회원1'
INSERT INTO MEMBER(MEMBER_ID, TEAM_ID, USERNAME) VALUES('member2','team1','회원2'
```

이제 회원1이 소속된 팀을 조회해보자.

```
SELECT T.*
FROM MEMBER M
    JOIN TEAM T ON M.TEAM_ID = T.ID
WHERE M.MEMBER_ID = 'member1'
```

이처럼 데이터베이스는 외래 키를 사용해서 연관관계를 탐색할 수 있는데 이것을 **조인**이라 한다.

- 객체 관계 매핑하기

지금까지 객체만 사용한 연관관계와 테이블만 사용한 연관관계를 각각 알아보았다. 이제 JPA를 사용해서 둘을 매핑해보자.

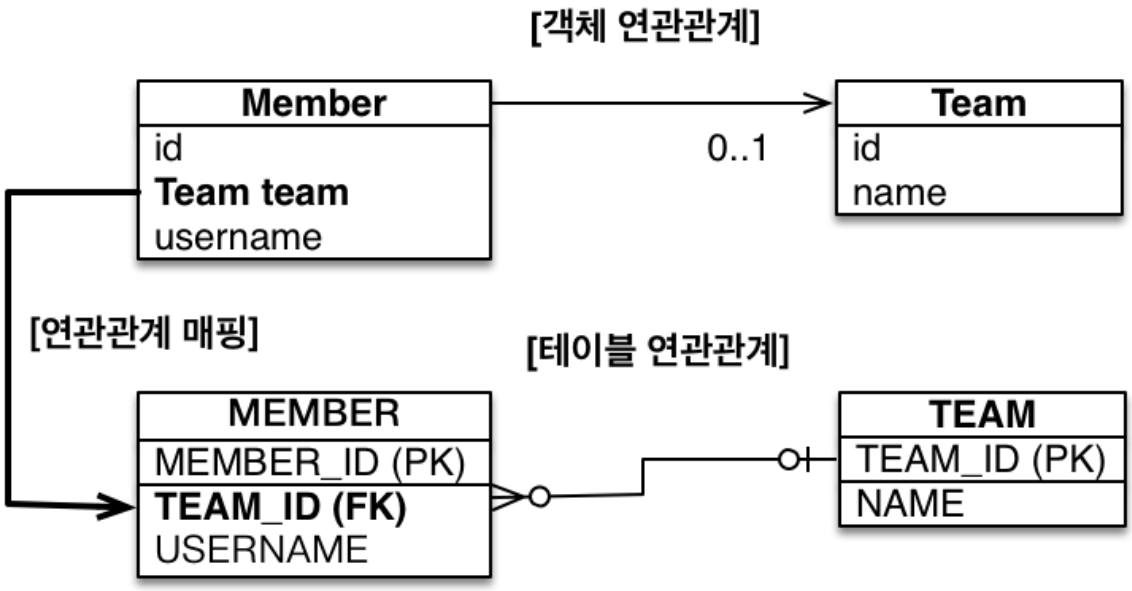

그림 - 다대일 연관관계1 I 다대일(N:1), 단방향

====== 매핑한 회원 엔티티 ======

```
@Entity
public class Member {

    @Id
    @Column(name = "MEMBER_ID")
    private String id;

    private String username;

    //연관관계 매핑
    @ManyToOne                              //**
    @JoinColumn(name="TEAM_ID")             //**
    private Team team;

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

    //Getter, Setter ...
}
```

====== 매핑한 팀 엔티티 ======

```
@Entity
```

```
public class Team {

    @Id
    @Column(name = "TEAM_ID")
    private String id;

    private String name;

    //Getter, Setter ...
}
```

회원과 팀 엔티티의 매핑을 완료했다. 코드를 분석하기 전에 먼저 [그림 - 다대일 연관관계1]에서 [연관관계 매핑] 부분을 보자.

- 객체 연관관계: 회원 객체의 `Member.team` 필드 사용
- 테이블 연관관계: 회원 테이블의 `MEMBER.TEAM_ID` 외래 키 컬럼을 사용

`Member.team`과 `MEMBER.TEAM_ID`를 매핑하는 것이 연관관계 매핑이다. 연관관계 매핑 코드를 분석해보자.

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

회원 엔티티에 있는 연관관계 매핑 부분인데 연관관계를 매핑하기 위한 새로운 어노테이션들이 있다.

- `@ManyToOne` : 이름 그대로 다대일(N:1) 관계라는 매핑정보다. 회원과 팀은 다대일 관계다. 연관관계를 매핑할 때 이렇게 다중성을 나타내는 어노테이션은 필수로 사용해야 한다.
- `@JoinColumn(name="TEAM_ID")` : 조인 컬럼은 외래 키를 매핑할 때 사용한다. `name` 속성에는 매핑할 외래 키 이름을 지정한다. 회원과 팀 테이블은 `TEAM_ID` 외래 키로 연관관계를 맺으므로 이 값을 지정하면 된다. 이 어노테이션은 생략할 수 있다.

연관관계 매핑 어노테이션을 자세히 알아보자.

- @JoinColumn

외래 키를 매핑할 때 사용한다.

표 - `@JoinColumn`의 주요 속성

속성	기능	기본값
`name`	매핑할 외래 키 이름	필드명 + `_` + 참조하는 테이블의 기본 키 컬럼명
`referencedColumnName`	외래 키가 참조하는 대상 테이블의 컬럼	참조하는 테이블의 기본 키 컬럼을 이름으로 사용
`foreignKey` (DDL)	외래 키 제약조건을 직접 지정할 수 있다. 이 속성은 테이블을 생성할 때만 사용한다.	

`unique`
`nullable`
`insertable`
`updatable`
`columnDefinition`
`table`
	`@Column`의 속성과 같다.	

> **참고:** `@JoinColumn` 생략
>
> ```
> @ManyToOne
> private Team team;
> ```
>
> 이처럼 `@JoinColumn`을 생략하면 외래 키를 찾을 때 기본 전략을 사용한다.
>
> - 기본 전략 : 필드명 + `_` + 참조하는 테이블의 기본 키 컬럼명
> - 예) : 필드명(`team`) + `_` (언더라인) + 참조하는 기본 키 컬럼명(`TEAM_ID`) = `team_TEAM_ID` 외래 키를 사용한다.

- @ManyToOne

이 어노테이션은 다대일 관계에서 사용한다.

`@ManyToOne`의 주요 속성

표 - `@ManyToOne` 속성

속성	기능	기본값
`optional`	`false`로 설정하면 연관된 엔티티가 항상 있어야 한다.	`true`
`fetch`	글로벌 페치 전략을 설정한다. 자세한 내용은 9. 프록시와 연관관계 관리에서 설명한다.	- `@ManyToOne=FetchType.EAGER` - `@OneToMany=FetchType.LAZY`
`cascade`	영속성 전이 기능을 사용한다. 자세한 내용은 9. 프록시와 연관관계 관리에서 설명한다.	
`targetEntity`	연관된 엔티티의 타입 정보를 설정한다. 이 기능은 거의 사용하지 않는다. 컬렉션을 사용해도 제네릭으로 타입 정보를 알 수 있다.	

====== `targetEntity` 속성 사용 예 ======

```
@OneToMany
private List<Member> members; //제네릭으로 타입 정보를 알 수 있다.

