# 5장 연관관계 매핑 기초

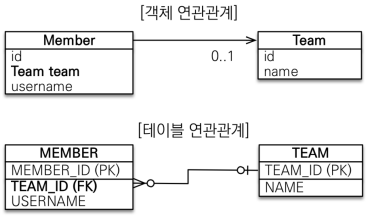
기간 : 21.12.17 ~ 21.12.29

▼ 목차



- 방향(Direction) : 단방향, 양방향
    - 방향은 객체관계에만 존재하고 테이블 관계는 항상 양방향이다.
- 다중성(Multiplicity) : 다대일(N:1), 일대다(1:N), 일대일(1:1), 다대다(N:M)
- 연관관계의 주인(Owner) : 객체를 양방향 연관관계로 만들면 연관관계의 주인을 정해야한다.

## 5.1 단방향 연관관계 164p




**객체 연관관계**

- 회원 객체는 `Member.team` 필드(멤버 변수)로 팀 객체와 연관관계를 맺는다.
- 회원 객체와 팀 객체는 **단방향 관계**다.
    - member → team 조회 : `member.getTeam()`
    - team → member 조회 : 불가능

**테이블 연관관계**

- 회원 테이블은 `TEAM_ID` 외래 키로 팀 테이블과 연관관계를 맺는다.

팀과 회원 조인

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

회원과 팀 조인

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

- 즉 **외래 키 하나**로 두 테이블의 연관 관계를 관리한다.
- 회원 테이블과 팀 테이블은 **양방향 관계**다.

#### 객체 연관관계와 테이블 연관관계의 가장 큰 차이

- 참조를 통한 연관관계는 늘 단방향이다.
  - 양 쪽에서 서로 참조하는 것을 양방향 연관관계라고 하는데, 정확히 이야기하면 양방향 관계가 아니라 서로 다른 단방향 관계 2개다.
- 테이블은 외래키 하나로 양방향으로 조인할 수 있다.

단방향 연관관계

```
class A {
    B b;
}

class B {

}
```

양방향 연관관계

```
class A {
    B b;
}

class B {
    A a;
}
```

객체를 양방향으로 참조하려면 단방향 연관관계를 2개 만들어야 한다.

#### 객체 연관관계 vs 테이블 연관관계 정리

| | 연관관계 | 연관된 데이터 조회 | 연관관계 방향 |
|---|---|---|---|
| **객체** | 참조(주소)로 맺음 | 참조 `a.getB().getC()` | 단방향 A -> B `(a.b)` |
| **테이블** | 외래키로 맺음 | 조인 `JOIN` | 양방향 `A JOIN B` , `B JOIN A` |

- 객체를 양방향으로 참조하려면 단방향 연관관계를 2개 만들어야 한다.
  - A → B `a.b`
  - B → A `b.a`

### 5.1.1 순수한 객체 연관관계 167p

- JPA를 사용하지 않은 순수한 회원과 팀 클래스의 코드
- 객체는 참조를 사용해서 연관관계를 탐색할 수 있는데 이것을 **객체 그래프 탐색** 이라 한다.
  `Team findTeam = member1.getTeam();`

### 5.1.2 테이블 연관관계 169p

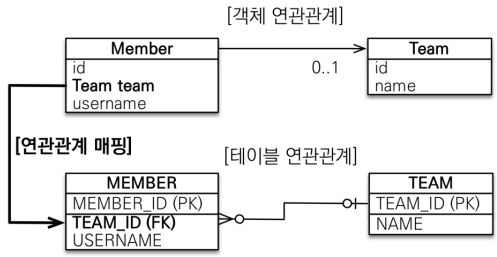
- 데이터베이스 테이블의 회원과 팀의 관계

### 5.1.3 객체 관계 매핑 170p

- JPA를 사용하여 매핑

```
@Entity
public class Member {
    //
    @Id
    @Column(name = "MEMBER_ID")
    private String id;

    private String username;

    // 연관관계 매핑
    @ManyToOne
    @JoinColumn(name="TEAM_ID")
```

```
@Entity
public class Team {
    //
    @Id
    @Column(name = "TEAM_ID")
    private String id;

    private String name;

    // Getter, Setter
}
```




```
    private Team team;

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

    // Getter, Setter
}
```

- **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.

- `@ManyToOne` : 다대일(N:1) 관계라는 매핑 정보
  - 회원과 팀은 다대일 관계
  - 연관관계를 매핑할 때 이렇게 다중성을 나타내는 어노테이션을 필수로 사용해야한다.
- `@JoinColumn(name="TEAM_ID")`
  - 외래 키를 매핑할 때 사용
  - name 속성 : 매핑할 외래키 이름을 지정
  - 회원과 팀 테이블은 TEMA_ID 외래키로 연관관계를 맺으므로 이 값을 지정하면 됨
  - 이 어노테이션은 생략 가능하다.
    - 생략할 경우 외래 키를 찾을 때 기본 전략을 사용한다.
    - 기본 전략 : 필드명 + _ + 참조하는 테이블의 컬럼명

```
@ManyToOne
private Team team;
```

ex) 필드명(team) + _(밑줄) + 참조하는 테이블의 컬럼명(TEAM_ID) ⇒ `team_TEAM_ID` 외래키를 사용한다.

