# 5장 연관관계 매핑 기초 - 12월 3,4주차

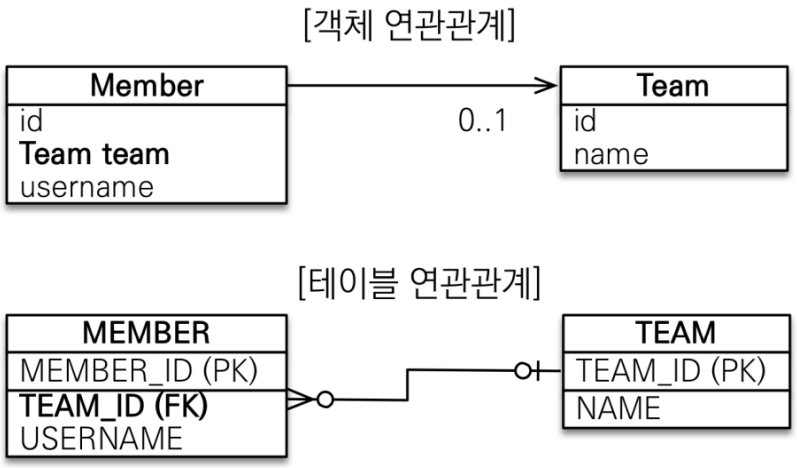
기간 : 21.12.17 ~ 21.12.23

페이지 : 122 ~ 181p (4장 ~ 5장 5.3)

▼ 목차



## 5.1 단방향 연관관계 164p




**객체 연관관계**

- 회원 객체는 `Member.team` 필드(멤버변수)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.
- 즉 **외래 키 하나**로 두 테이블의 연관관계를 관리한다.
- 회원 테이블과 팀 테이블은 **양방향 관계**다.

```
# 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
```

**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

```
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
}

public class Team {
    //
    private String id;
    private String name;

    // Getter, Setter
}
```

```
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(); // 회원1이 속한 팀1을 조회
}
```

- 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

회원 테이블과 팀 테이블의 DDL

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

CREATE TABLE TAEM (
    TEAM_ID VARCHAR(255) NOT NULL,
    NAME VARCHAR(255),
    PRIVATE KEY (TEAM_ID)
)
```

회원 테이블의 TEAM_ID에 외래 키 제약조건을 설정

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

팀1에 회원1, 회원2 소속시키기

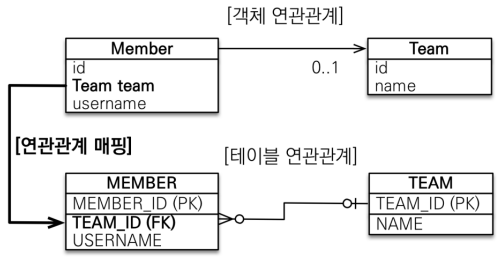
```
INSERT INTO TEAM(TEAM_ID, NAME) VALUES('team1', '팀1');
INSERT INTO MEMBER(MEMBER_ID, TEAM_ID, USERNAME) VALUES('member1', 'team1', '회원1');
INSERT INTO MEMBER(MEMBER_ID, TEAM_ID, USERNAME) VALUES('member2', 'team1', '회원2');
```

회원1이 소속된 팀 조회

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

### 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")
    private Team team;

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

    // Getter, Setter
}
```

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

    private String name;

    // 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>

| 속성 | 설명 | 기본값 |
|---|---|---|
| optional | false로 설정하면 연관된 엔티티가 항상 있어야 한다. | true |
| fetch | 글로벌 페치 전략을 설정한다. (8장 참고) | @ManyToOne=FetchType.EAGER @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. | |

targetEntity 속성 사용 예시

```
@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

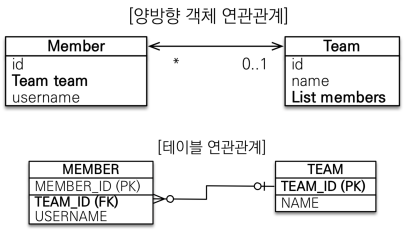
- 연관된 엔티티를 삭제하려면 기존에 있던 연관관계를 먼저 제거하고 삭제해야 한다.
- 그렇지 않으면 외래 키 제약조건으로 인해 데이터베이스에서 오류가 발생한다.

- 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

```
@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
}
```

```
@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

```
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
```