@OneToMany(targetEntity=Member.class)
private List members; //제네릭이 없으면 타입 정보를 알 수 없다.
```

연관관계 매핑 작업이 끝났다. 이제 매핑한 연관관계를 사용해보자.

> **참고:** 다대일(`@ManyToOne`)과 비슷한 일대일(`@OneToOne`) 관계도 있다. 단방향 관계를 매핑할 때 둘 중 어떤 것을 사용해야 할지는 반대편 관계에 달려있다. 반대편이 일대다 관계면 다대일을 사용하고 반대편이 일대일 관계면 일대일을 사용하면 된다. 참고로 일대일 관계는 다음장에서 설명한다.

연관관계 사용하기

- 저장

연관관계를 매핑한 엔티티를 어떻게 저장하는지 예제 코드로 알아보자.

===== 회원과 팀을 저장하는 코드 =====

```
public void testSave() {

    //팀1 저장
    Team team1 = new Team("team1", "팀1");
    em.persist(team1);

    //회원1 저장
    Member member1 = new Member("member1", "회원1");
    member1.setTeam(team1); //연관관계 설정 member1 -> team1
    em.persist(member1);

    //회원2 저장
    Member member2 = new Member("member2","회원2");
    member2.setTeam(team1); //연관관계 설정 member2 -> team1
    em.persist(member2);
}
```

> **주의:** JPA에서 엔티티를 저장할 때 연관된 모든 엔티티는 영속 상태여야 한다.

중요한 부분을 분석해보자.

```
member1.setTeam(team1); //회원 -> 팀 참조
em.persist(member1); //저장
```

회원 엔티티는 팀 엔티티를 참조하고 저장했다. JPA는 참조한 팀의 식별자(`Team.id`)를 외래 키로 사용해서 적절한 등록 쿼리를 생성한다. 다음 실행된 SQL을 보면 회원 테이블의 외래 키 값으로 참조한 팀의 식별자 값인 `team1` 이 입력된 것을 확인할 수 있다.

====== 실행된 SQL ======

```
INSERT INTO TEAM (TEAM_ID, NAME) VALUES ('team1', '팀1')
INSERT INTO MEMBER (MEMBER_ID, NAME, TEAM_ID) VALUES ('member1', '회원1', 'team1'
```

```
INSERT INTO MEMBER (MEMBER_ID, NAME, TEAM_ID) VALUES ('member2', '회원2', 'team1'
```

데이터가 잘 입력되었는지 데이터베이스에서 확인해보자.

```
SELECT M.MEMBER_ID, M.NAME, M.TEAM_ID, T.NAME AS TEAM_NAME
FROM MEMBER M
    JOIN TEAM T ON M.TEAM_ID = T.TEAM_ID
```

====== 결과 ======

MEMBER_ID	NAME	TEAM_ID	TEAM_NAME
member1	회원1	team1	팀1
member2	회원2	team1	팀1

- 조회

연관관계가 있는 엔티티를 조회하는 방법은 크게 2가지가 있다.

- 객체 그래프 탐색 (객체 연관관계를 사용한 조회)
- 객체 지향 쿼리 사용 (JPQL)

방금 저장한 대로 회원1, 회원2가 팀1에 소속해 있다고 가정하자.

객체 그래프 탐색

```
Member member = em.find(Member.class, "member1");
Team team = member.getTeam(); //객체 그래프 탐색
System.out.println("팀 이름 = " + team.getName());

//출력결과: 팀 이름 = 팀1
```

`member.getTeam()`을 사용해서 `member`와 연관된 `team` 엔티티를 조회할 수 있다. 이처럼 객체를 통해 연관된 엔티티를 조회하는 것을 객체 그래프 탐색이라 한다. 객체 그래프 탐색에 대한 더 자세한 내용은 9. 프록시와 연관관계 관리에서 설명하겠다.

객체 지향 쿼리 사용

객체 지향 쿼리인 JPQL에서 연관관계를 어떻게 사용하는지 알아보자.

예를 들어 회원을 대상으로 조회하는데 팀1에 소속된 회원만 조회하려면 회원과 연관된 팀 엔티티를 검색 조건으로 사용해야 한다. SQL은 연관된 테이블을 조인해서 검색조건을 사용하면 된다. JPQL도 조인을 지원한다.(문법은 약간 다르다.)

팀1에 소속된 모든 회원을 조회하는 JPQL 예제를 보자.

===== JPQL 조인 검색 =====

```
private static void queryLogicJoin(EntityManager em) {

    String jpql = "select m from Member m join m.team t where t.name=:teamName";

    List<Member> resultList = em.createQuery(jpql, Member.class)
                                .setParameter("teamName", "팀1");
                                .getResultList();

    for (Member member : resultList) {
        System.out.println("[query] member.username=" + member.getUsername());
    }
}
//결과 : [query] member.username=회원1
//결과 : [query] member.username=회원2
```

JPQL의 `from Member m join m.team t` 부분을 보면 회원이 팀과 관계를 가지고 있는 필드(`m.team`)를 통해서 `Member`와 `Team`을 조인했다. 그리고 `where` 절을 보면 조인한 `t.name`을 검색조건으로 사용해서 팀1에 속한 회원만 검색했다.

===== JPQL =====

```
select m from Member m join m.team t
where t.name=:teamName
```

===== 실행되는 SQL =====

```
SELECT M.* FROM MEMBER MEMBER
INNER JOIN
    TEAM TEAM ON MEMBER.TEAM_ID = TEAM1_.ID
WHERE
    TEAM1_.NAME='팀1'
```

실행된 SQL과 JPQL을 비교하면 JPQL은 객체(엔티티)를 대상으로 하고 SQL보다 간결하다. JPQL을 포함한 객체 쿼리에 대한 상세한 내용은 객체 지향 쿼리 장에서 다룬다.

- 수정

이번에는 연관관계를 어떻게 수정하는지 알아보자. 팀1 소속이던 회원을 새로운 팀2에 소속하도록 수정해보자.

===== 연관관계를 수정하는 코드 =====

```
private static void updateRelation(EntityManager em) {

    // 새로운 팀2
    Team team2 = new Team("team2","팀2");
    em.persist(team2);

    // 회원1에 새로운 팀2 설정
    Member member = em.find(Member.class, "member1");
    member.setTeam(team2);
}
```

===== 실행되는 수정 SQL =====

```
UPDATE MEMBER
SET
    TEAM_ID='team2', ...
WHERE
    ID='member1'
```

영속성 관리에서 이야기했지만 수정은 `em.update()` 같은 메서드가 없다. 단순히 불러온 엔티티의 값만 변경해두면 트랜잭션을 커밋할 때 플러시가 일어나면서 **변경 감지** 기능이 작동한다. 그리고 변경사항을 데이터베이스에 자동으로 반영한다. 이것은 연관관계를 수정할 때도 같은데, 참조하는 대상만 변경하면 나머지는 JPA가 자동으로 처리한다.

- 연관관계 제거

이번에는 연관관계를 제거해보자. 회원1을 팀에 소속하지 않도록 변경하자.

===== 연관관계를 삭제하는 코드 =====

```
private static void deleteRelation(EntityManager em) {

    Member member1 = em.find(Member.class, "member1");

    member1.setTeam(null); //연관관계 제거
}
```

연관관계를 널(`null`)로 설정했다.

===== 실행되는 연관관계 제거 SQL =====

```
UPDATE MEMBER
SET
    TEAM_ID=null, ...
WHERE
    ID='member1'
```

- 연관된 엔티티 삭제

연관된 엔티티를 삭제하려면 기존에 있던 연관관계를 먼저 제거하고 삭제해야 한다. 그렇지 않으면 외래 키 제약조건으로 인해, 데이터베이스에서 오류가 발생한다.

팀1에는 회원1과 회원2가 소속되어 있다. 이때 팀1을 삭제하려면 연관관계를 먼저 끊어야 한다.

