6. 논리적 모델링2 - 참여도와 일대다 관계

#0.강의/2.데이터베이스로드맵/3.설계1

- /논리적 모델링 - 관계
- /참여도
- /일대다(1:N) 다대일(N:1) 관계 - 외래 키 위치1
- /일대다(1:N) 다대일(N:1) 관계 - 외래 키 위치2
- /일대다(1:N) 다대일(N:1) 관계 - 조인과 뻥튀기
- /정리

논리적 모델링 - 관계

우리는 지난 시간에 데이터의 유일성을 보장하는 다양한 '키(Key)'에 대해 학습했다. 이제 그 키들을 사용해서, 테이블이라는 섬들을 연결하는 '관계'라는 다리를 놓을 차례다. **개념적 모델링에서 그렸던 추상적인 관계선이 논리적 모델링에서는 외래 키(Foreign Key)라는 구체적인 형태로 구현**된다.

이 관계를 어떻게 설정하느냐에 따라 데이터 모델의 유연성과 확장성이 결정된다. 이번 시간에는 다양한 관계의 종류를 깊이 있게 이해하고, 특히 현대적인 설계에서 왜 특정 방식의 관계 설정을 선호하는지, 그 실무적인 이유를 자세히 알아보자.

관계형 데이터베이스와 관계의 방향

관계형 데이터베이스의 관계를 처음 배울 때 가장 흔하게 하는 오해 중 하나는 관계에 '방향'이 있다고 생각하는 것이다. 특히 개발자의 경우 객체 지향 프로그래밍(OOP)의 참조 개념에 익숙하다 보니, 한쪽에서 다른 쪽으로만 접근할 수 있는 단방향 관계라고 생각하기 쉽다.

결론부터 말하자면, **관계형 데이터베이스의 관계(Relationship)에는 방향이 없다.** 외래 키는 두 테이블을 연결하는 제약조건일 뿐, 데이터 조회 방향을 제한하지 않는다.

이번에는 `team` 과 `member` 의 관계를 예시로 직접 데이터를 만들고 조회하며 이 개념을 확실히 이해해 보자.

예제 테이블 및 데이터 준비

먼저 예제로 사용할 `team` 과 `member` 테이블을 생성하고, 샘플 데이터를 추가한다. 하나의 팀(`team`)은 여러 회원(`member`)을 가질 수 있는 1:N 관계로 모델링한다. `member` 테이블의 `team_id` 가 `team` 테이블의 `team_id` 를

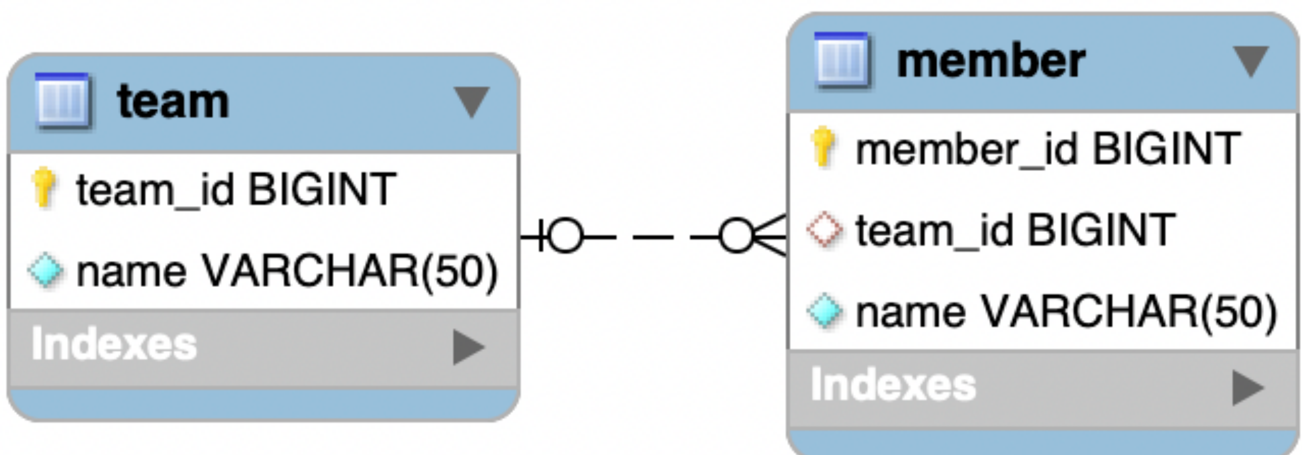
참조하는 외래 키가 된다.

```
DROP TABLE IF EXISTS member_detail; -- 다른 예제에서 발생
DROP TABLE IF EXISTS member;
DROP TABLE IF EXISTS team;

-- 팀 테이블 생성
CREATE TABLE team (
    team_id BIGINT NOT NULL AUTO_INCREMENT,
    name VARCHAR(50) NOT NULL,
    PRIMARY KEY (team_id)
);

-- 회원 테이블 생성
CREATE TABLE member (
    member_id BIGINT NOT NULL AUTO_INCREMENT,
    team_id BIGINT NULL, -- 회원은 팀에 소속되지 않을 수도 있다 (NULL 허용)
    name VARCHAR(50) NOT NULL,
    PRIMARY KEY (member_id),
    CONSTRAINT fk_member_team FOREIGN KEY (team_id)
        REFERENCES team (team_id)
);

-- 샘플 데이터 추가
INSERT INTO team(name) VALUES ('개발팀'), ('기획팀');
INSERT INTO member(name, team_id) VALUES ('션', 1), ('잡스', 1), ('네이트', 2);
```



