- **연관관계 매핑 - 실전**
  - **다대일**
    - - 다대일 단방향
    - - 다대일 양방향
  - **일대다**
    - - 일대다 단방향
    - - 일대다 양방향
  - **일대일**
    - - 주 테이블에 외래 키
      - – 단방향
      - – 양방향
    - - 대상 테이블에 외래 키
      - – 단방향
      - – 양방향
  - **다대다**
    - - 다대다 - 단방향
    - - 다대다 - 양방향
    - - 다대다 - 매핑의 한계와 극복, 연결 엔티티 사용
    - - 다대다 - 새로운 기본 키 사용
- **[실전 예제] - 3. 다양한 연관관계 매핑하기**
  - **일대일 매핑하기**
  - **다대다 매핑하기**

연관관계 매핑 - 실전

연관관계 매핑 이론장에서 설명한 내용을 다시 한번 정리해보자.

엔티티의 연관관계를 매핑할 때는 다음 3가지를 고려해야 한다.

- 다중성
- 단방향, 양방향
- 연관관계의 주인

먼저 연관관계가 있는 두 엔티티가 일대일 관계인지 일대다 관계인지 다중성을 고려해야 한다. 다음으로 두 엔티티 중 한쪽만 참조하는 단방향 관계인지 서로 참조하는 양방향 관계인지 고려해야 한다. 마지막으로 양방향 관계면 연관관계의 주인을 정해야 한다.

다중성

연관관계에는 다음과 같은 다중성이 있다.

- 다대일(`@ManyToOne`)
- 일대다(`@OneToMany`)
- 일대일(`@OneToOne`)
- 다대다(`@ManyToMany`)

다중성을 판단하기 어려울 때는 반대방향을 생각해보면 된다. 참고로 일대다의 반대방향은 항상 다대일이고, 일대일의 반대방향은 항상 일대일이다.

보통 다대일과 일대다 관계를 가장 많이 사용하고 다대다 관계는 실무에서 거의 사용하지 않는다.

단방향, 양방향

테이블은 외래 키 하나로 조인을 사용해서 양방향으로 쿼리가 가능하므로 사실상 방향이라는 개념이 없다. 반면에 객체는 참조용 필드를 가지고 있는 객체만 연관된 객체를 조회할 수 있다. 한쪽만 참조하는 것을 단방향 관계라 하고, 양쪽이 서로 참조하는 것을 양뱡향 관계라 한다.

연관관계의 주인

데이터베이스는 외래 키 하나로 두 테이블이 연관관계를 맺는다. 따라서 테이블의 연관관계를 관리하는 포인트는 외래 키 하나다. 반면에 엔티티를 양방향으로 매핑하면 `A -> B`, `B -> A` 2곳에서 서로를 참조한다. 따라서 객체의 연관관계를 관리하는 포인트는 2곳이다.

JPA는 두 객체 연관관계 중 하나를 정해서 데이터베이스 외래 키를 관리하는데 이것을 연관관계의 주인이라 한다. 따라서 `A -> B` 또는 `B -> A` 둘 중 하나를 정해서 외래 키를 관리해야 한다. 외래 키를 가진 테이블과 매핑한 엔티티가 외래 키를 관리하는게 효율적이므로 보통 이곳을 연관관계의 주인으로 선택한다. 주인이 아닌 방향은 외래 키를 변경할 수 없고 읽기만 가능하다.

연관관계의 주인은 `mappedBy` 속성을 사용하지 않는다. 연관관계의 주인이 아니면 `mappedBy` 속성을 사용하고 연관관계의 주인 필드 이름을 값으로 입력해야 한다.

지금부터 다중성과 단방향, 양방향을 고려한 가능한 모든 연관관계를 하나하나 알아보자.

- 다대일: 단방향, 양방향
- 일대다: 단방향, 양방향
- 일대일: 주 테이블 단방향, 양방향
- 일대일: 대상 테이블 단방향, 양방향
- 다대다: 단방향, 양방향

참고로 다중성은 왼쪽을 연관관계의 주인으로 정했다. 예를 들어 다대일 양방향이라 하면 다(N)가 연관관계의 주인이다.

> **참고:** 이번 장을 진행하면서 웹 애플리케이션 만들기 장에 있는 2. 도메인 모델과 테이블 설계를 함께 보자. 이 예제는 다양한 연관관계를 포함하므로 이번 장에서 학습한 내용을 실제 어떻게 적용해야 할지 이해하는 데 도움을 준다.

다대일

다대일 관계의 반대 방향은 항상 일대다 관계고 일대다 관계의 반대 방향은 항상 다대일 관계다. 데이터베이스 테이블의 일(1), 다(N) 관계에서 외래 키는 항상 다쪽에 있다. 따라서 객체 양방향 관계에서 연관관계의 주인은 항상 다쪽이다. 예를 들어 회원(N)과 팀(1)이 있으면 회원 쪽이 연관관계의 주인이다.

- 다대일 단방향

다대일 단방향 연관관계를 알아보자.

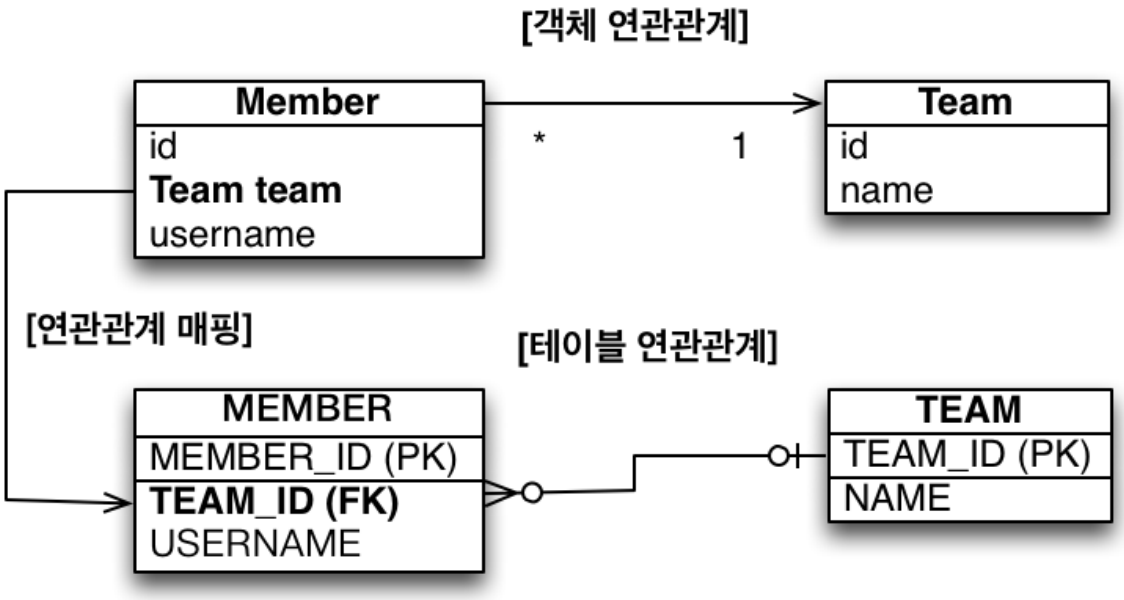

그림 - 다대일 단방향

======회원 엔티티======

```
@Entity
public class Member {

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

    private String username;

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

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

====== 팀 엔티티======

```
@Entity
public class Team {

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

```
    private String name;

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

회원은 `Member.team`으로 팀 엔티티를 참조할 수 있지만 반대로 팀에는 회원을 참조하는 필드가 없다. 따라서 회원과 팀은 다대일 단방향 연관관계다.

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

`@JoinColumn(name = "TEAM_ID")`을 사용해서 `Member.team` 필드를 `TEAM_ID` 외래 키와 매핑했다. 따라서 `Member.team` 필드로 회원 테이블의 `TEAM_ID` 외래 키를 관리한다.

- 다대일 양방향

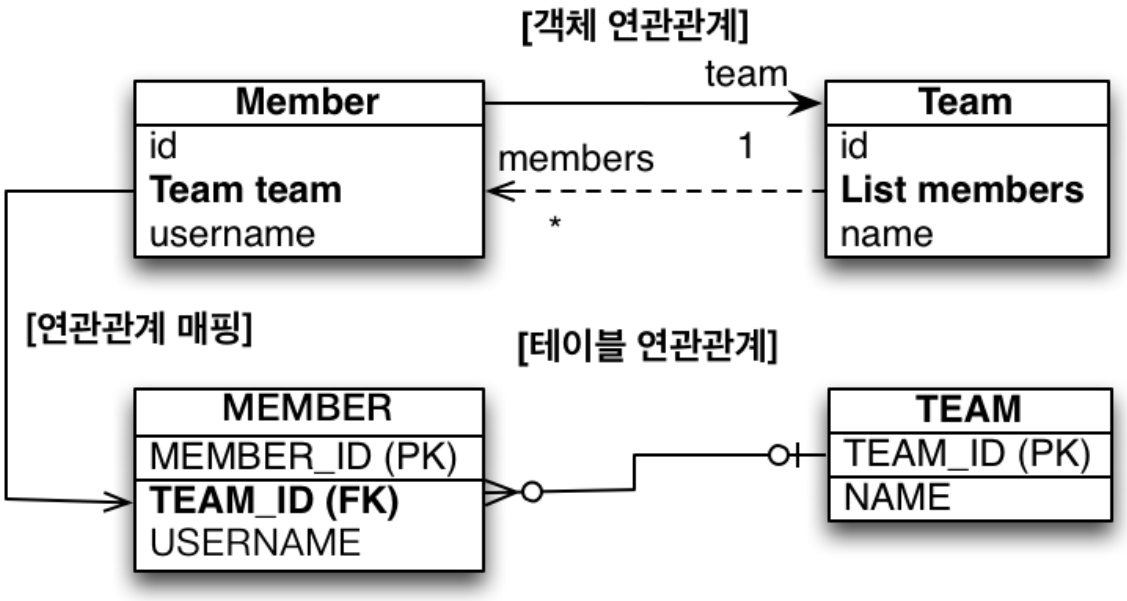

그림 - 다대일 양방향

그림 - 다대일 양방향의 객체 연관관계에서 실선이 연관관계의 주인(`Member.team`)이고 점선(`Team.members`)은 연관관계의 주인이 아니다.

======회원 엔티티======