```
member1.setTeam(null); //회원1 연관관계 제거
member2.setTeam(null); //회원2 연관관계 제거
em.remove(team); //팀 삭제
```

단방향 연관관계 정리

회원에서 팀으로 접근하는 예제로 **다대일 단방향 매핑**을 알아보았다. 이제부터 반대로 팀에서 회원으로 접근하는 일대다 연관관계를 매핑해보자. 그리고 회원에서 팀으로 접근하고 팀에서도 회원으로 접근할 수 있도록 양방향 관계로 매핑해보자.

양방향 연관관계

이번에는 반대방향인 팀에서 회원으로 관계를 추가해서 양방향 연관관계를 만들어보자.

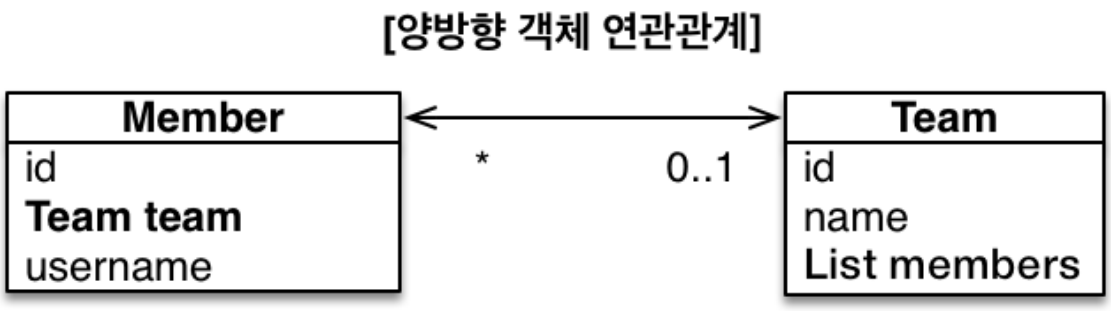

그림 I 양방향 객체 연관관계

회원과 팀은 다대일 관계다. 반대로 팀에서 회원은 일대다 관계다. 일대다 관계는 여러 건과 연관관계를 맺을 수 있으므로 컬렉션을 사용해야 한다. `Team.members` 를 `List` 컬렉션으로 추가했다.

객체 연관관계를 정리하면 다음과 같다.

- 회원 -> 팀 (`Member.team`)
- 팀 -> 회원 (`Team.members`)

> **참고:** JPA는 `List` 를 포함해서 `Collection` , `Set` , `Map` 같은 다양한 컬렉션을 지원한다. 자세한 내용은 20. 컬렉션과 부가기능 장을 참고하자.

데이터베이스 테이블은 외래 키 하나로 양방향 조회가 가능하다.

두 테이블의 연관관계는 외래 키 하나만으로 양방향 조회가 가능하므로 처음부터 양방향 관계다. 따라서 데이터베이스에 추가할 내용은 전혀 없다.

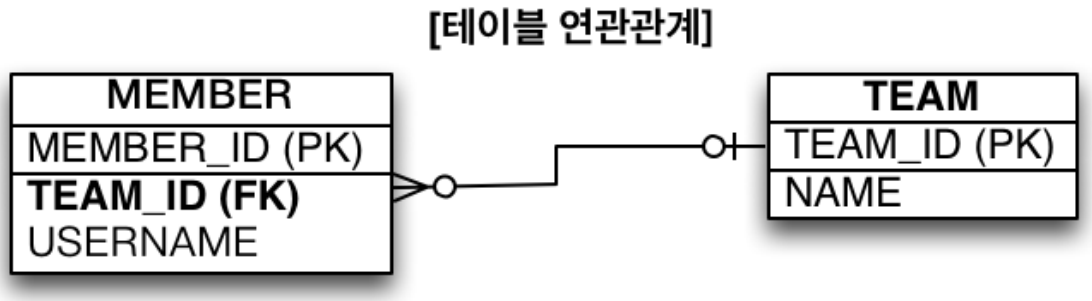

그림 I 테이블의 연관관계

이 장 첫 부분에서도 이야기했지만 `TEAM_ID` 외래 키를 사용해서 `MEMBER JOIN TEAM` 이 가능하고 반대로 `TEAM JOIN MEMBER` 도 가능하다.

- 양방향 연관관계 매핑하기

이제 양방향 관계를 매핑하자. 먼저 회원 엔티티를 보자.

====== 매핑한 회원 엔티티 ======

```
@Entity
public class Member {

    @Id
    @Column(name = "MEMBER_ID")
    private String id;

    private String username;

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

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

    //Getter, Setter ...
}
```

엔티티에는 변경한 부분이 없다. 다음으로 팀 엔티티를 보자.

====== 매핑한 팀 엔티티 ======

```
@Entity
public class Team {

    @Id
    @Column(name = "TEAM_ID")
    private String id;

    private String name;

    //==추가==//
    @OneToMany(mappedBy = "team") //**
    private List<Member> members = new ArrayList<Member>(); //**

    //Getter, Setter ...
}
```

팀과 회원은 일대다 관계다. 따라서 컬렉션인 `List<Member> members`를 추가했다. 그리고 일대다 관계를 매핑하기 위해 `@OneToMany` 매핑 정보를 사용했다. `mappedBy` 속성은 양방향 매핑일 때 사용하는데 반대쪽 매핑의 필드 이름을 값으로 주면 된다. 반대쪽 매핑이 `Member.team`이므로 `team`을 값으로 주었다. `mappedBy`에 대한 자세한 내용은 바로 다음에 나오는 연관관계의 주인에서 설명하겠다.

이것으로 양방향 매핑을 완료했다. 이제부터 팀에서 회원 컬렉션으로 객체 그래프를 탐색할 수 있다. 이것을 사용해서 팀1에 소속된 모든 회원을 찾아보자.

- 일대다 컬렉션 조회

====== 일대다 방향으로 객체 그래프 탐색 ======

```
public void biDirection() {

    Team team = em.find(Team.class, "team1");
    List<Member> members = team.getMembers(); //(팀 -> 회원) 객체 그래프 탐색