객체 지향 프로그래밍의 방향성 있는 관계

이 개념을 명확히 이해하기 위해, 방향이 있는 관계를 가진 객체 지향 프로그래밍의 예시를 먼저 살펴보자. `Team`과 `Member` 객체를 만든다고 가정해 보자.

```
class Team {
    Long teamId;
    String name;
    List<Member> members = new ArrayList<>(); // 팀에 소속된 회원 목록 참조
}

class Member {
    Long memberId;
    String name;
    // Member 객체는 어느 팀에 속했는지 모른다. Team 참조가 없다.
}
```

Team → Member 접근O

이 설계에서 `Team` 객체는 `members` 리스트를 통해 소속된 `Member` 객체들을 알고 있다. 따라서 `team.members` 처럼 팀에서 회원으로 접근하는 것은 매우 쉽다.

Member → Team 접근X

하지만 반대로, 특정 `Member` 객체만 보면 이 회원이 어떤 `Team`에 속했는지 역으로 찾아가는 것은 불가능하다. `Member` 객체는 `Team`에 대한 참조를 가지고 있지 않기 때문이다.

이것이 바로 **방향이 있는 단방향 관계**다.

만약 **Member → Team**의 접근이 필요하다면 다음과 같이 `Member -> Team`으로 접근할 수 있는 코드를 추가해야 한다.

```
class Member {
    Long memberId;
    String name;
    Team team; // member -> team 접근 추가
}
```

이렇게 서로 양쪽에서 참조할 수 있으면 **양방향 관계**가 된다.

데이터베이스의 양방향 관계

반면, 관계형 데이터베이스는 다르다. `member` 테이블의 `team_id` 외래 키가 두 테이블을 연결하는 다리 역할을 하기 때문에, `JOIN`을 통해 어느 쪽에서든 자유롭게 양방향으로 데이터를 탐색할 수 있다.

1. 회원(Member) 관점에서 팀(Team) 조회하기

먼저 외래 키가 있는 `member`를 중심으로 조회해보자.

'회원 잡스가 소속된 팀의 이름은 무엇인가?' 라는 요구사항을 처리해보자. `member` 테이블에서 시작한다.

```
SELECT
    m.name AS member_name,
    t.name AS team_name
FROM member m
JOIN team t ON m.team_id = t.team_id
WHERE m.name = '잡스';
```

[실행 결과]

member_name	team_name
잡스	개발팀

이 관계는 FK ➜ PK로 조인한다.

2. 팀(Team) 관점에서 회원(Member) 조회하기

이번에는 외래 키가 없는 `team`을 중심으로 조회해보자.

'`개발팀`에 소속된 모든 회원의 이름을 조회하라'는 요구사항은 `team` 테이블에서 시작하여 `member` 테이블을 `JOIN` 하면 된다.

```
SELECT
    t.name AS team_name,
    m.name AS member_name
FROM team t
JOIN member m ON t.team_id = m.team_id
WHERE t.name = '개발팀';
```

[실행 결과]

team_name	member_name
개발팀	션
개발팀	잡스

이 관계는 PK ➔ FK로 조인한다.