```
@Entity
public class Member {

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

    private String username;

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

    public void setTeam(Team team) {
        this.team = team;
        if (!team.getMembers().contains(this)) { //무한루프에 빠지지 않도록 체크
            team.getMembers().add(this);
        }
    }
```

======팀 엔티티======

```
@Entity
public class Team {

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

    private String name;

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

    public void addMember(Member member) {
        this.members.add(member);
        if (member.getTeam() != this) { //무한루프에 빠지지 않도록 체크
            member.setTeam(this);
        }
    }
```

양방향은 외래 키가 있는 쪽이 연관관계의 주인이다.

일대다와 다대일 연관관계는 항상 다(N)에 외래 키가 있다. 여기서는 다쪽인 `MEMBER` 테이블이 외래 키를 가지고 있으므로 `Member.team`이 연관관계의 주인이다. JPA는 외래 키를 관리할 때 연관관계의 주인만 사용한다. 주인이 아닌 `Team.members`는 조회를 위한 JPQL이나 객체 그래프를 탐색할 때 사용한다.

양방향 연관관계는 항상 서로를 참조해야 한다.

양방향 연관관계는 항상 서로 참조해야 한다. 어느 한 쪽만 참조하면 양방향 연관관계가 성립하지 않는다. 항상 서로 참조하게 하려면 연관관계 편의 메서드를 작성하는 것이 좋은데 회원의 `setTeam()`, 팀의 `addMember()` 메서드가 이런 편의 메서드들이다. 편의 메서드는 한 곳에만 작성하거나 양쪽 다 작성할 수 있는데, 양쪽에 다 작성하면 무한루프에 빠지므로 주의해야 한다. 예제 코드는 편의 메서드를 양쪽에 다 작성해서 둘 중 하나만 호출하면 된다. 또한 무한루프에 빠지지 않도록 검사하는 로직도 있다.

일대다

일대다 관계는 다대일 관계의 반대 방향이다. 일대다 관계는 엔티티를 하나 이상 참조할 수 있으므로 자바 컬렉션인 `Collection`, `List`, `Set`, `Map` 중에 하나를 사용해야 한다.

- 일대다 단방향

하나의 팀은 여러 회원을 참조할 수 있는데 이런 관계를 일대다 관계라 한다. 그리고 팀은 회원들은 참조하지만 반대로 회원은 팀을 참조하지 않으면 둘의 관계는 단방향이다. [그림 - 일대다 단방향]을 통해 일대다 단방향 관계를 알아보자. (일대다 단방향 관계는 JPA 2.0부터 지원한다.)

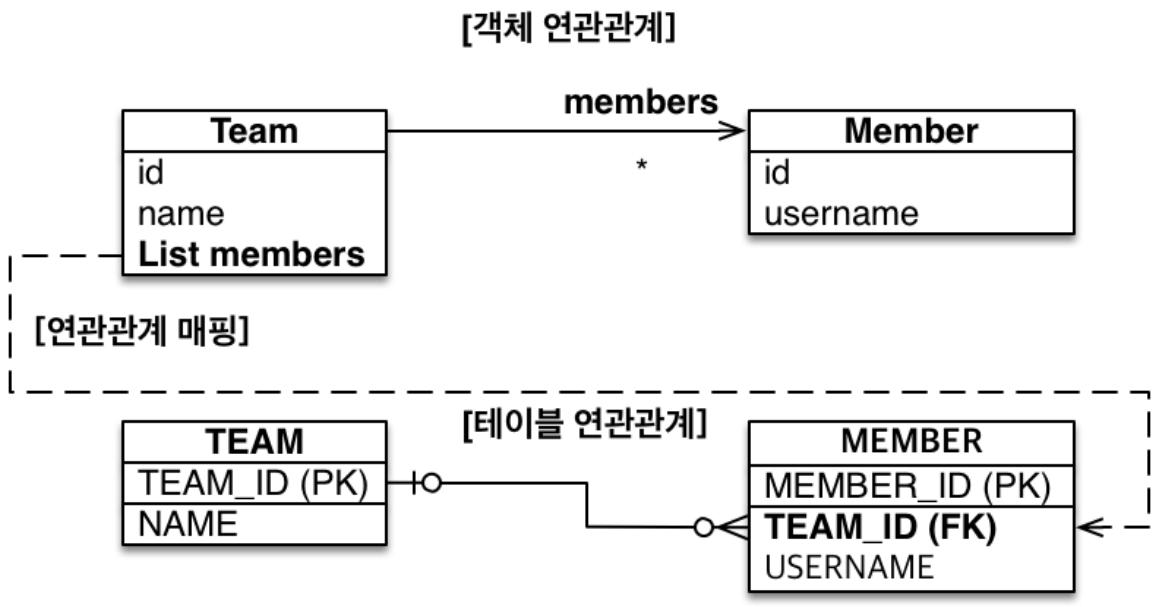

그림 - 일대다 단방향

일대다 단방향 관계는 약간 특이한데 그림을 보면 팀 엔티티의 `Team.members`로 회원 테이블의 `TEAM_ID` 외래 키를 관리한다. 보통 자신이 매핑한 테이블의 외래 키를 관리하는데 이 매핑은 반대쪽 테이블에 있는 외래 키를 관리한다.

그럴 수밖에 없는 것이 일대다 관계에서 외래 키는 항상 다 쪽 테이블에 있다. 하지만 다 쪽인 `Member` 엔티티에는 외래 키를 매핑할 수 있는 참조 필드가 없다. 대신에 반대쪽인 `Team` 엔티티에만 참조 필드인 `members`가 있다. 따라서 반대편 테이블의 외래 키를 관리하는 특이한 모습이 나타난다.

====== 일대다 단방향 팀 엔티티 ======

```
@Entity
public class Team {

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

    private String name;

    @OneToMany
    @JoinColumn(name = "TEAM_ID") //MEMBER 테이블의 TEAM_ID (FK) //**
    private List<Member> members = new ArrayList<Member>();

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

====== 일대다 단방향 회원 엔티티 ======

```
@Entity
public class Member {

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

    private String username;

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

일대다 단방향 관계를 매핑할 때는 `@JoinColumn`을 명시해야 한다. 그렇지 않으면 JPA는 연결 테이블을 중간에 두고 연관관계를 관리하는 조인 테이블(JoinTable) 전략을 기본으로 사용해서 매핑한다. 조인 테이블에 대한 자세한 내용은 “8. 연관관계 매핑 심화2” 장에서 알아보겠다.

일대다 단방향 매핑의 단점

일대다 단방향 매핑의 단점은 매핑한 객체가 관리하는 외래 키가 다른 테이블에 있다는 점이다. 본인 테이블에 외래 키가 있으면 엔티티의 저장과 연관관계 처리를 INSERT SQL 한 번으로 끝낼 수 있지만, 다른 테이블에 외래 키가 있으면 연관관계 처리를 위한 UPDATE SQL을 추가로 실행해야 한다.

예제를 보자.

====== 일대다 단방향 매핑의 단점 ======

```
public void testSave() {

    Member member1 = new Member("member1");
    Member member2 = new Member("member2");

    Team team1 = new Team("team1");
    team1.getMembers().add(member1);
    team1.getMembers().add(member2);

    em.persist(member1);//INSERT-member1
    em.persist(member2);//INSERT-member2
    em.persist(team1);//INSERT-team1, UPDATE-member1.fk, UPDATE-member2.fk
    transaction.commit();
}
```

======실행된 SQL======