### 5.1.4 @JoinColumn 172p

- 외래키를 매핑할 때 사용

<주요 속성>

| 속성 | 설명 | 기본값 |
|---|---|---|
| name | 매핑할 외래 키 이름 | 필드명 + _ + 참조하는 테이블의 기본 키 컬럼명 |
| referencedColumnName | 외래 키가 참조하는 대상 테이블의 컬럼명 | 참조하는 테이블의 기본 키 컬럼명 |
| foreignKey(DDL) | - 외래 키 제약조건을 직접 지정할 수 있다. - 이 속성은 테이블을 생성할 때만 사용한다. | |
| unique nullable insertable updatable columnDefinition table | @Column의 속성과 같다. | |

### 5.1.5 @ManyToOne 172p

- 다대일 관계에서 사용

<속성>

targetEntity 속성 사용 예시

| 속성 | 설명 | 기본값 |
| --- | --- | --- |
| optional | false로 설정하면 연관된 엔티티가 항상 있어야 한다. | true |
| fetch | 글로벌 페치 전략을 설정한다. (8장 참고) | @ManyToOne=FetchType.EAGER<br>@OneToMany=FetchType.LAZY |
| cascade | 속성 전이 기능을 사용한다. (8장 참고) | |
| targetEntity | 연관된 엔티티의 타입 정보를 설정한다. 이 기능은 거의 사용하지 않는다. 컬렉션을 사용해도 제네릭으로 타입 정보를 알 수 있다. | |

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

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

## 5.2 연관관계 사용

### 5.2.1 저장 173p

- 연관관계를 매핑한 엔티티 저장하는 방법

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

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

💡 JPA에서 엔티티를 저장할 때 연관된 모든 엔티티는 영속 상태여야 한다.

JPA는 참조한 팀의 식별자(Team.id 인 team1)를 외래키로 사용해서 등록 쿼리를 생성한다.

```
INSERT INTO TEAM (TEAM_ID, NAME) VALUES ('team1', '팀1');
INSERT INTO MEMBER (MEMBER_ID, NAME, TEAM_ID) VALUES ('member1', '회원1', 'team1'); # 회원 테이블의 외래키 값으로 참조한 팀의 식별자인 team1이 입력
INSERT INTO MEMBER (MEMBER_ID, NAME, TEAM_ID) VALUES ('member2', '회원2', 'team1');
```

### 5.2.2 조회 175p

연관관계가 있는 엔티티 조회하는 방법

1. 객체 그래프 탐색(**객체 연관관계를 사용한 조회**)
2. **객체지향 쿼리 사용** JPQL

예제 : 위에서 저장한 대로 회원1, 회원2가 팀1에 소속해 있다고 가정

1. **객체 그래프 탐색** (8장 참고)

   `member.getTeam()` 을 사용해서 member와 연관된 team 엔티티 조회

   ```
   Member member = em.find(Member.class, "member1");
   Team team = member.getTeam(); // 객체 그래프 탐색
   System.out.println("팀 이름 = " + team.getName()); // 팀 이름 = 팀1
   ```

   - 객체 그래프 탐색 : 객체를 통해 엔티티를 조회하는 것

2. **객체지향 쿼리 사용** (10장 참고)

- 객체지향 쿼리인 JPQL에서 연관관계를 어떻게 사용할까?
  - 팀 1에 소속된 회원만 조회하려면 회원과 연관된 팀 엔티티를 검색 조건으로 사용해야 한다.
  - SQL은 연관된 테이블을 조인해서 검색조건을 사용하면 된다.
  - JPQL도 조인을 지원한다(문법은 약간 다름).