    for (Member member : members) {
        System.out.println("member.username = " + member.getUsername());
    }
}
//==결과==
//member.username = 회원1
//member.username = 회원2
```

팀에서 회원 컬렉션으로 객체 그래프 탐색을 사용해서 조회한 회원들을 출력했다.

연관관계의 주인(Owner)

`@OneToMany`는 직관적으로 이해가 될 것이다. 문제는 `mappedBy` 속성이다. 단순히 `@OneToMany`만 있으면 되지 `mappedBy`는 왜 필요할까?

엄밀히 이야기하면 객체에는 양방향 연관관계라는 것이 없다. 서로 다른 단방향 연관관계 2개를 애플리케이션 로직으로 잘 묶어서 양방향인 것처럼 보이게 할 뿐이다. 반면에 데이터베이스 테이블은 앞서 설명했듯이 외래 키 하나로 양쪽이 서로 조인할 수 있다. 따라서 테이블은 외래 키 하나만으로 양방향 연관관계를 맺는다.

객체 연관관계

- 회원 -> 팀 연관관계 1개(단방향)
- 팀 -> 회원 연관관계 1개(단방향)

테이블 연관관계

- 회원 <-> 팀의 연관관계 1개(양방향)

다시 강조하지만, **테이블은 외래 키 하나로 두 테이블의 연관관계를 관리**한다.

엔티티를 단방향으로 매핑하면 참조를 하나만 사용하므로 이 참조로 외래 키를 관리하면 된다. 그런데 엔티티를 양방향으로 매핑하면 `회원 -> 팀`, `팀 -> 회원` 두 곳에서 서로를 참조한다. 따라서 객체의 연관관계를 관리하는 포인트는 2곳으로 늘어난다.

엔티티를 양방향 연관관계로 설정하면 객체의 참조는 둘인데 외래 키는 하나인 차이가 발생한다. 그렇다면 둘 중 어떤 관계를 사용해서 외래 키를 관리해야 할까?

이런 차이로 인해 JPA에서는 **두 객체 연관관계 중 하나를 정해서 테이블의 외래 키를 관리해야 하는데 이것을 연관관계의 주인**이라 한다.

- 양방향 매핑의 규칙 - 연관관계의 주인(Owner)

양방향 연관관계 매핑시 지켜야 할 규칙이 있는데 두 연관관계 중 하나를 연관관계의 주인으로 정해야 한다. **연관관계의 주인만이 데이터베이스 연관관계와 매핑되고 외래 키를 관리(등록, 수정, 삭제)할 수 있다. 반면에 주인이 아닌 쪽은 읽기만 할 수 있다.**

어떤 연관관계를 주인으로 정할지는 `mappedBy` 속성을 사용하면 된다.

- 주인은 `mappedBy` 속성을 사용하지 않는다.
- 주인이 아니면 `mappedBy` 속성을 사용해서 속성의 값으로 연관관계의 주인을 지정해야 한다.

그렇다면 `Member.team`, `Team.members` 둘 중 어떤 것을 연관관계의 주인으로 정해야 할까?

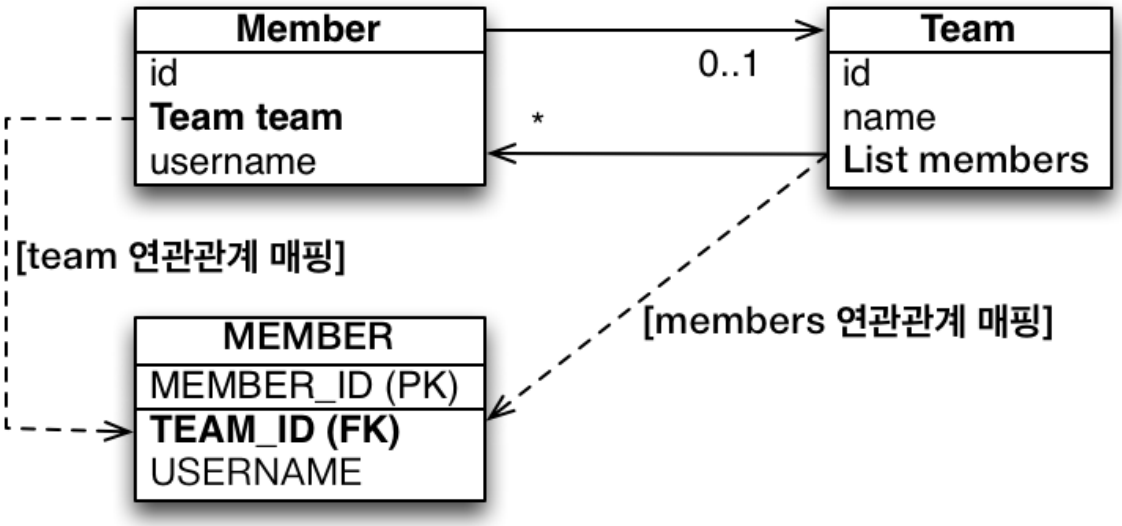

그림 I 둘 중 하나를 연관관계의 주인으로 선택해야 한다.

====== 회원 -> 팀(`Member.team`) ======

```
class Member {
    @ManyToOne
    @JoinColumn(name="TEAM_ID")
    private Team team;
    ...
}
```

====== 팀 -> 회원(`Team.members`) ======

```
class Team {
    @OneToMany
    private List<Member> members = new ArrayList<Member>();
    ...
}
```

연관관계의 주인을 정한다는 것은 사실 외래 키 관리자를 선택하는 것이다. 여기서는 회원 테이블에 있는 `TEAM_ID` 외래 키를 관리할 관리자를 선택해야 한다. 만약 회원 엔티티에 있는 `Member.team`을 주인으로 선택하면 자기 테이블에 있는 외래 키를 관리하면 된다. 하지만 팀 엔티티에 있는 `Team.members`를 주인으로 선택하면 물리적으로 전혀 다른 테이블의 외래 키를 관리해야 한다. 왜냐하면 이 경우 `Team.members`가 있는 `Team` 엔티티는 `TEAM` 테이블에 매핑되어 있는데 관리해야할 외래 키는 `MEMBER` 테이블에 있기 때문이다.

- 연관관계의 주인은 외래 키가 있는 곳

연관관계의 주인은 테이블에 외래 키가 있는 곳으로 정해야 한다. 여기서는 회원 테이블이 외래 키를 가지고 있으므로 `Member.team` 이 주인이 된다. 주인이 아닌 `Team.members` 에는 `mappedBy="team"` 속성을 사용해서 주인이 아님을 설정하고 `mappedBy` 속성의 값으로는 연관관계의 주인인 `team` 을 주면 된다.

```
class Team {

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

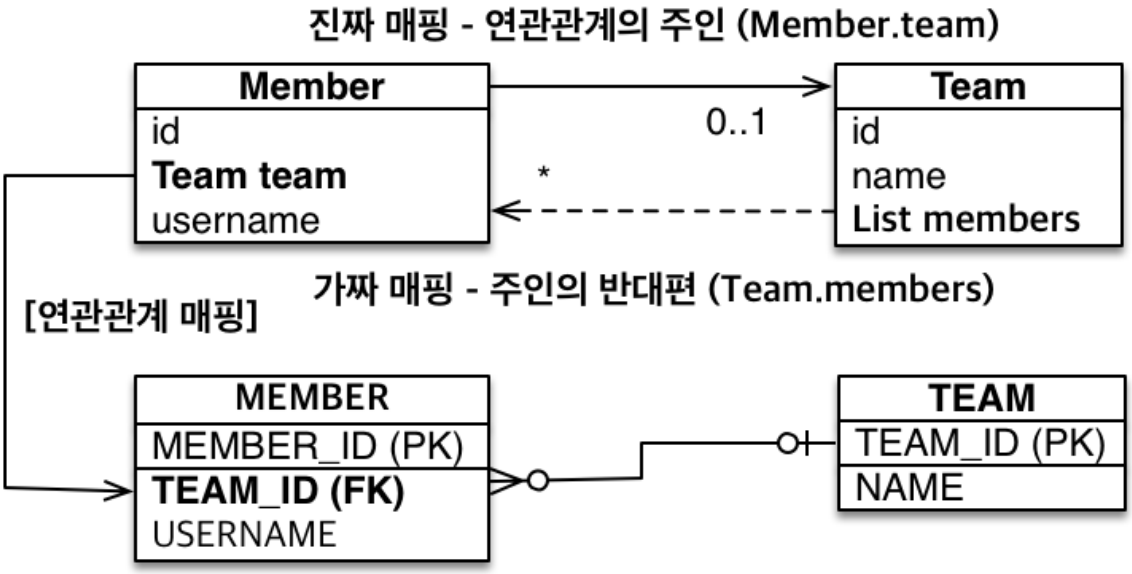

그림 I 연관관계의 주인과 반대편

정리하면 **연관관계의 주인만 데이터베이스 연관관계와 매핑되고 외래 키를 관리할 수 있다. 주인이 아닌 반대편(inverse, non-owning side)은 읽기만 가능하고 외래 키를 변경하지는 못한다.**

> **참고:** 데이터베이스 테이블의 다대일, 일대다 관계에서는 항상 다 쪽이 외래 키를 가진다. 따라서 다 쪽인 `@ManyToOne` 는 항상 연관관계의 주인이 되므로 `mappedBy` 를 설정할 수 없다. 따라서 `@ManyToOne` 에는 `mappedBy` 속성이 없다.