```
insert into Member (MEMBER_ID, username) values (null, ?)
insert into Member (MEMBER_ID, username) values (null, ?)
insert into Team (TEAM_ID, name) values (null, ?)
update Member set TEAM_ID=? where MEMBER_ID=?
update Member set TEAM_ID=? where MEMBER_ID=?
```

`Member` 엔티티는 `Team` 엔티티를 모른다. 그리고 연관관계에 대한 정보는 `Team` 엔티티의 `members`가 관리한다. 따라서 `Member` 엔티티를 저장할 때는 `MEMBER` 테이블의 `TEAM_ID` 외래 키에 아무 값도 저장되지 않는다. 대신 `Team` 엔티티를 저장할 때 `Team.members`의 참조 값을 확인해서 회원 테이블에 있는 `TEAM_ID` 외래 키를 업데이트 한다.

일대다 단방향 매핑보다는 다대일 양방향 매핑을 사용하자

일대다 단방향 매핑을 사용하면 엔티티를 매핑한 테이블이 아닌 다른 테이블의 외래 키를 관리해야 한다. 이것은 성능 문제도 있지만 관리도 부담스럽다. 문제를 해결하는 좋은 방법은 일대다 단방향 매핑 대신에 다대일 양방향 매핑을 사용하는 것이다. 다대일 양방향 매핑은 관리해야 하는 외래 키가 본인 테이블에 있다. 따라서 일대다 단방향 매핑 같은 문제가 발생하지 않는다. 두 매핑의 테이블 모양은 완전히 같으므로 엔티티만 약간 수정하면 된다. 상황에 따라 다르겠지만 일대다 단방향 매핑보다는 다대일 양방향 매핑을 권장한다.

- 일대다 양방향

일대다 양방향 매핑은 존재하지 않는다. 대신 다대일 양방향 매핑을 사용해야 한다. (일대다 양방향과 다대일 양방향은 사실 똑같은 말이다. 여기서는 왼쪽을 연관관계의 주인으로 가정해서 분류했다. 예를 들어 다대일이면 다(N)가 연관관계의 주인이다.)

더 정확히 말하자면 양방향 매핑에서 `@OneToMany`는 연관관계의 주인이 될 수 없다. 왜냐하면 관계형 데이터베이스의 특성상 일대다, 다대일 관계는 항상 다 쪽에 외래 키가 있다. 따라서 `@OneToMany`, `@ManyToOne` 둘 중에 연관관계의 주인은 항상 다 쪽인 `@ManyToOne`을 사용한 곳이다. 이런 이유로 `@ManyToOne`에는 `mappedBy` 속성이 없다.

그렇다고 일대다 양방향 매핑이 완전히 불가능한 것은 아니다. 일대다 단방향 매핑 반대편에 같은 외래 키를 사용하는 다대일 단방향 매핑을 읽기 전용으로 하나 추가하면 된다.

다음 그림과 코드를 보자.

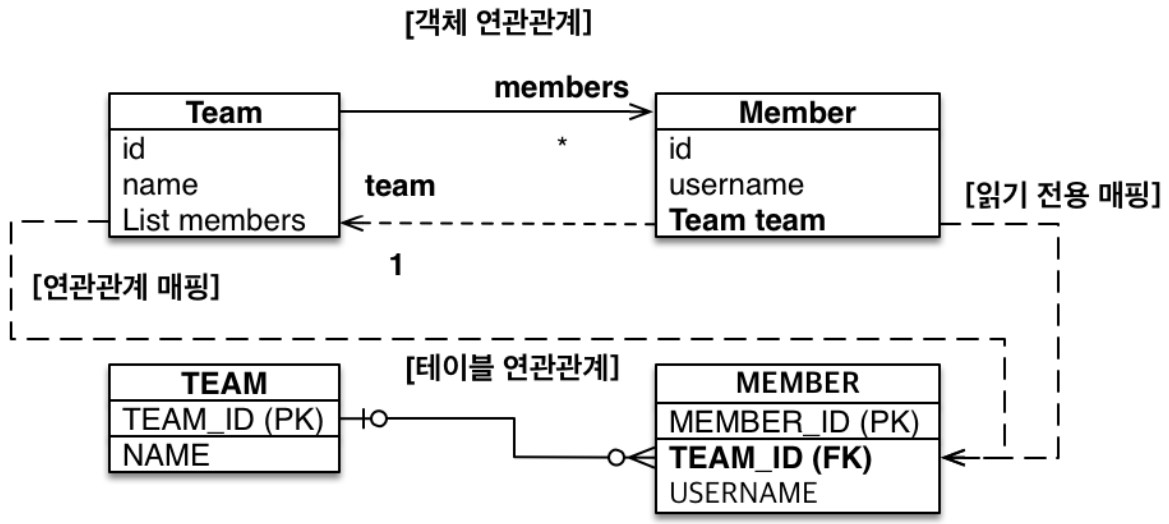

그림 - 일대다 단방향

====== 일대다 양방향 팀 엔티티 ======

```
@Entity
public class Team {

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

    private String name;

    @OneToMany
    @JoinColumn(name = "TEAM_ID") //**
    private List<Member> members = new ArrayList<Member>();

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

====== 일대다 양방향 회원 엔티티 ======

```
@Entity
public class Member {

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

    @ManyToOne
    @JoinColumn(name = "TEAM_ID", insertable = false, updatable = false) //**
    private Team team;

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

일대다 단방향 매핑 반대편에 다대일 단방향 매핑을 추가했다. 이때 일대다 단방향 매핑과 같은 `TEAM_ID` 외래 키 컬럼을 매핑했다. 이렇게 되면 둘다 같은 키를 관리하므로 문제가 발생할 수 있다. 따라서 반대편인 다대일 쪽은 `insertable = false, updatable = false` 로 설정해서 읽기만 가능하게 했다.

이 방법은 일대다 양방향 매핑이라기보다는 일대다 단방향 매핑 반대편에 다대일 단방향 매핑을 읽기 전용으로 추가해서 일대다 양방향처럼 보이도록 하는 방법이다. 따라서 일대다 단방향 매핑이 가지는 단점을 그대로 가진다. 될 수 있으면 다대일 양방향 매핑을 사용하자.

일대일

일대일 관계는 양쪽이 서로 하나의 관계만 가진다. 예를 들어 회원은 하나의 사물함만 사용하고 사물함도 하나의 회원에 의해서만 사용된다.

일대일 관계는 다음과 같은 특징이 있다.

- 일대일 관계는 그 반대도 일대일 관계다.
- 테이블 관계에서 일대다, 다대일은 항상 다(N)쪽이 외래 키를 가진다. 반면에 일대일 관계는 주 테이블이나 대상 테이블 둘 중 어느 곳이나 외래 키를 가질 수 있다.

테이블은 주 테이블이든 대상 테이블이든 외래 키 하나만 있으면 양쪽으로 조회할 수 있다. 그리고 일대일 관계는 그 반대쪽도 일대일 관계다. 따라서 일대일 관계는 주 테이블이나 대상 테이블 중에 누가 외래 키를 가질지 선택해야 한다.

주 테이블에 외래 키

주 객체가 대상 객체를 참조하는 것처럼 주 테이블에 외래 키를 두고 대상 테이블을 참조한다. 외래 키를 객체 참조와 비슷하게 사용할 수 있어서 객체지향 개발자들이 선호한다. 이 방법의 장점은 주 테이블이 외래 키를 가지고 있으므로 주 테이블만 확인해도 대상 테이블과 연관관계가 있는지 알 수 있다.

대상 테이블에 외래 키

전통적인 데이터베이스 개발자들은 보통 대상 테이블에 외래 키를 두는 것을 선호한다. 이 방법의 장점은 테이블 관계를 일대일에서 일대다로 변경할 때 테이블 구조를 그대로 유지할 수 있다.

이제 모든 일대일 관계를 하나씩 살펴보자.

- 주 테이블에 외래 키

일대일 관계를 구성할 때 객체지향 개발자들은 주 테이블에 외래 키가 있는 것을 선호한다. JPA도 주 테이블에 외래 키가 있으면 좀 더 편리하게 매핑할 수 있다.

주 테이블에 외래 키가 있는 단방향 관계를 먼저 살펴보고 양방향 관계도 살펴보자.

– 단방향

회원과 사물함의 일대일 단방향 관계를 알아보자. `MEMBER`가 주 테이블이고 `LOCKER`는 대상 테이블이다.

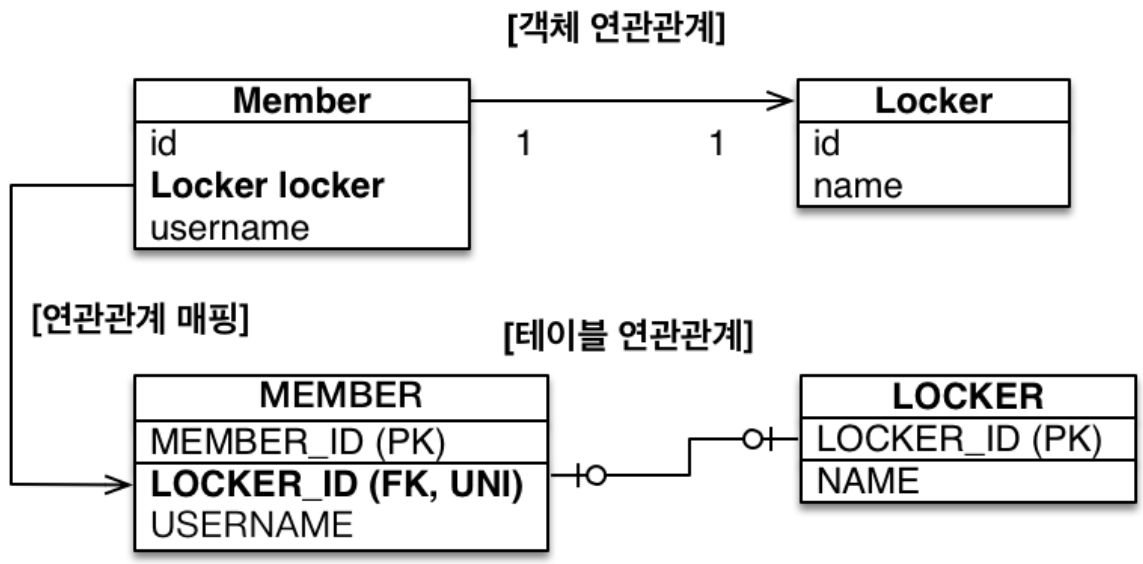

그림 - 일대일 주 테이블에 외래 키, 단방향

====== 일대일 주 테이블에 외래 키, 단방향 예제 코드 ======

```
@Entity
public class Member {

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

    private String username;

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

```
@Entity
public class Locker {

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

    private String name;
    ...
}
```

일대일 관계이므로 객체 매핑에 `@OneToOne`을 사용했고 데이터베이스에는 `LOCKER_ID` 외래 키에 유니크 제약 조건(UNI)을 추가했다. 참고로 이 관계는 다대일 단방향(`@ManyToOne`)과 거의 비슷하다.