- 팀1에 소속된 모든 회원 조회

```
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
[query] member.username = 회원2
```

- `from Member m join m.team t` ⇒ 회원이 팀과 관계를 가지고 있는 필드(`m.team`)를 통해서 Member와 Team을 조인함
- `:teamName` ⇒ : 로 시작하는 것은 파라미터를 바인딩하는 문법

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'
```

### 5.2.3 수정 177p

- 팀1 소속이던 회원을 새로운 팀2에 소속되도록 수정

```
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 SQL

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

### 5.2.3 연관관계 제거 177p

- 회원1을 팀에 소속하지 않도록 변경

```
private static void deleteRelation(EntityManager em) {
    //
    Member member1 = em.find(Member.class, "member1");
    member1.setTeam(null); // 연관관계 제거
}
```

실행되는 SQL

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

### 5.2.4 연관된 엔티티 삭제 178p

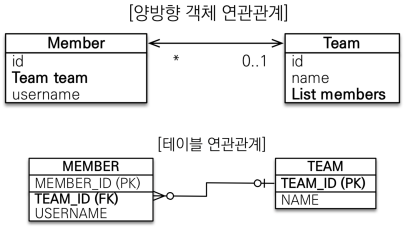
- 연관된 엔티티를 삭제하려면 기존에 있던 연관관계를 먼저 제거하고 삭제해야 한다.
- 그렇지 않으면 외래 키 제약조건으로 인해 데이터베이스에서 오류가 발생한다.

- 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
```

## 5.3 양방향 연관관계 178p

- 위에서는 회원에서 팀으로만 접근하는 다대일 단방향 매핑을 알아봤다
- 이번에는 반대 방향인 팀에서 회원으로 접근하는 관계를 추가
- 회원 → 팀, 팀 → 회원 접근할 수 있도록 양방향 연관관계로 매핑하면 아래와 같아




[객체 연관관계]

- 회원과 팀은 다대일 관계
  - 회원 → 팀 `Member.team`
- 팀에서 회원은 일대다 관계
  - 팀 → 회원 `Team.member`

[테이블 연관관계]

- 데이터베이스 테이블은 외래키 하나로 양방향으로 조회할 수 있다.
- 외래키(TEAM_ID)를 사용해서 `MEMBER JOIN TEAM` 이 가능하고 반대로 `TEAM JOIN MEMBER` 도 가능

### 5.3.1 양방향 연관관계 매핑 180p

```java
@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
}
```

```java
@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 관계이다
  - 팀 엔티티에 컬렉션인 members 추가
- 일대다 관계를 매핑하기 위해 `@OneToMany` 매핑정보를 사용
  - `mappedBy` 속성
    - 양방향 매핑일 때 사용.
    - 반대쪽 매핑의 필드 이름을 값으로 주면 된다.

### 5.3.2 일대다 컬렉션 조회 181p

- 팀에서 회원 컬렉션으로 객체 그래프 탐색을 사용해서 조회한 회원을 출력

```java
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
```

21.12.23 ~

## 5.4 연관관계의 주인 181p

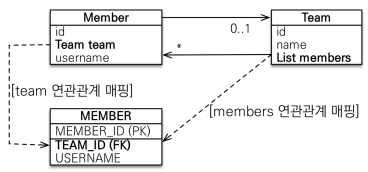
- 테이블은 외래키 하나로 두 테이블의 연관관계를 관리한다.
- **엔티티를 단방향으로 매핑하면 참조를 하나만 사용**하므로 이 참조로 외래키를 관리하면 된다.
- 엔티티를 양방향 연관관계로 설정하면 객체의 참조는 둘(회원 → 팀, 팀 → 회원)인데 외래 키는 하나다.
- JPA에서는 **두 객체 연관관계 중 하나를 정해서 테이블의 외래키를 관리해야 하는데, 이것을 연관관계의 주인**이라 한다.

### 5.4.1 양방향 매핑의 규칙 : 연관관계의 주인 182p

- **연관관계의 주인만이 데이터베이스 연관관계와 매핑되고 외래 키를 관리(등록, 수정, 삭제)할 수 있다.**
- 주인은 `mappedBy` 속성을 사용하지 않는다.
- 주인이 아니면 `maapedBy` 속성을 사용해서 속성의 값으로 연관관계의 주인을 지정해야 한다.
- **연관관계의 주인을 정한다는 것은 외래 키 관리자를 선택하는 것**이다.

둘 중 하나를 연관관계의 주인으로 선택해야 한다.

회원 → 팀( `Member.team` ) 방　　팀 → 회원( `Team.members` ) 방향




향

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

```
class Team {
    //
    @OneToMany
    private List<Member> members = new ArrayList<Meml
}
```

▼ `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.

  ⇒ 이 경우 `Team.members` 가 있는 Team 엔티티는 TEAM 테이블에 매핑되어 있는데 관리해야할 외래 키는 MEMBER 테이블에 있기 때문

### 5.4.2 연관관계의 주인은 외래 키가 있는 곳 184p

- 연관관계의 주인은 테이블에 외래 키가 있는 곳으로 정해야 한다.
  - 예제에서는 회원 테이블이 외래 키를 가지고 있으므로 `Member.team` 이 주인이 된다.
  - 주인이 아닌 `Team.members` 에는 `mappedBy = "team"` 속성을 사용해서 주인이 아님을 설정한다.
- 연관관계의 주인만 데이터베이스 연관관계와 매핑되고 외래 키를 관리할 수 있다.
- 주인이 아닌 반대편은 읽기만 가능하고 외래 키를 변경하지는 못한다.

```
class Team {
    //
    @OneToMany(mappedBy="team") // mappedBy 속성의 값인 team은 연관관계의 주인인 Member
    private List<Member> members = new ArrayList<Member>();
}
```

💡 **데이터베이스 테이블의 다대일, 일대다 관계에서는 항상 다 쪽이 외래키를 갖는다.**
다 쪽인 `@ManyToOne` 은 항상 연관관계의 주인이 되므로 `mappedBy` 를 설정할 수 없다. ⇒ `@ManyToOne` 에는 `mappedBy` 속성이 없다.