어떻게 FK 하나로 양방향 조회가 가능할까?
조인은 FK ➔ PK도 가능하지만, 그 반대인 PK ➔ FK도 가능하다.
외래 키가 있는 곳에서는 조인을 사용해서 FK를 통해서 PK를 찾으면 되고, 그 반대편에서는 PK를 통해서 FK를 찾으면 된다.

이처럼 관계형 데이터베이스의 관계는 양방향 도로와 같다. 객체 지향처럼 한쪽에서만 접근할 수 있는 일방통행 길이 아니다.

결과적으로 두 테이블이 관계를 맺으려면 FK를 둘 중 한 곳에 두면 된다. 그럼 조인을 통해서 두 테이블을 서로 참조할 수 있다.
그럼 FK를 둘 중 어디에 두면 될까? FK를 어디에 두는가에 따라 일대다와 같은 카디널리티를 포함한 수 많은 설계 정책에 영향을 준다. FK의 위치는 설계에서 너무 중요하기 때문에 매우 자세히 다루겠다.

관계의 2대 핵심 요소: 카디널리티와 참여도

논리적 모델링에서의 관계 역시 개념적 모델링에서와 마찬가지로 **카디널리티(Cardinality)**와 **참여도(Optionality)**라는 두 가지 핵심 속성을 가진다. 이 두 가지를 어떻게 조합하느냐에 따라 관계의 성격이 결정된다.

- **카디널리티**: 한 테이블의 행이 다른 테이블의 행과 몇 개나 연결될 수 있는지를 나타낸다 (1:1, 1:N, N:1, M:N).
- **참여도**: 한 테이블의 행이 관계를 맺고 있는 다른 테이블에 반드시 대응되는 행을 가져야 하는지(필수 참여), 아니면 갖지 않아도 되는지(선택 참여)를 나타낸다.

앞서 개념적 모델링에서 이미 학습한 개념들을 이제는 실제 관계형 데이터베이스에서 어떻게 사용하는지 논리적 모델링 관점에서 살펴보자.
먼저 간단한 참여도부터 시작해보자.

참여도

관계의 핵심 요소로는 카디널리티와 함께 **참여도(Optionality)**가 있다. 참여도는 한 엔티티가 관계에 **필수적으로 참여해야 하는지(Mandatory)**, 아니면 **선택적으로 참여할 수 있는지(Optional)**를 나타낸다.

- **필수적 관계(Mandatory Relationship)**: 관계를 맺는 상대방 엔티티에 반드시 대응되는 행이 있어야 한다. (예: 모든 회원은 반드시 팀에 소속되어야 한다.)
- **선택적 관계(Optional Relationship)**: 상대방 엔티티에 대응되는 행이 없어도 된다. (예: 회원은 팀에 소속될 수도 있고, 아닐 수도 있다.)

논리적 모델링에서 이 참여도는 외래 키(FK) 컬럼의 `NULL` **허용 여부(**`NULL` **또는** `NOT NULL`**)**로 구현된다. 하지만 이 제약조건만으로 모든 비즈니스 규칙을 강제할 수는 없다. 지금부터 `회원` 과 `팀` 예제를 통해 참여도를 데이터베이스로 구현하는 방법과 그 한계를 명확히 알아보자.