다음으로 반대 방향을 추가해서 일대일 양방향 관계로 만들어보자.

– 양방향

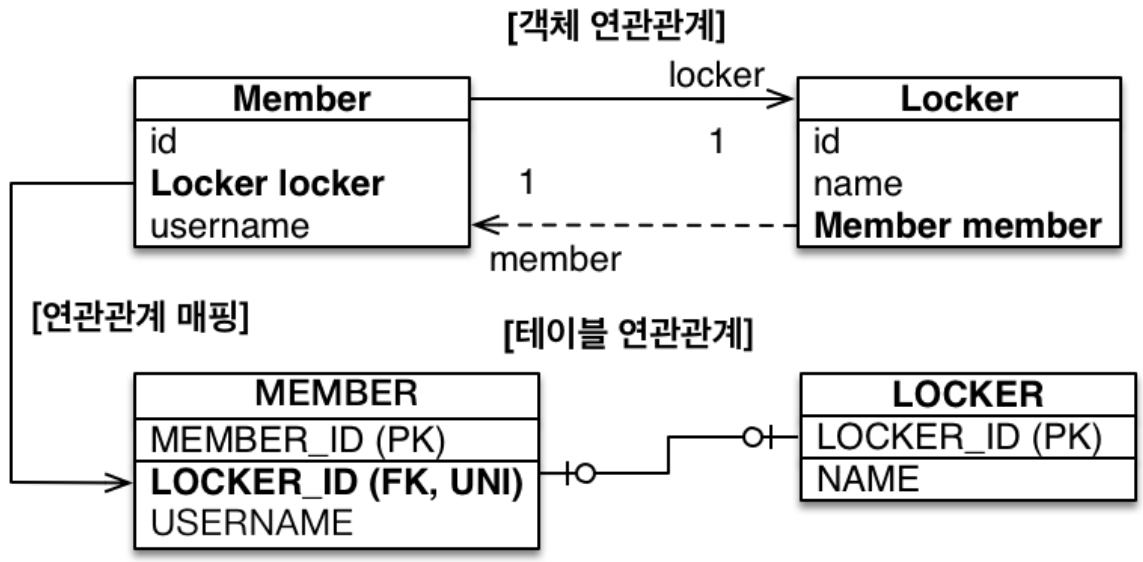

그림 - 일대일 주 테이블에 외래 키, 양방향

====== 일대일 주 테이블에 외래 키, 양방향 예제 코드 ======

```
@Entity
public class Member {

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

    private String username;

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

```
@Entity
public class Locker {
```

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

        private String name;

        @OneToOne(mappedBy = "locker")
        private Member member;
        ...
    }
```

양방향이므로 연관관계의 주인을 정해야 한다. `MEMBER` 테이블이 외래 키를 가지고 있으므로 `Member` 엔티티에 있는 `Member.locker` 가 연관관계의 주인이다. 따라서 반대 매핑인 사물함의 `Locker.member` 는 `mappedBy` 를 선언해서 연관관계의 주인이 아니라고 설정했다.

- 대상 테이블에 외래 키

– 단방향

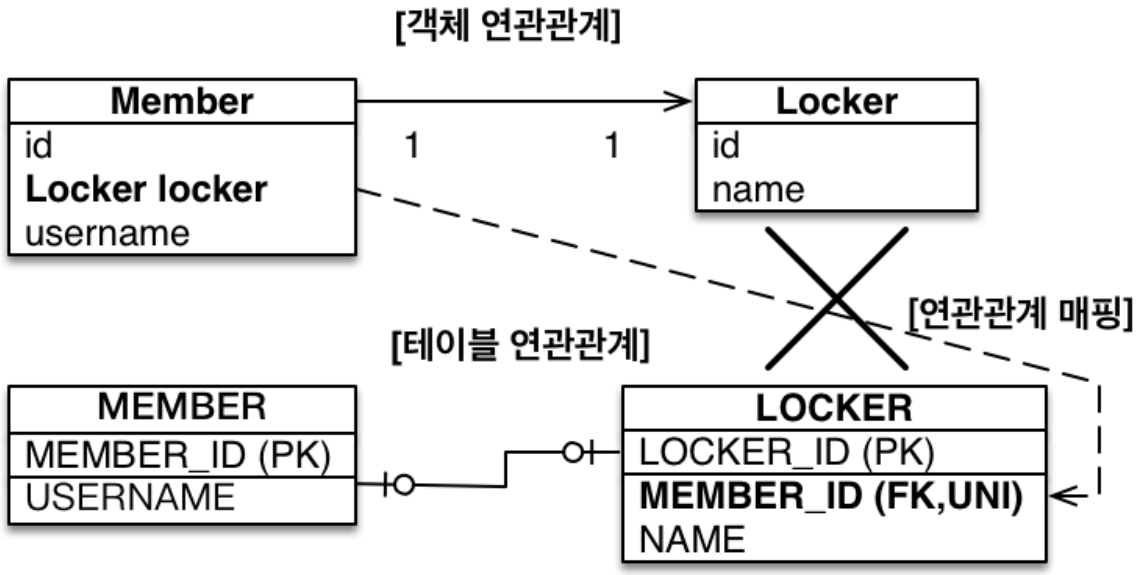

그림 - 일대일 대상 테이블에 외래 키, 단방향

일대일 관계 중 대상 테이블에 외래 키가 있는 단방향 관계는 JPA에서 지원하지 않는다.(이런 모양으로 매핑할 수 있는 방법도 없다.) 이때는 단방향 관계를 `Locker` 에서 `Member` 방향으로 수정하거나, 양방향 관계로 만들고 `Locker` 를 연관관계의 주인으로 설정해야 한다. 이 방법은 다음 양방향에서 알아보자.

참고로 JPA2.0부터 일대다 단방향 관계에서 대상 테이블에 외래 키가 있는 매핑을 허용했다. 하지만 일대일 단방향은 이런 매핑을 허용하지 않는다.

– 양방향

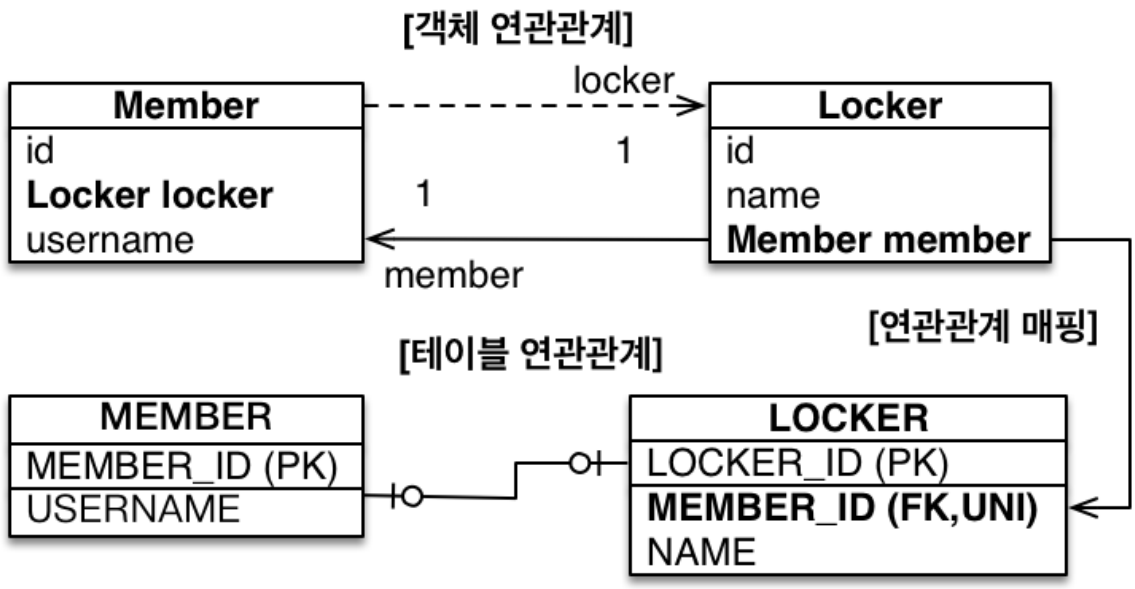

그림 - 일대일 대상 테이블에 외래 키, 양방향

====== 일대일 대상 테이블에 외래 키, 양방향 예제 코드 ======

```
@Entity
public class Member {

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

    private String username;

    @OneToOne(mappedBy = "member")
    private Locker locker;
    ...
}
```

```
@Entity
public class Locker {

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

    private String name;
```

```
    @OneToOne
    @JoinColumn(name = "MEMBER_ID")
    private Member member;
    ...
}
```

일대일 매핑에서 대상 테이블에 외래 키를 두고 싶으면 이렇게 양방향으로 매핑한다. 주 엔티티인 `Member` 엔티티 대신에 대상 엔티티인 `Locker`를 연관관계의 주인으로 만들어서 `LOCKER` 테이블의 외래 키를 관리하도록 했다.