양방향 연관관계 저장하기

양방향 연관관계를 사용해서 팀1, 회원1, 회원2를 저장해보자.

```
public void testSave() {

    //팀1 저장
    Team team1 = new Team("team1", "팀1");
    em.persist(team1);

    //회원1 저장
    Member member1 = new Member("member1", "회원1");
    member1.setTeam(team1); //연관관계 설정 member1 -> team1
    em.persist(member1);

    //회원2 저장
    Member member2 = new Member("member2","회원2");
    member2.setTeam(team1); //연관관계 설정 member2 -> team1
    em.persist(member2);
}
```

팀1을 저장하고 회원1, 회원2에 연관관계의 주인인 `Member.team` 필드를 통해서 회원과 팀의 연관관계를 설정하고 저장했다.

데이터베이스에서 회원 테이블을 조회해 보자.

```
SELECT * FROM MEMBER;
```

회원 테이블 조회

MEMBER_ID	USERNAME	TEAM_ID
member1	회원1	team1
member2	회원2	team1

`TEAM_ID` 외래 키에 팀의 기본 키 값이 저장되어 있다.

양방향 연관관계는 연관관계의 주인이 외래 키를 관리한다. 따라서 주인이 아닌 방향은 값을 설정하지 않아도 데이터베이스에 외래 키 값이 정상 입력된다.

```
team1.getMembers().add(member1); //무시(연관관계의 주인이 아님)
```

```
team1.getMembers().add(member2); //무시(연관관계의 주인이 아님)
```

이런 코드가 추가로 있어야 할 것 같지만 `Team.members` 는 연관관계의 주인이 아니다. 주인이 아닌 곳에 입력된 값은 외래 키에 영향을 주지 않는다. 따라서 위 코드는 데이터베이스에 저장시 무시된다.

```
member1.setTeam(team1); //연관관계 설정(연관관계의 주인)
member2.setTeam(team1); //연관관계 설정(연관관계의 주인)
```

`Member.team` 은 연관관계의 주인이다. 엔티티 매니저는 이곳에 입력된 값을 사용해서 외래 키를 관리한다.

양방향 연관관계의 주의점

양방향 연관관계를 설정하고 가장 흔히 하는 실수는 연관관계의 주인에는 값을 입력하지 않고, 주인이 아닌 곳에만 값을 입력하기 때문에 발생한다. 데이터베이스에 외래 키 값이 정상적으로 저장되지 않으면 이것부터 의심해보자.

주인이 아닌 곳에만 값을 설정하면 어떻게 되는지 예제로 알아보자.

====== 양방향 연관관계 주의점 ======

```
public void testSaveNonOwner() {

    //회원1 저장
    Member member1 = new Member("member1", "회원1");
    em.persist(member1);

    //회원2 저장
    Member member2 = new Member("member2","회원2");
    em.persist(member2);

    Team team1 = new Team("team1", "팀1");
    //주인이 아닌 곳만 연관관계 설정
    team1.getMembers().add(member1);
    team1.getMembers().add(member2);

    em.persist(team1);
}
```

회원1, 회원2를 저장하고 팀의 컬렉션에 담은 후에 팀을 저장했다. 데이터베이스에서 회원 테이블을 조회해보자.

```
SELECT * FROM MEMBER;
```

회원 테이블 조회

MEMBER_ID	USERNAME	TEAM_ID
member1	회원1	null
member2	회원2	null

외래 키 `TEAM_ID`에 `team1`이 아닌 `null` 값이 입력되어 있다. 왜냐하면 연관관계의 주인이 아닌 `Team.members`에만 값을 저장했기 때문이다. 다시 한번 강조하지만, 연관관계의 주인만이 외래 키의 값을 변경할 수 있다. 예제 코드는 연관관계의 주인인 `Member.team`에 아무 값도 입력하지 않았다. 따라서 `TEAM_ID` 외래 키의 값도 `null`이 저장된다.

- 순수한 객체까지 고려한 양방향 연관관계

그렇다면 정말 연관관계의 주인에만 값을 저장하고 주인이 아닌 곳에는 값을 저장하지 않아도 될까? 사실은 **객체 관점에서 양쪽 방향에 모두 값을 입력해 주는 것이 가장 안전하다.** 양쪽 방향 모두 값을 입력하지 않으면 JPA를 사용하지 않는 순수한 객체 상태에서 심각한 문제가 발생할 수 있다.

예를 들어 JPA를 사용하지 않고 엔티티에 대한 테스트 코드를 작성한다고 가정해보자. ORM은 객체와 관계형 데이터베이스 둘다 중요하다. 데이터베이스뿐만 아니라 객체도 함께 고려해야 한다. 다음 코드를 보자.

====== 순수한 객체 연관관계 ======

```
public void test순수한객체_양방향() {

    //팀1
    Team team1 = new Team("team1", "팀1");
    Member member1 = new Member("member1", "회원1");
    Member member2 = new Member("member2","회원2");

    member1.setTeam(team1); //연관관계 설정 member1 -> team1
    member2.setTeam(team1); //연관관계 설정 member2 -> team1
```

```
    List<Member> members = team1.getMembers();
    System.out.println("members.size = " + members.size());
}
//결과: members.size = 0
```

예제 코드는 JPA를 사용하지 않는 순수한 객체다. 코드를 보면 `Member.team`에만 연관관계를 설정하고 반대 방향은 연관관계를 설정하지 않았다. 마지막 줄에서 팀에 소속된 회원이 몇 명인지 출력해보면 결과는 0이 나온다. 이것은 우리가 기대하는 양방향 연관관계의 결과가 아니다.

```
member1.setTeam(team1); //회원 -> 팀
```

양방향은 양쪽다 관계를 설정해야 한다. 이처럼 회원 -> 팀을 설정하면 다음 코드처럼 반대방향인 팀 -> 회원도 설정해야 한다.

```
team1.getMembers().add(member1); //팀 -> 회원
```

양쪽 모두 관계를 설정한 전체 코드를 보자.

```
public void test순수한객체_양방향() {

    //팀1
    Team team1 = new Team("team1", "팀1");
    Member member1 = new Member("member1", "회원1");
    Member member2 = new Member("member2","회원2");

    member1.setTeam(team1);          //연관관계 설정 member1 -> team1
    team1.getMembers().add(member1);//연관관계 설정 team1 -> member1

    member2.setTeam(team1);          //연관관계 설정 member2 -> team1
    team1.getMembers().add(member2);//연관관계 설정 team1 -> member2

    List<Member> members = team1.getMembers();
    System.out.println("members.size = " + members.size());
}
//결과: members.size = 2
```

양쪽 모두 관계를 설정했다. 결과도 기대했던 2가 출력된다.

```
member1.setTeam(team1); //회원 -> 팀
```

```
team1.getMembers().add(member1);//팀 -> 회원
```

객체까지 고려하면 이렇게 양쪽다 관계를 맺어야 한다.