## 5.5 양방향 연관관계 저장 185p

```
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);
}
```

SELECT * FROM MEMBER;

| MEMBER_ID | USERNAME | TEAM_ID |
|---|---|---|
| member1 | 회원1 | team1 |
| member2 | 회원2 | team1 |

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

- 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 => 주인이 아닌 곳에 입력된 값은 외래 키에 영향을 주지 않아, 데이터베이스에 저장할 때 무시된다.
team1.getMembers().add(member2); // 무시
```

- 연관관계 설정

```
member1.setTeam(team1); // 연관관계 설정 => 엔티티 매니저는 연관관계의 주인인 Member.team에 입력된 값을 사용해서 외래 키를 관리한다.
member2.setTeam(team1);
```

## 5.6 양방향 연관관계의 주의점

- 양방향 연관관계를 설정하고 연관관계의 주인에는 값을 입력하지 않고, 주인이 아닌 곳에만 값을 입력하는 실수 주의!

```
public void testSaveNonOwner() {
    //
    // 팀1 저장
    Team team1 = new Team("team1", "팀1");
    em.persist(team1);

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

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

    em.persist(team1);
}
```

SELECT * FROM MEMBER;

| MEMBER_ID | USERNAME | TEAM_ID |
|---|---|---|
| member1 | 회원1 | null |
| member2 | 회원2 | null |

- 연관관계의 주인이 아닌 `Team.members` 에만 값을 저장했기 때문에 `TEAM_ID` 에 `team1` 이 아닌 null 값이 저장된 것.

### 5.6.1 순수한 객체까지 고려한 양방향 연관관계 187p

- 정말 연관관계 주인에만 값을 저장하고 주인이 아닌 곳에 값을 저장하지 않아도 되나?
  - 사실 객체 관전에서 양쪽 방향에 모두 값을 입력해주는 것이 가장 안전하다.
    - 양쪽 방향 모두 값을 입력하지 않으면 JPA를 사용하지 않는 순수한 객체 상태에서 심각한 문제가 발생할 수 있다.

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

JPA를 사용

```
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 -> memb
    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 -> memb
    em.persist(member2);
}
```

### 5.6.2 연관관계 편의 메소드 190p

### 5.6.3 연관관계 편의 메소드 작성 시 주의사항 191p

- `member1.setTeam(team1)` 과 `team1.getMembers().add(member1)` 를 각각 호출하면 실수할 수 있어서 위험
- 양방향 관계에서 두 코드는 아래처럼 하나인 것처럼 사용하는 것이 안전!

```
public class Member {
    //
    private Team team;

    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);
    }
}
```

```
public void testORM_양방향() {
    //
    // 팀1 저장
    Team team1 = new Team("team1", "팀1");
    em.persist(team1);

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

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

//    team1.getMembers().add(member1); 삭제
//    team1.getMembers().add(member2); 삭제
}
```

기존 팀과 관계를 제거 부분을 넣지 않으면

- `member1.setTeam(team1)` 을 호출한 직후 객체 연관관계는 member1과 team1가 양방향 관계가 된다.
- 다음으로 `member1.setTeam(team2)` 을 호출한 직후 member1과 team2가 양방향 관계가 되고, 기존에 team1 → member1 관계는 제거되지 않는다.

```
member1.setTeam(team1);
member1.setTeam(team2);
Member findMember = team1.getMember(); // 이때 여전히 member1이 조회된다.
```

⇒ 따라서 기존 관계를 제거하는 코드를 넣어야 한다.

💡 이후에 새로운 영속성 컨텍스트에서 team1을 조회해서 `team1.getMembers()` 를 호출하면 데이터베이스 외래 키에는 관계가 끊어져 있으므로 아무것도 조회되지 않는다.
하지만 관계를 변경하고 영속성 컨텍스트가 아직 살아있는 상태에서는 member1이 반환되므로 관계를 제거하는 것이 안전하다. (193p 참고)

## 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.

- 단방향은 항상 외래 키가 있는 곳을 기준으로 매핑하면 된다.
- 양방향은 외래 키가 있는 다 쪽이 연관관계의 주인이 된다.