> **참고:** 부모 테이블의 기본 키를 자식 테이블에서도 기본 키로 사용하는 일대일 관계는 다음 장의 일대일 식별 관계에서 설명한다.

> **주의:** 프록시를 사용할 때 외래 키를 직접 관리하지 않는 일대일 관계는 지연 로딩으로 설정해도 즉시 로딩된다. 예를 들어 방금 본 예제에서 `Locker.member`는 지연 로딩할 수 있지만, `Member.locker`는 지연 로딩으로 설정해도 즉시 로딩된다. 이것은 프록시의 한계 때문에 발생하는 문제인데 프록시 대신에 bytecode instrumentation을 사용하면 해결할 수 있다. 자세한 내용과 다양한 해결책은 다음 주소를 참고하자.
>
> 주소: `https://developer.jboss.org/wiki/SomeExplanationsOnLazyLoadingone-to-one`
>
> 지금은 프록시와 지연 로딩에 관해 설명하지 않았으므로 방금 설명한 내용을 이해하지 못해도 된다. 프록시와 지연 로딩은 8. 프록시와 연관관계 관리에서 알아보겠다.

다대다

관계형 데이터베이스는 정규화된 테이블 2개로 다대다 관계를 표현할 수 없다. 그래서 보통 다대다 관계를 일대다, 다대일 관계로 풀어내는 연결 테이블을 사용한다.

예를 들어 회원들은 원하는 상품을 주문한다. 반대로 상품들은 회원들에 의해 주문된다. 둘은 다대다 관계다. 따라서 회원 테이블과 상품 테이블만으로는 이 관계를 표현할 수 없다.

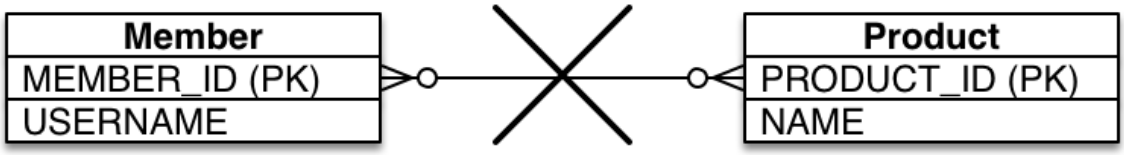

그림 3.15 I 테이블, N:M 다대다 불가능

그래서 중간에 연결 테이블을 추가해야 한다. 다음 그림을 보면 `Member_Product` 라는 연결 테이블을 추가했다. 이 테이블을 사용해서 다대다 관계를 일대다, 다대일 관계로 풀어 낼 수 있다. 이 연결 테이블은 회원이 주문한 상품을 나타낸다.

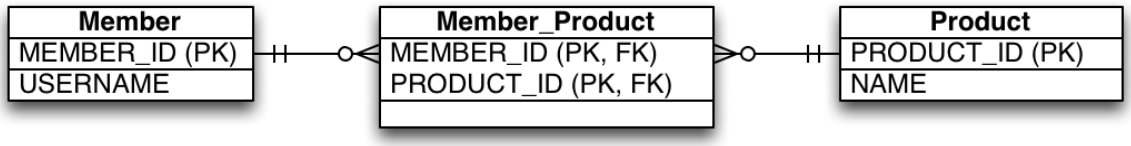

그림 3.16 I 테이블, N:M 다대다 연결테이블

그런데 객체는 테이블과 다르게 객체 2개로 다대다 관계를 만들 수 있다. 예를 들어 회원 객체는 컬렉션을 사용해서 상품들을 참조하면 되고 반대로 상품들도 컬렉션을 사용해서 회원들을 참조하면 된다.

`@ManyToMany` 를 사용하면 이런 다대다 관계를 편리하게 매핑할 수 있다.

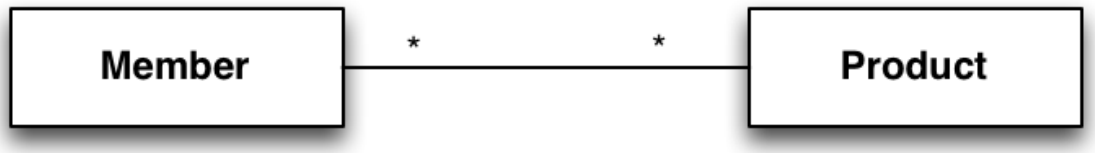

그림 3.17 I 클래스, 다대다 엔티티

- 다대다 - 단방향

====== 회원 ======

```
@Entity
public class Member {

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

    private String username;
```

```
    @ManyToMany
    @JoinTable(name = "MEMBER_PRODUCT",
            joinColumns = @JoinColumn(name = "MEMBER_ID"),
            inverseJoinColumns = @JoinColumn(name = "PRODUCT_ID"))
    private List<Product> products = new ArrayList<Product>();
    ...
}
```

====== 상품 ======

```
@Entity
public class Product {

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

    private String name;
    ...
}
```

회원 엔티티와 상품 엔티티를 `@ManyToMany` 로 매핑했다. 여기서 중요한 점은 `@ManyToMany` 와 `@JoinTable` 을 사용해서 연결 테이블을 바로 매핑한 것이다. 따라서 회원과 상품을 연결하는 회원_상품(`Member_Product`) 엔티티 없이 매핑을 완료할 수 있다.

`@JoinTable` 속성 정리

- `@JoinTable.name` : 연결 테이블을 지정한다. 여기서는 `MEMBER_PRODUCT` 테이블을 선택했다.
- `@JoinTable.joinColumns` : 현재 방향인 회원과 매핑할 조인 컬럼 정보를 지정한다. `MEMBER_ID` 로 지정했다.
- `@JoinTable.inverseJoinColumns` : 반대 방향인 상품과 매핑할 조인 컬럼 정보를 지정한다. `PRODUCT_ID` 로 지정했다.

`MEMBER_PRODUCT` 테이블은 다대다 관계를 일대다, 다대일 관계로 풀어내기 위해 필요한 연결 테이블일 뿐이다. `@ManyToMany` 로 매핑한 덕분에 다대다 관계를 사용할 때는 이 연결 테이블을 신경쓰지 않아도 된다.

다음으로 다대다 관계를 사용하는 코드를 보자.

====== 저장 ======

```
public void save() {

    Product productA = new Product();
    productA.setId("productA");
    productA.setName("상품A");
    em.persist(productA);

    Member member1 = new Member();
    member1.setId("member1");
    member1.setUsername("회원1");
    member1.getProducts().add(productA) //연관관계 설정
    em.persist(member1);

}
```

회원1과 상품A의 연관관계를 설정했으므로 회원1을 저장할 때 연결 테이블에도 값이 저장된다. 따라서 이 코드를 실행하면 다음과 같은 SQL이 실행된다.

```
INSERT INTO PRODUCT ...
INSERT INTO MEMBER ...
INSERT INTO MEMBER_PRODUCT ...
```

====== 탐색 ======

```
public void find() {

    Member member = em.find(Member.class, "member1");
    List<Product> products = member.getProducts(); //객체 그래프 탐색
    for (Product product : products) {
        System.out.println("product.name = " + product.getName());
    }
}
```

순서대로 저장한 후에 탐색해보면 저장해두었던 상품1이 조회된다.

`member.getProducts()`를 호출해서 상품 이름을 출력하면 다음 SQL이 실행된다.

```
SELECT * FROM MEMBER_PRODUCT MP
INNER JOIN PRODUCT P ON MP.PRODUCT_ID=P.PRODUCT_ID
WHERE MP.MEMBER_ID=?
```

실행된 SQL을 보면 연결 테이블인 `MEMBER_PRODUCT`와 상품 테이블을 조인해서 연관된 상품을 조회하는 것을 확인할 수 있다.

`@ManyToMany` 덕분에 복잡한 다대다 관계를 애플리케이션에서는 아주 단순하게 사용할 수 있다.

이제 이 관계를 양방향으로 만들어보자.

- 다대다 - 양방향

====== 역방향 추가 ======

```
@Entity
public class Product {

    @Id
    private String id;

    @ManyToMany(mappedBy = "products") //역방향 추가
    private List<Member> members;
    ...
}
```

다대다 매핑이므로 역방향도 `@ManyToMany`를 사용한다. 그리고 양쪽 중 원하는 곳에 `mappedBy`로 연관관계의 주인을 지정한다. (물론 `mappedBy`가 없는 곳이 연관관계의 주인이다.)

다대다의 양방향 연관관계는 다음처럼 설정하면 된다.

```
member.getProducts().add(product);
product.getMembers().add(member);
```

양방향 연관관계는 연관관계 편의 메서드를 추가해서 관리하는 것이 편리하다. 다음처럼 회원 엔티티에 연관관계 편의 메서드를 추가해보자.

```
public void addProduct(Product product) {
    ...
    products.add(product);
    product.getMembers().add(this);
}
```

연관관계 편의 메서드를 추가했으므로 다음처럼 간단히 양방향 연관관계를 설정하면 된다.

```
member.addProduct(product);
```

양방향 연관관계로 만들었으므로 `product.getMembers()`를 사용해서 역방향으로 객체그래프를 탐색할 수 있다.