이제 JPA를 사용해서 코드를 완성해보자.

```
public void testORM_양방향() {

    //팀1 저장
    Team team1 = new Team("team1", "팀1");
    em.persist(team1);

    Member member1 = new Member("member1", "회원1");

    //양방향 연관관계 설정
    member1.setTeam(team1);          //연관관계 설정 member1 -> team1
    team1.getMembers().add(member1);//연관관계 설정 team1 -> member1
    em.persist(member1);

    Member member2 = new Member("member2","회원2");

    //양방향 연관관계 설정
    member2.setTeam(team1);          //연관관계 설정 member2 -> team1
    team1.getMembers().add(member2);//연관관계 설정 team1 -> member2
    em.persist(member2);
}
```

양쪽에 연관관계를 설정했다. 물론 외래 키의 값은 연관관계의 주인인 `Member.team` 값을 사용한다.

```
member1.setTeam(team1); //연관관계의 주인
team1.getMembers().add(member1);//주인이 아니다. 저장시 사용되지 않는다.
```

- `Member.team` : 연관관계의 주인, 이 값으로 외래 키를 관리한다.
- `Team.members` : 연관관계의 주인이 아니다. 따라서 저장시에 사용되지 않는다.

앞서 이야기한 것처럼 객체까지 고려해서 주인이 아닌 곳에도 값을 입력하자.

결론: 객체의 양방향 연관관계는 양쪽 모두 관계를 맺어주자.

- 연관관계 편의 메서드

양방향 연관관계는 결국 양쪽다 신경을 써야한다. 다음처럼 `member.setTeam(team)` 과 `team.getMembers().add(member)`를 각각 호출하다 보면 실수로 둘 중 하나만 호출해서 양방향이 깨질 수 있다.

```
member.setTeam(team);
team.getMembers().add(member);
```

양방향 관계에서 두 코드는 하나인 것처럼 사용하는 것이 안전하다.

`Member` 클래스의 `setTeam()` 메서드를 수정해서 코드를 리팩토링 해보자.

```
public class Member {

    private Team team;

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

`setTeam()` 메서드 하나로 양방향 관계를 모두 설정하도록 변경했다.

연관관계를 설정하는 부분을 수정하자.

```
//연관관계 설정
member1.setTeam(team1);
member2.setTeam(team1);

//== 기존 코드 삭제 시작==//
//teamA.getMembers().add(member1); //팀1->회원1
//teamA.getMembers().add(member2); //팀1->회원2
//== 기존 코드 삭제 종료 ==//
```

이렇게 리팩토링하면 실수도 줄어들고 좀 더 그럴듯하게 양방향 연관관계를 설정할 수 있다.

전체 코드를 보자.

====== 양방향 리팩토링 전체코드 ======

```
public void testORM_양방향_리팩토링() {

    Team team1 = new Team("team1", "팀1");
    em.persist(team1);

    Member member1 = new Member("member1", "회원1");
    member1.setTeam(team1); //양방향 설정
    em.persist(member1);

    Member member2 = new Member("member2","회원2");
    member2.setTeam(team1); //양방향 설정
    em.persist(member2);
}
```

이렇게 한 번에 양방향 관계를 설정하는 메서드를 **연관관계 편의 메서드**라 한다.

- 연관관계 편의 메서드 작성시 주의사항

사실 `setTeam()` 메서드에는 버그가 있다. (리팩토링 전에도 버그는 있었다.)

```
member1.setTeam(teamA); //1
member1.setTeam(teamB); //2
```

이 시나리오를 그림으로 분석해보자.

먼저 `member1.setTeam(teamA)`을 호출한 직후 객체 연관관계를 보자.

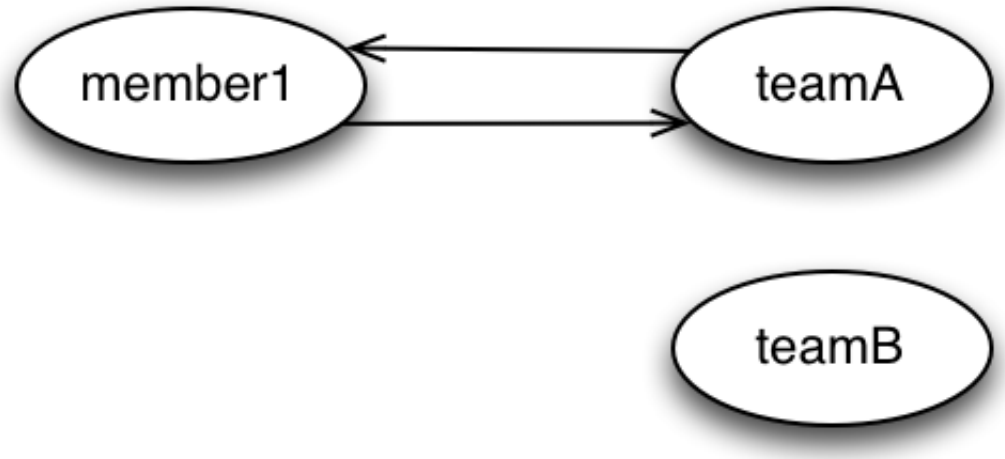

그림 3.6 I 삭제되지 않은 관계1

다음으로 `member1.setTeam(teamB)` 을 호출한 직후 객체 연관관계를 보자.

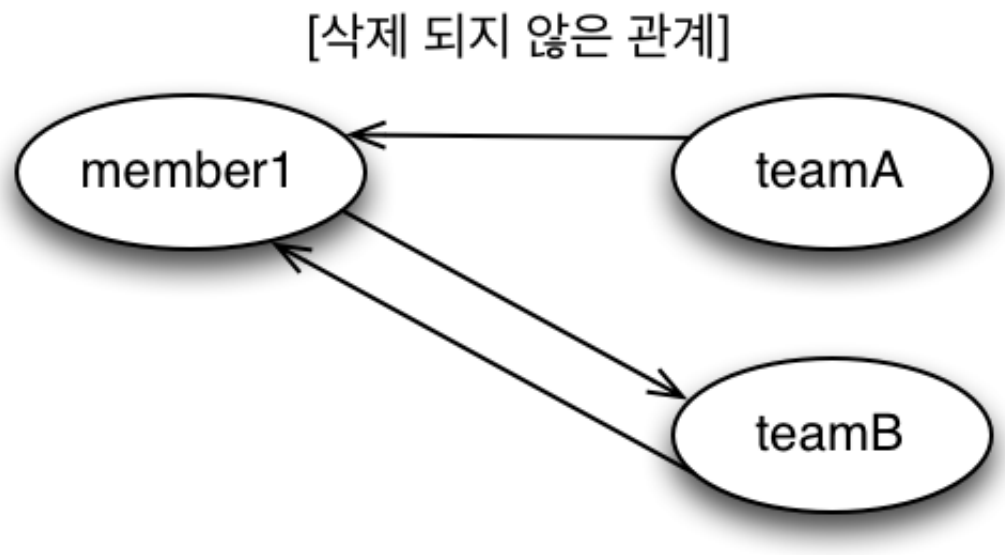

그림 3.7 I 삭제되지 않은 관계1

무엇이 문제인지 보이는가? `teamB` 로 변경할 때 `teamA` -> `member1` 관계를 제거하지 않았다. 연관관계를 변경할 때는 기존 팀이 있으면 기존 팀과 회원의 연관관계를 삭제하는 코드를 추가해야 한다. 따라서 다음처럼 코드를 수정해야 한다.