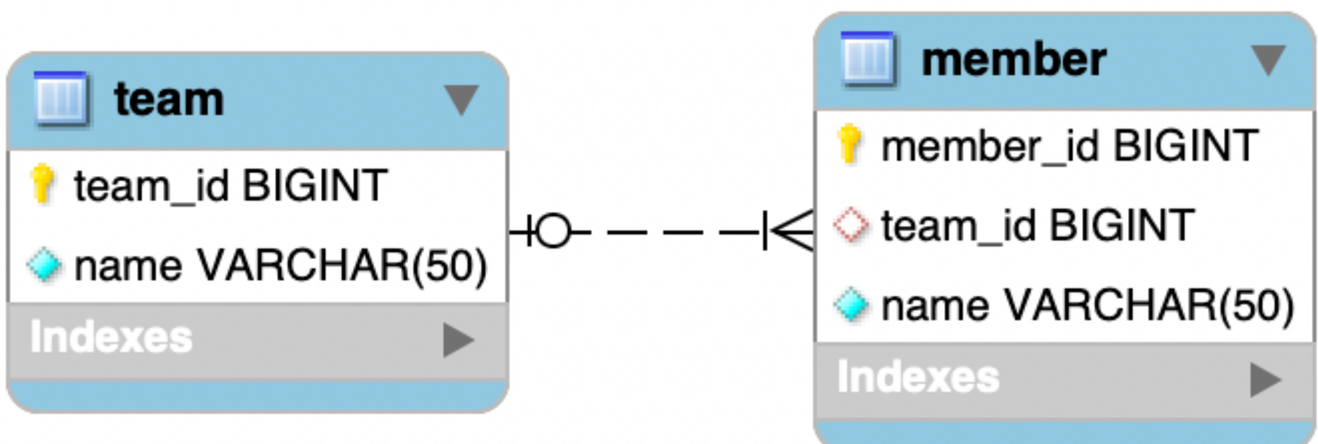
외래 키와 참여도 구현

팀(`team`)과 회원(`member`)의 예를 보자.

외래 키가 있는 '다(N)' 쪽, 즉 `member` 테이블에서 참여도를 구현하는 것은 매우 직관적이다.

1. 회원은 팀에 소속될 수 있다 (선택적 참여)

- **비즈니스 규칙**
  - 회원은 하나의 팀에 소속될 수도 있고, 아직 소속되지 않은 상태일 수도 있다. → **여기에 집중하자.**
  - 팀은 여러 회원을 가질 수 있다. 팀은 회원을 필수로 가져야 한다.
- **구현**: `member` 테이블의 `team_id` 외래 키 컬럼을 `NULL` **허용**으로 설정한다.



- `team_id` FK 빨간색 마름모가 비어있는 것을 확인할 수 있다. NULL 허용이라는 뜻이다.
- `member -> team` 이 `0` 로 선택적 관계인 것을 확인할 수 있다.

> 까마귀발 표기법 - Crow's Foot
> - 까마귀발 표기법을 읽을 때 가장 중요한 원칙은 **"한 엔티티 쪽 끝에 있는 기호는 반대편 엔티티에 대한 규칙을 설명한다"**는 것이다.
> - 관계선 끝의 기호는 반대편 엔티티의 최소, 최대 참여 수(필수/선택, 1/여러)를 뜻한다.

```sql
DROP TABLE IF EXISTS member;
DROP TABLE IF EXISTS team;

CREATE TABLE team (
    team_id     BIGINT       NOT NULL AUTO_INCREMENT,
    name        VARCHAR(50)  NOT NULL,
    PRIMARY KEY (team_id)
);

CREATE TABLE member (
    member_id    BIGINT       NOT NULL AUTO_INCREMENT,
    team_id      BIGINT       NULL, -- NULL 허용으로 선택적 참여 구현
    name         VARCHAR(50)  NOT NULL,
    PRIMARY KEY (member_id),
    CONSTRAINT fk_member_team FOREIGN KEY (team_id)
        REFERENCES team (team_id)
);
```

이렇게 설계하면 팀이 정해지지 않은 신입 회원을 `team_id` 없이도 `member` 테이블에 추가할 수 있다.

```sql
INSERT INTO member(name, team_id) VALUES ('이기획', NULL);
```

이 쿼리는 `team_id` 가 `NULL` 이므로 성공적으로 실행된다.

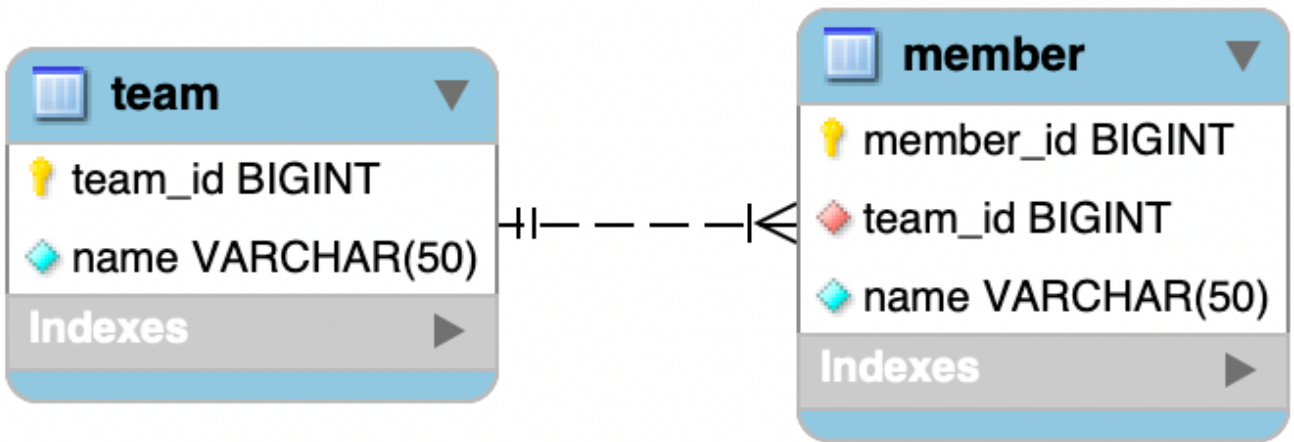
```sql
SELECT * FROM member;
```

[실행 결과]