====== 역방향 탐색 ======

```
public void findInverse() {

    Product product = em.find(Product.class, "productA");
    List<Member> members = product.getMembers();
    for (Member member : members) {
        System.out.println("member = " + member.getUsername());
    }
}
```

- 다대다 - 매핑의 한계와 극복, 연결 엔티티 사용

`@ManyToMany`를 사용하면 연결 테이블을 자동으로 처리해주므로 도메인 모델이 단순해지고 여러 가지로 편리하다. 하지만 이 매핑을 실무에서 사용하기에는 한계가 있다. 예를 들어 회원이 상품을 주문하면 연결 테이블에 단순히 주문한 회원 아이디와 상품 아이디만 담고 끝나지 않는다. 보통은 연결 테이블에 주문 수량 컬럼이나 주문한 날짜 같은 컬럼이 더 필요하다.

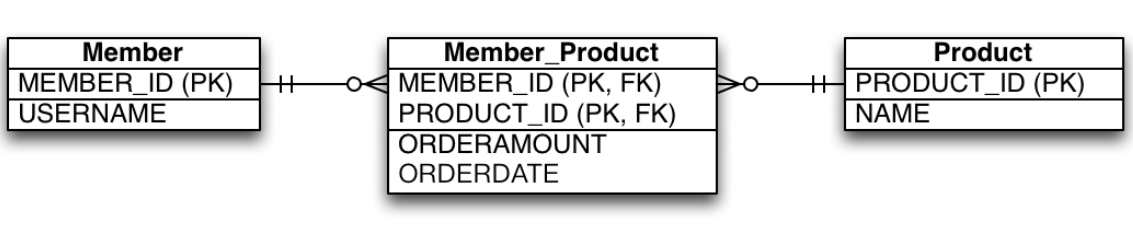

그림 3.18 I 테이블, 연결테이블에 필드 추가

연결 테이블에 주문 수량(`ORDERAMOUNT`)과 주문 날짜(`ORDERDATE`) 컬럼을 추가했다. 이렇게 컬럼을 추가하면 더는 `@ManyToMany`를 사용할 수 없다. 왜냐하면, 주문 엔티티나 상품 엔티티에는 추가한 컬럼들을 매핑할 수 없기 때문이다. 결국, 연결 테이블을 매핑하는 연결 엔티티를 만들고 이곳에 추가한 컬럼들을 매핑해야 한다. 그리고 엔티티 간의 관계도 테이블 관계처럼 다대다에서 일대다, 다대일 관계로 풀어야 한다. 여기서는 회원상품(`MemberProduct`) 엔티티를 추가했다.

그림 3.19 I 클래스, 다대다를 푸는 연결 엔티티

연결 테이블에 주문 수량(`ORDERAMOUNT`)과 주문 날짜(`ORDERDATE`) 컬럼을 추가했고 나머지 테이블은 기존과 같다. 매핑한 엔티티를 분석해보자.

====== 회원 코드 ======

```
@Entity
public class Member {

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

    //역방향
    @OneToMany(mappedBy = "member")
    private List<MemberProduct> memberProducts;

    ...
}
```

회원과 회원상품을 양방향 관계로 만들었다. 회원상품 엔티티 쪽이 외래 키를 가지고 있으므로 연관관계의 주인이다. 따라서 연관관계의 주인이 아닌 회원의 `Member.memberProducts` 에는 `mappedBy` 를 사용했다.

====== 상품 코드 ======

```
@Entity
public class Product {

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

    ...
}
```

상품 코드를 보면 상품 엔티티에서 회원상품 엔티티로 객체프래프 탐색 기능이 필요하지 않다고 판단해서 연관관계를 만들지 않았다.

다음으로 가장 중요한 회원상품 엔티티를 보자.

====== 회원상품 엔티티 ======

```
@Entity
@IdClass(MemberProductId.class) //**
public class MemberProduct {

    @Id
    @ManyToOne
    @JoinColumn(name = "MEMBER_ID")
    private Member member; //MemberProductId.member와 연결

    @Id
    @ManyToOne
    @JoinColumn(name = "PRODUCT_ID")
    private Product product; //MemberProductId.product와 연결

    private int orderAmount;

    ...
}
```

====== 회원상품 식별자 클래스 ======

```
public class MemberProductId implements Serializable {

    private String member; //MemberProduct.member와 연결
    private String product; //MemberProduct.product와 연결

    // hashCode and equals

    @Override
    public boolean equals(Object o) {...}

    @Override
    public int hashCode() {...}
}
```

회원상품(`MemberProduct`) 엔티티를 보면 기본 키를 매핑하는 `@Id` 와 외래 키를 매핑하는 `@JoinColumn` 을 동시에 사용해서 기본 키 + 외래 키를 한번에 매핑했다. 그리고 `@IdClass` 를 사용해서 복합 기본 키를 매핑했다.

복합 기본 키(간단히 복합 키라 하겠다.)

회원상품 엔티티는 기본 키가 `MEMBER_ID` 와 `PRODUCT_ID` 로 이루어진 복합 키다. JPA에서 복합 키를 사용하려면 별도의 식별자 클래스를 만들어야 한다. 그리고 엔티티에 `@IdClass` 를 사용해서 식별자 클래스를 지정하면 된다. 여기서는 `MemberProductId` 클래스를 복합 키용 식별자 클래스로 사용한다.

복합 키를 위한 식별자 클래스는 다음과 같은 특징이 있다.

- 복합 키는 별도의 식별자 클래스로 만들어야 한다.
- `Serializable` 을 구현해야 한다.
- `equals` 와 `hashCode` 메서드를 구현해야 한다.
- 기본 생성자가 있어야 한다.
- 식별자 클래스는 `public` 이어야 한다.
- `@IdClass` 를 사용하는 방법 외에 `@EmbeddedId` 를 사용하는 방법도 있다.

> **참고**: 자바 IDE에는 대부분 `equals` , `hashCode` 메서드를 자동으로 생성해주는 기능이 있다.

식별 관계

회원상품은 회원과 상품의 기본 키를 받아서 자신의 기본 키로 사용한다. 이렇게 부모 테이블의 기본 키를 받아서 자신의 기본 키 + 외래 키로 사용하는 것을 데이터베이스 용어로 식별 관계(Identifying Relationship)라 한다.

종합해보면 회원상품(`MemberProduct`)은 회원의 기본 키를 받아서 자신의 기본 키로 사용함과 동시에 회원과의 관계를 위한 외래 키로 사용한다. 그리고 상품의 기본 키도 받아서 자신의 기본 키로 사용함과 동시에 상품과의 관계를 위한 외래 키로 사용한다. 또한 `MemberProductId` 식별자 클래스로 두 기본 키를 묶어서 복합 기본 키로 사용한다.

> **참고:** 복합 키와 식별 관계 그리고 `@IdClass` , `@EmbeddedId` 대한 자세한 내용은 “8. 연관관계 매핑 심화2” 장에서 자세히 다룬다.

이렇게 구성한 관계를 어떻게 사용하는지 코드로 보자.

====== 저장하는 코드 ======

```
public void save() {

    //회원 저장
    Member member1 = new Member();
    member1.setId("member1");
    member1.setUsername("회원1");
    em.persist(member1);

    //상품 저장
    Product productA = new Product();
    productA.setId("productA");
    productA.setName("상품1");
    em.persist(productA);

    //회원상품 저장
    MemberProduct memberProduct = new MemberProduct();
    memberProduct.setMember(member1);    //주문 회원 - 연관관계 설정
    memberProduct.setProduct(productA); //주문 상품 - 연관관계 설정
    memberProduct.setOrderAmount(2);     //주문 수량

    em.persist(memberProduct);
}
```

회원상품 엔티티를 만들면서 연관된 회원 엔티티와 상품 엔티티를 설정했다. 회원상품 엔티티는 데이터베이스에 저장될 때 연관된 회원의 식별자와 상품의 식별자를 가져와서 자신의 기본 키 값으로 사용한다.

다음으로 조회하는 코드를 보자.

====== 조회 코드 ======

```
public void find() {

    //기본 키 값 생성
    MemberProductId memberProductId = new MemberProductId();
    memberProductId.setMember("member1");
    memberProductId.setProduct("productA");

    MemberProduct memberProduct = em.find(MemberProduct.class, memberProductId);

    Member member = memberProduct.getMember();
    Product product = memberProduct.getProduct();
```

```
        System.out.println("member = " + member.getUsername());
        System.out.println("product = " + product.getName());
        System.out.println("orderAmount = " + memberProduct.getOrderAmount());

    }
```

지금까지는 기본 키가 단순해서 기본 키를 위한 객체를 사용하는 일이 없었지만 복합 키가 되면 이야기가 달라진다. 복합 키는 항상 식별자 클래스를 만들어야 한다. `em.find()`를 보면 생성한 식별자 클래스로 엔티티를 조회한다.

복합 키는 복잡하다.

단순히 컬럼 하나만 기본 키로 사용하는 것과 비교해서 복합 키를 사용하면 ORM 매핑에서 처리할 일이 상당히 많아진다. 복합 키를 위한 식별자 클래스도 만들어야 하고 `@IdClass` 또는 `@EmbeddedId`도 사용해야 한다. 그리고 식별자 클래스에 `equals`, `hashCode`도 구현해야 한다.

다음으로 복합 키를 사용하지 않고 간단히 다대다 관계를 구성하는 방법을 알아보자.