```
public void setTeam(Team team) {

    //기존 팀과 관계를 제거
    if (this.team != null) {
        this.team.getMembers().remove(this);
    }
    this.team = team;
    team.getMembers().add(this);
}
```

이 코드는 객체에서 서로 다른 단방향 연관관계 2개를 양방향인 것처럼 보이게 하려고 얼마나 많은 고민과 수고가 필요한지 보여준다. 반면에 관계형 데이터베이스는 외래 키 하나로 문제를 단순하게 해결한다. 정리하자면 객체에서 양방향 연관관계를 사용하려면 로직을 견고하게 작성해야 한다.

> **심화** : [그림 삭제되지 않은 관계]에서 `teamA` -> `member1` 관계가 제거되지 않아도 데이터베이스 외래 키를 변경하는데는 문제가 없다. 왜냐하면 `teamA` -> `member1` 관계를 설정한 `Team.members` 는 연관관계의 주인이 아니기 때문이다. 연관관계의 주인인

> `Member.team`의 참조를 `member1` -> `teamB`로 변경했으므로 데이터베이스에 외래 키는 `teamB`를 참조하도록 정상 반영된다.
>
> 그리고 이후에 새로운 영속성 컨텍스트에서 `teamA`를 조회해서 `teamA.getMembers()`를 호출하면 데이터베이스 외래 키에는 관계가 끊어져 있으므로 아무것도 조회되지 않는다. 여기까지만 보면 특별한 문제가 없는 것 같다.
>
> 문제는 관계를 변경하고 영속성 컨텍스트가 아직 살아있는 상태에서 `teamA`의 `getMembers()`를 호출하면 `member1`이 반환된다는 점이다. 따라서 변경된 연관관계는 설명한 것처럼 관계를 제거하는 것이 안전하다.

양방향 매핑 정리

단방향 매핑과 비교해서 양방향 매핑은 복잡하다. 연관관계의 주인도 정해야 하고, 두 개의 단방향 연관관계를 양방향으로 만들기 위해 로직도 잘 관리해야 한다. 중요한 사실은 연관관계가 하나인 단방향 매핑은 언제나 연관관계의 주인이라는 점이다. 양방향은 여기에 주인이 아닌 연관관계를 하나 추가했을 뿐이다. 결국 단방향과 비교해서 **양방향의 장점은 반대방향으로 객체 그래프 탐색 기능이 추가된 것** 뿐이다.

```
member.getTeam();  //회원 -> 팀
team.getMembers(); //팀 -> 회원 (양방향 매핑으로 추가된 기능)
```

주인의 반대편은 `mappedBy`로 주인을 지정해야 한다. 그리고 주인의 반대편은 단순히 보여주는 일(객체 그래프 탐색)만 할 수 있다.

정리

- **단방향 매핑만으로 테이블과 객체의 연관관계 매핑은 이미 완료되었다.**
- **단방향을 양방향으로 만들면 반대방향으로 객체 그래프 탐색 기능이 추가된다.**
- **양방향 연관관계를 매핑하려면 객체에서 양쪽 방향을 모두 관리해야 한다.**

양방향 매핑은 복잡하다. 비즈니스 로직의 필요에 따라 다르겠지만 우선 단방향 매핑을 사용하고 반대방향으로 객체 그래프 탐색 기능(JPQL 쿼리 탐색 포함)이 필요할 때 양방향을 사용하도록 코드를 추가해도 된다.

> **중요 : 연관관계의 주인을 정하는 기준**

> 단방향은 항상 외래 키가 있는 곳을 기준으로 매핑하면 된다. 하지만 양방향은 연관관계의 주인(Owner)이라는 이름으로 인해 오해가 있을 수 있다. 비즈니스 로직상 더 중요하다고 연관관계의 주인으로 선택하면 안 된다. 비즈니스 중요도를 배제하고 단순히 외래 키 관리자 정도의 의미만 부여해야 한다.
>
> 예를 들어 회원과 팀 엔티티는 외래 키가 있는 다 쪽인 회원이 연관관계의 주인이 된다. 물론 비즈니스 중요도를 생각해보면 팀보다 회원이 더 중요한 것 같은 느낌도 들 것이다. 하지만 자동차의 자체와 바퀴를 생각해보면 바퀴가 외래 키가 있는 다 쪽이다. 따라서 바퀴가 연관관계의 주인이 된다. 차체가 더 중요한 것 같아 보이지만 연관관계의 주인은 단순히 외래 키를 매핑한 바퀴를 선택하면 된다.
>
> 따라서 **연관관계의 주인은 외래 키의 위치와 관련해서 정해야지 비즈니스 중요도로 접근하면 안 된다.**

> **참고:** 일대다를 연관관계의 주인으로 선택하는 것이 불가능한 것만은 아니다. 예를 들어 팀 엔티티의 `Team.members`를 연관관계의 주인으로 선택하는 것이다. 하지만 성능과 관리 측면에서 권장하지 않는다. 될 수 있으면 외래 키가 있는 곳을 연관관계의 주인으로 선택하자. 자세한 내용은 6. 연관관계 매핑 - 실전 장에서 설명하겠다.

[실전 예제] - 2. 연관관계 매핑 시작하기

> **예제 코드 :** `ch05-model2`

앞의 실전 예제는 외래 키를 엔티티에 그대로 가져오는 문제가 있었다. 엔티티에서 외래 키로 사용된 필드는 제거하고 참조를 사용하도록 변경해보자.

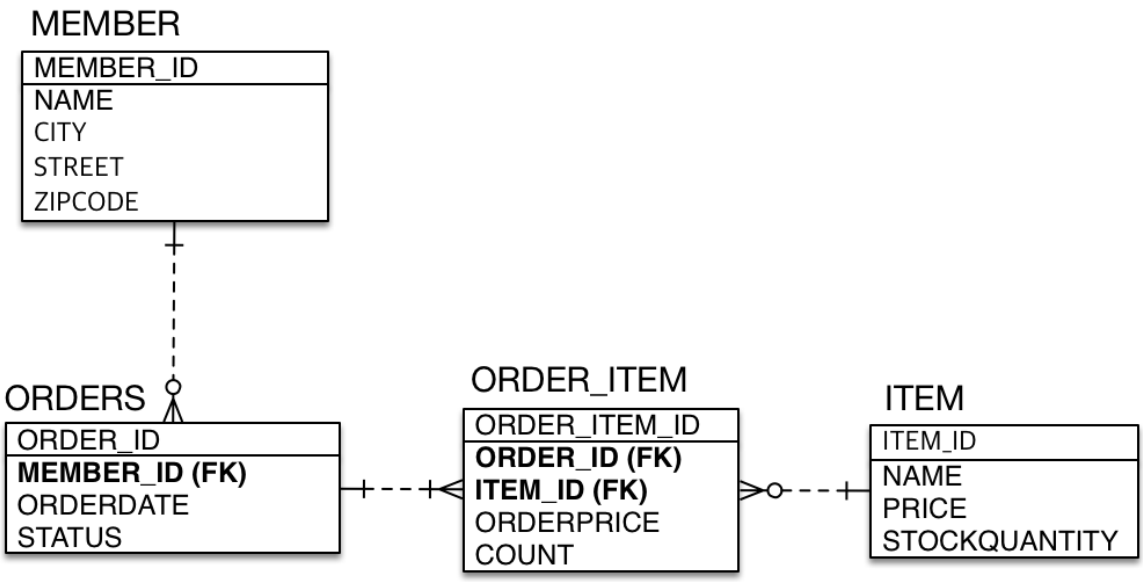

그림 - ERD2

테이블 구조는 이전 예제와 같다.

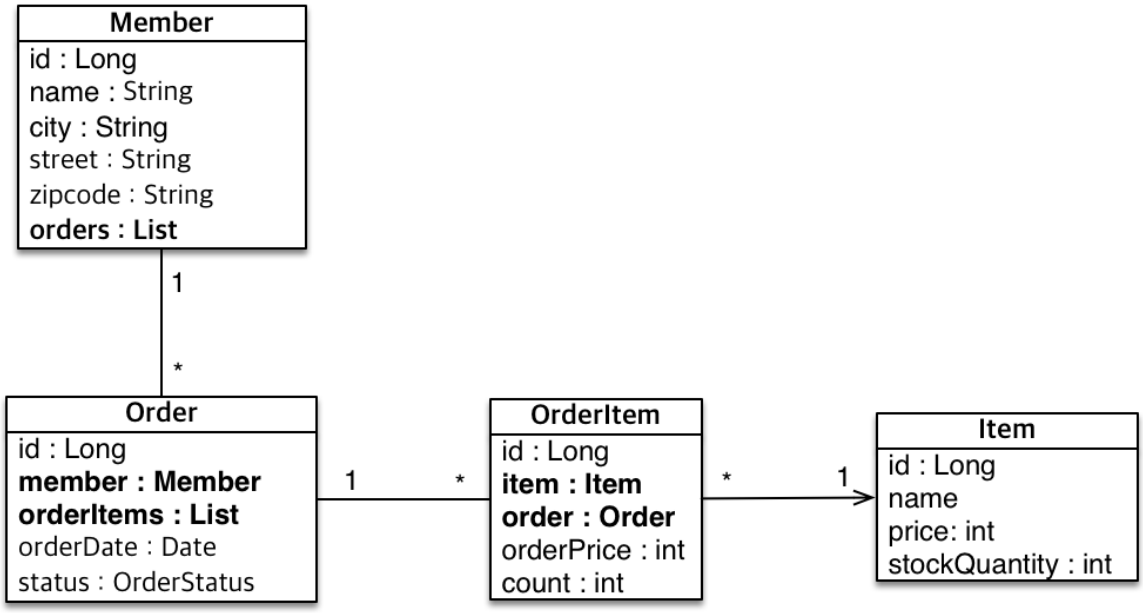

그림 - UML 상세2

객체 관계는 외래 키를 직접 사용하는 것에서 참조를 사용하도록 변경했다.

일대다, 다대일 연관관계 매핑하기

=== 회원(`Member`) ===