member_id	team_id	name
1	NULL	이기획

2. 회원은 반드시 팀에 소속되어야 한다 (필수적 참여)

- **비즈니스 규칙**: 모든 회원은 가입과 동시에 반드시 특정 팀에 소속되어야 한다.
- **구현**: `member` 테이블의 `team_id` 외래 키 컬럼을 `NOT NULL`로 설정한다.



- `team_id` FK 빨간색 마름모가 꽉 차있는 것을 확인할 수 있다. `NOT NULL`이라는 뜻이다.
- `member -> team`이 필수적 참여이므로 `0` ➜ `|`로 변한 것을 확인할 수 있다.

```
DROP TABLE IF EXISTS member;

CREATE TABLE member (
    member_id   BIGINT      NOT NULL AUTO_INCREMENT,
    team_id     BIGINT      NOT NULL, -- NOT NULL로 필수적 참여 구현
    name        VARCHAR(50) NOT NULL,
    PRIMARY KEY (member_id),
    CONSTRAINT fk_member_team FOREIGN KEY (team_id)
        REFERENCES team (team_id)
);
```

이제 팀 정보 없이 회원을 추가하려고 하면 데이터베이스가 오류를 발생시켜 참조 무결성을 지켜준다.

```
INSERT INTO member(name, team_id) VALUES ('이기획', NULL);
```

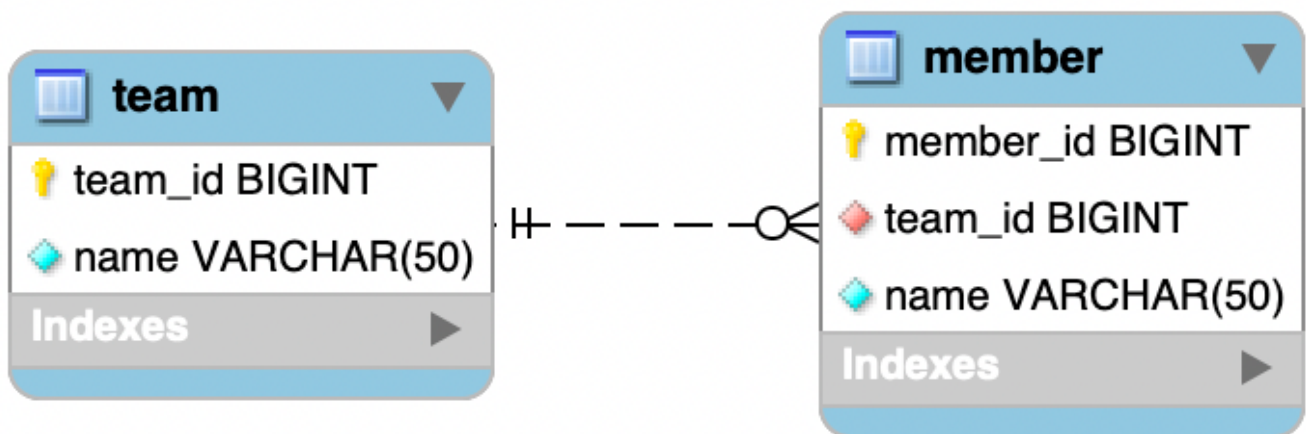
[실행 결과]

```
Error Code: 1048. Column 'team_id' cannot be null
```