- 다대다 - 새로운 기본 키 사용

추천하는 기본 키 생성 전략은 데이터베이스에서 자동으로 생성해주는 대리 키를 `Long` 값으로 사용하는 것이다. 이것의 장점은 간편하고 거의 영구히 쓸 수 있으며 비즈니스에 의존하지 않는다. 그리고 ORM 매핑시에 복합 키를 만들지 않아도 되므로 간단히 매핑을 완성할 수 있다.

이번에는 연결 테이블에 새로운 기본 키를 사용해보자. 이 정도 되면 회원상품(`MemberProduct`)보다는 주문(`Order`)이라는 이름이 더 어울릴 것이다. (`ORDER`는 일부 데이터베이스에 예약어로 잡혀있어서 대신에 `ORDERS`를 사용하기도 한다.)

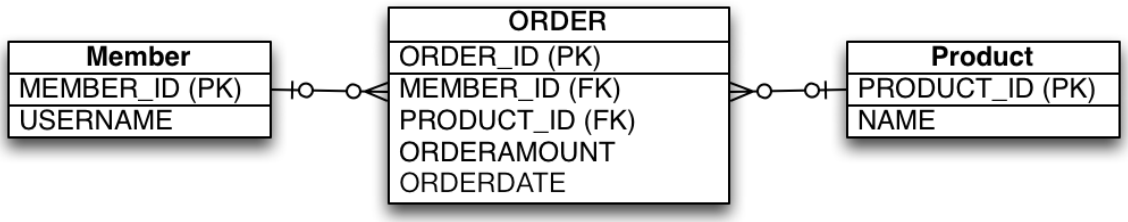

그림 - 테이블, N:M 다대다 새로운 기본 키

[그림 - 테이블, N:M 다대다 새로운 기본 키]를 보면 `ORDER_ID`라는 새로운 기본 키를 하나 만들고 `MEMBER_ID`, `PRODUCT_ID` 컬럼은 외래 키로만 사용한다.

주문 엔티티 코드를 보자.

====== 주문 ======

```
@Entity
public class Order {

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

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

    @ManyToOne
    @JoinColumn(name = "PRODUCT_ID")
    private Product product;

    private int orderAmount;
    ...
}
```

대리 키를 사용함으로써 이전에 보았던 식별 관계에 복합 키를 사용하는 것보다 매핑이 단순하고 이해하기 쉽다. 회원 엔티티와 상품 엔티티는 변경사항이 없다.

====== 회원 엔티티와 상품 엔티티 ======

```
@Entity
public class Member {

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

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

```
@Entity
public class Product {

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

```
    ...
}
```

이제 저장하고 조회하는 예제를 보자.

====== 저장 코드 ======

```
public void save() {

    //회원 저장
    Member member1 = new Member();
    member1.setId("member1");
    member1.setUsername("회원1");
    em.persist(member1);

    //상품 저장
    Product productA = new Product();
    productA.setId("productA");
    productA.setName("상품1");
    em.persist(productA);

    //주문 저장
    Order order = new Order();
    order.setMember(member1);   //주문 회원 - 연관관계 설정
    order.setProduct(productA); //주문 상품 - 연관관계 설정
    order.setOrderAmount(2);    //주문 수량
    em.persist(order);
}
```

====== 조회 코드 ======

```
public void find() {

    Long orderId = 1L;
    Order order = em.find(Order.class, orderId);

    Member member = order.getMember();
    Product product = order.getProduct();

    System.out.println("member = " + member.getUsername());
    System.out.println("product = " + product.getName());
    System.out.println("orderAmount = " + order.getOrderAmount());
}
```

식별자 클래스를 사용하지 않아서 코드가 한결 단순해졌다. 이처럼 새로운 기본 키를 사용해서 다대다 관계를 풀어내는 것도 좋은 방법이다.

정리

다대다 관계를 일대다 다대일 관계로 풀어내기 위해 연결 테이블을 만들 때 식별자를 어떻게 구성할지 선택해야 한다.

1. 식별 관계 : 받아온 식별자를 기본 키 + 외래 키로 사용한다.
2. 비식별 관계 : 받아온 식별자는 외래 키로만 사용하고 새로운 식별자를 추가한다.

데이터베이스 설계에서는 1번처럼 부모 테이블의 기본 키를 받아서 자식 테이블의 기본 키 + 외래 키로 사용하는 것을 식별 관계라 하고, 2번처럼 단순히 외래 키로만 사용하는 것을 비식별 관계라 한다. 객체 입장에서 보면 2번처럼 비식별 관계를 사용하는 것이 복합 키를 위한 식별자 클래스를 만들지 않아도 되므로 단순하고 편리하게 ORM 매핑을 할 수 있다.

이런 이유로 식별 관계보다는 비식별 관계를 추천한다. 자세한 내용은 8. 연관관계 매핑 - 심화2 장에서 알아보겠다.

[실전 예제] - 3. 다양한 연관관계 매핑하기

> **예제 코드 :** `ch06-model3`

다음 요구사항이 추가되었다.

- 상품을 주문할 때 배송 정보를 입력할 수 있다. 주문과 배송을 일대일 관계다.
- 상품을 카테고리로 구분할 수 있다.

배송 엔티티와 카테고리 엔티티를 추가했다.