```
@Entity
public class Member {

    @Id @GeneratedValue
    @Column(name = "MEMBER_ID")
    private Long id;

    private String name;

    private String city;
    private String street;
    private String zipcode;

    @OneToMany(mappedBy = "member")
    private List<Order> orders = new ArrayList<Order>();

    //Getter, Setter
    ...
```

=== 주문(`Order`) ===

```
package jpabook.model.entity;

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

@Entity
@Table(name = "ORDERS")
public class Order {

    @Id @GeneratedValue
    @Column(name = "ORDER_ID")
    private Long id;

    @ManyToOne
    @JoinColumn(name = "MEMBER_ID")
    private Member member;      //주문 회원

    @OneToMany(mappedBy = "order")
    private List<OrderItem> orderItems = new ArrayList<OrderItem>();

    @Temporal(TemporalType.TIMESTAMP)
    private Date orderDate;     //주문시간

    @Enumerated(EnumType.STRING)
```

```
    private OrderStatus status;//주문상태

    //==연관관계 메서드==//
    public void setMember(Member member) {
        //기존 관계 제거
        if (this.member != null) {
            this.member.getOrders().remove(this);
        }
        this.member = member;
        member.getOrders().add(this);
    }

    public void addOrderItem(OrderItem orderItem) {
        orderItems.add(orderItem);
        orderItem.setOrder(this);
    }

    //Getter, Setter
    ...
}

public enum OrderStatus {
    ORDER, CANCEL
}
```

회원과 주문은 일대다 관계고 그 반대인 주문과 회원은 다대일 관계다.

`Order -> Member` 로 참조하는 `Order.member` 필드와 `Member -> Order` 로 참조하는 `Member.orders` 필드 중에 외래 키가 있는 `Order.member` 가 연관관계의 주인이다. 따라서 주인이 아닌 `Member.orders` 에는 `@OneToMany` 속성에 `mappedBy` 를 선언해서 연관관계의 주인인 `member` 를 지정했다. 참고로 여기서 지정한 `member` 는 `Order.member` 필드다.

연관관계 편의 메서드

양방향 연관관계인 두 엔티티간에 관계를 맺을 때는 원래 다음처럼 설정해야 한다.

```
Member member = new Member();
Order order = new Order();

member.getOrders().add(order); //member -> order
order.setMember(member);       //order -> member
```

여기서는 `Order` 엔티티에 `setMember()` 라는 연관관계 편의 메서드를 추가했으므로, 다음처럼 한 곳만 관계를 설정하면 된다.

```
Member member = new Member();
Order order = new Order();
order.setMember(member);    //member -> order, order -> member
```

=== 주문상품(`OrderItem`) ===

```
package jpabook.model.entity;

import javax.persistence.*;

@Entity
@Table(name = "ORDER_ITEM")
public class OrderItem {

    @Id @GeneratedValue
    @Column(name = "ORDER_ITEM_ID")
    private Long id;

    @ManyToOne
    @JoinColumn(name = "ITEM_ID")
    private Item item;      //주문 상품

    @ManyToOne
    @JoinColumn(name = "ORDER_ID")
    private Order order;    //주문

    private int orderPrice; //주문 가격
    private int count;      //주문 수량

    //Getter, Setter
    ...
}
```

주문과 주문상품은 일대다 관계고 그 반대는 다대일 관계다.

`OrderItem -> Order`로 참조하는 `OrderItem.order` 필드와 `Order -> OrderItem`으로 참조하는 `Order.orderItems` 필드 둘 중에 외래 키가 있는 `OrderItem.order`가 연관관계의 주인이다. 따라서 `Order.orderItems` 필드에는 `mappedBy` 속성을 사용해서 주인이 아님을 표시했다.

=== 상품(`Item`) ===

```
package jpabook.model.entity;
```

```
import javax.persistence.*;

@Entity
public class Item {

    @Id
    @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id;

    private String name;         //이름
    private int price;           //가격
    private int stockQuantity;   //재고수량

    //Getter, Setter
    ...
}
```

비즈니스 요구사항을 분석해본 결과 주문상품에서 상품을 참조할 일을 많지만, 상품에서 주문상품을 참조할 일은 거의 없었다. 따라서 주문상품과 상품은 다대일 단방향 관계로 설정했다. 즉 `OrderItem -> Item` 방향으로 참조하는 `OrderItem.item` 필드만 사용해서 다대일 단방향 관계로 설정했다.

객체 그래프 탐색

이제 객체에서 참조를 사용할 수 있으므로, 객체 그래프를 탐색할 수 있고, JPQL에서도 사용할 수 있다.

주문한 회원을 객체 그래프로 탐색해보자.

```
Order order = em.find(Order.class, orderId);
Member member = order.getMember(); //주문한 회원, 참조 사용
```

주문한 상품 하나를 객체 그래프로 탐색해보자.

```
Order order = em.find(Order.class, orderId);
orderItem = order.getOrderItems().get(0);
Item = orderItem.getItem();
```