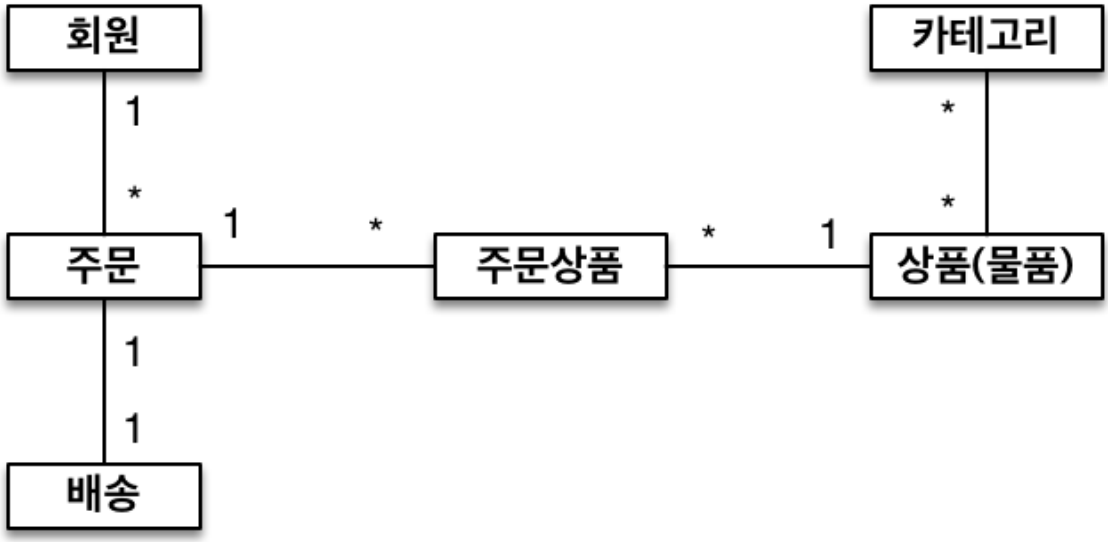

그림 - UML3

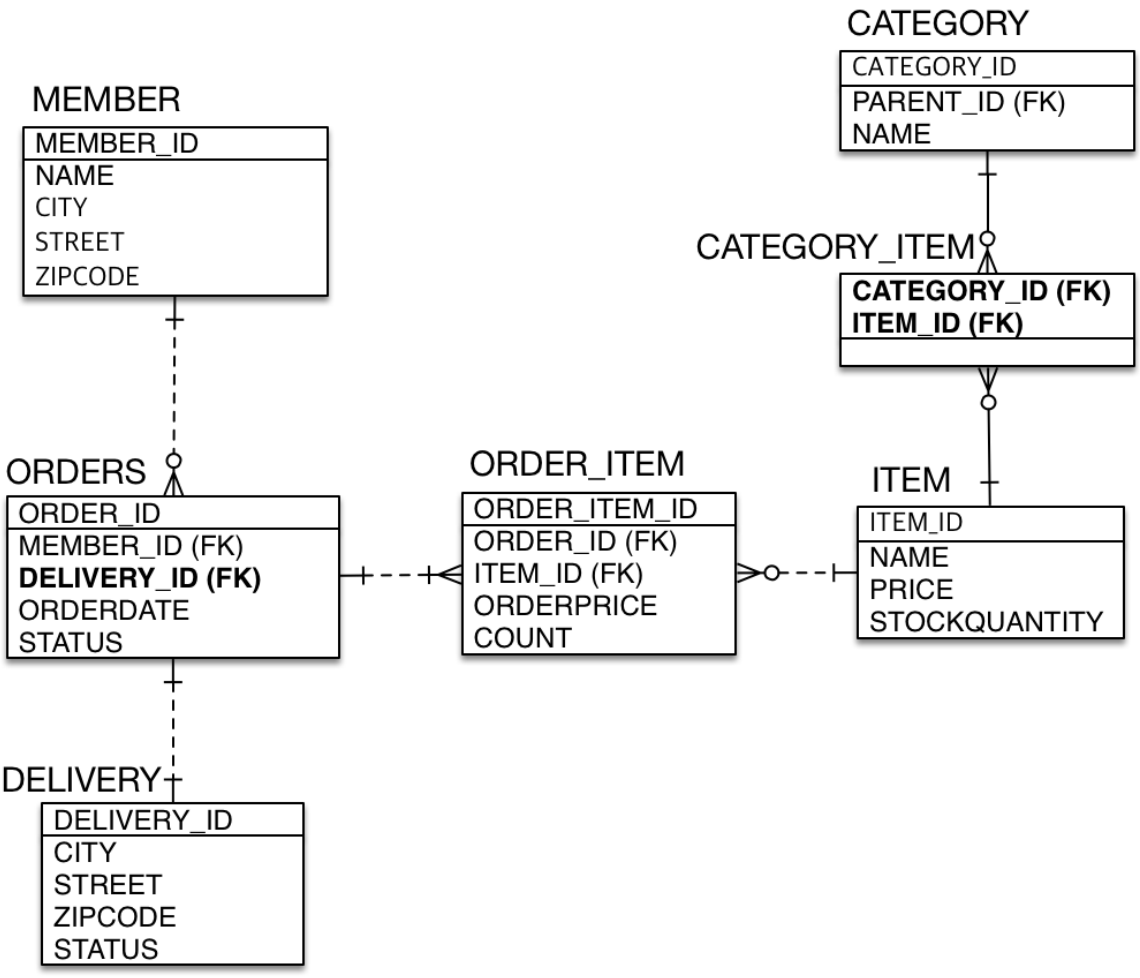

그림 - ERD3

주문과 배송: 주문(`ORDERS`)과 배송(`DELIVERY`)은 일대일 관계다. 객체 관계를 고려할 때 주문에서 배송으로 자주 접근할 예정이므로 외래 키를 주문 테이블에 두었다. 참고로 일대일 관계이므로 `ORDERS` 테이블에 있는 `DELIVERY_ID` 외래 키에는 유니크 제약조건을 주는 것이 좋다.

상품과 카테고리: 한 상품은 여러 카테고리(`CATEGORY`)에 속할 수 있고, 한 카테고리도 여러 상품을 가질 수 있으므로 둘은 다대다 관계다. 테이블로 이런 다대다 관계를 표현하기는 어려우므로 `CATEGORY_ITEM` 연결 테이블을 추가해서 다대다 관계를 일대다, 다대일 관계로 풀어냈다.

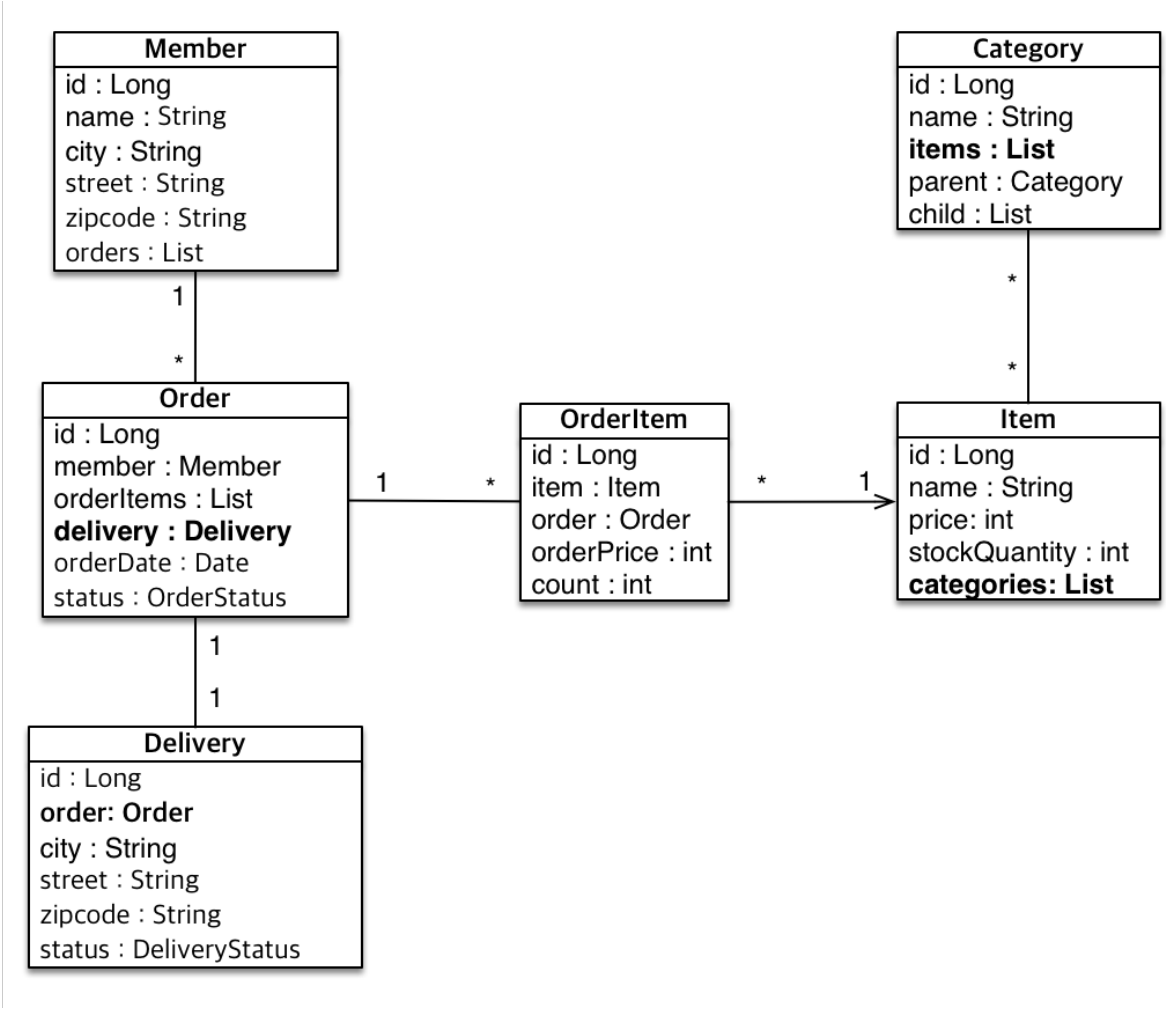

그림 - UML 상세3

일대일 매핑하기

주문과 배송의 관계를 보자.

=== 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>();

    @OneToOne                                   //**
    @JoinColumn(name = "DELIVERY_ID")           //**
    private Delivery delivery;  //배송정보        //**

    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);
    }

    public void setDelivery(Delivery delivery) { //**
        this.delivery = delivery;                //**
        delivery.setOrder(this);                 //**
    }                                            //**

    //Getter, Setter
```

```
    ...
}
```

=== 배송(`Delivery`) ===

```
package jpabook.model.entity;

import javax.persistence.*;

@Entity
public class Delivery {

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

    @OneToOne(mappedBy = "delivery")
    private Order order;

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

    @Enumerated(EnumType.STRING)
    private DeliveryStatus status;

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

=== 배송상태(`DeliveryStatus`) ===

```
package jpabook.model.entity;

public enum DeliveryStatus {
    READY, //준비
    COMP   //배송
}
```

`Order` 와 `Delivery` 는 일대일 관계고 그 반대도 일대일 관계다. 여기서는 `Order` 가 매핑된 `ORDERS` 를 주 테이블로 보고 주 테이블에 외래 키를 두었다. 따라서 외래 키가 있는 `Order.delivery` 가 연관관계의 주인이다. 주인이 아닌 `Delivery.order` 필드에는 `mappedBy` 속성을 사용해서 주인이 아님을 표시했다.

다대다 매핑하기

카테고리와 상품의 관계를 보자.

=== 카테고리(`Category`) ===

```
package jpabook.model.entity;

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

@Entity
public class Category {

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

    private String name;

    @ManyToMany
    @JoinTable(name = "CATEGORY_ITEM",
            joinColumns = @JoinColumn(name = "CATEGORY_ID"),
            inverseJoinColumns = @JoinColumn(name = "ITEM_ID"))
    private List<Item> items = new ArrayList<Item>();

    //카테고리의 계층 구조를 위한 필드들
    @ManyToOne
    @JoinColumn(name = "PARENT_ID")
    private Category parent;

    @OneToMany(mappedBy = "parent")
    private List<Category> child = new ArrayList<Category>();

    //==연관관계 메서드==//
    public void addChildCategory(Category child) {
        this.child.add(child);
        child.setParent(this);
    }

    public void addItem(Item item) {
        items.add(item);
    }

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

=== 상품(`Item`) ===

```
package jpabook.model.entity;

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

@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

    @ManyToMany(mappedBy = "items")                                  //**
    private List<Category> categories = new ArrayList<Category>(); //**

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

`Category` 와 `Item` 은 다대다 관계고 그 반대도 다대다 관계다. `Category.items` 필드를 보면 `@ManyToMany` 와 `@JoinTable` 을 사용해서 `CATEGORY_ITEM` 연결 테이블을 바로 매핑했다. 그리고 여기서는 `Category` 를 연관관계의 주인으로 정했다. 따라서 주인이 아닌 `Item.categories` 필드에는 `mappedBy` 속성을 사용해서 주인이 아님을 표시했다.

다대다 관계는 연결 테이블을 JPA가 알아서 처리해주므로 편리하지만, 연결 테이블에 필드가 추가되면 더는 사용할 수 없으므로 실무에서 활용하기에는 무리가 있다. 따라서 `CategoryItem` 이라는 연결 엔티티를 만들어서 일대다, 다대일 관계로 매핑하는 것을 권장한다.