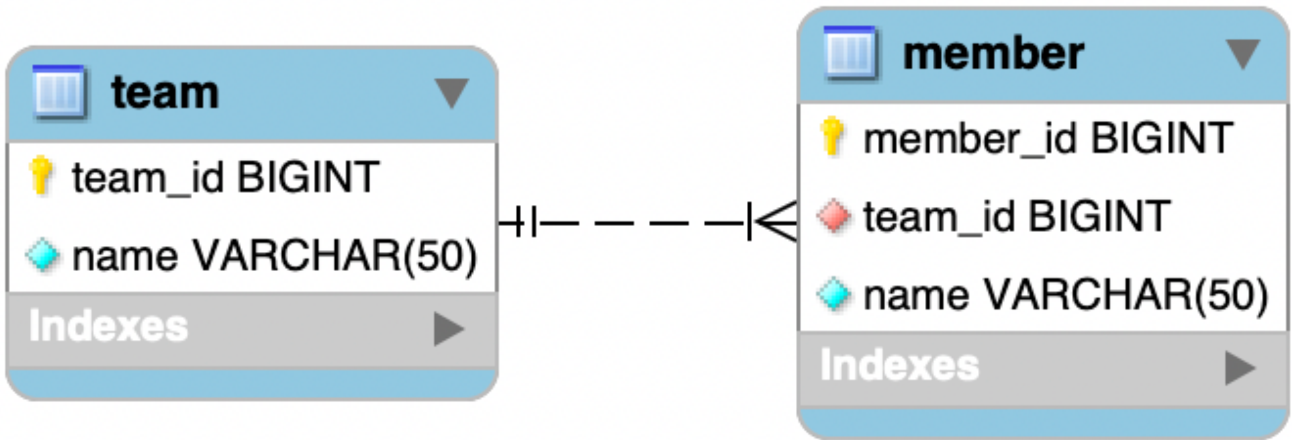
이처럼 외래 키가 있는 쪽의 참여도는 `NULL` 제약조건으로 간단하고 강력하게 제어할 수 있다.

관계 반대편의 참여도: 데이터베이스 제약의 한계

그렇다면 관계의 반대편, 즉 '일(1)' 쪽인 `team` 테이블 입장에서의 참여도는 어떨까?



- **선택적 참여**: 팀은 회원이 한 명도 없을 수 있다. (예: 신설팀)
- `team -> member` 이 선택적 참여이므로 `0`



- **필수적 참여**: 팀은 생성될 때 반드시 최소 한 명 이상의 회원을 가져야 한다.
- `team -> member` 가 필수적 참여이므로 `0` ➜ `|` 로 변한 것을 확인할 수 있다.

선택적 참여는 데이터베이스에서 자연스럽게 구현된다. 그냥 `team` 테이블에 새로운 팀을 추가하기만 하면 된다. `member` 테이블에 해당 팀을 참조하는 데이터가 있든 없든 상관없다. 이 관계에서 `team` 테이블에만 데이터를 넣는다고 생각해보자. `member` 없이 `team` 에만 데이터를 넣어도 아무런 문제가 없다.

하지만 **필수적 참여**, 즉 '팀은 최소 한 명 이상의 회원을 가져야 한다'는 규칙은 어떻게 강제할 수 있을까? 결론부터 말하면, **외래 키와 같은 기본 제약조건만으로는 이 규칙을 강제할 수 없다.**

왜 데이터베이스 제약으로 강제할 수 없을까?

이유는 아주 근본적인 곳에 있다. 바로 **제약을 걸 대상, 즉 외래 키 컬럼이 `team` 테이블에 존재하지 않기 때문**이다.

생각해보자. `member` 테이블에서는 `team_id` 라는 명확한 외래 키 컬럼이 있었다. 우리는 이 컬럼에 `NOT NULL` 제약조건을 걸어서 '팀 없는 회원'의 생성을 막을 수 있었다. 즉, **규칙을 강제할 수단(컬럼)이 명확히 있었다.**

하지만 `team` 테이블에는 회원을 가리키는 어떠한 외래 키도 존재하지 않는다. 따라서 `NOT NULL` 과 같은 제약조건을 걸 대상 자체가 없는 것이다. 데이터베이스 입장에서는 규칙을 강제할 수단이 구조적으로 없는 셈이다.

앞에서 그린 ERD를 비교해보자.

- `member -> team` 의 관계는 `member` 에 있는 `team_id` FK에 `NULL` , `NOT NULL` 제약조건을 사용해서 필수적 선택적 관계를 변경했다. `team_id` FK 마름모가 비어있고, 꽉차있는 모습으로 변했다.
- 반면에 `team -> member` 의 관계는 `team` 에 FK가 없다. 따라서 `team -> member` 의 관계를 제약할 수 있는 방법이 없다.

결국 관계형 데이터베이스의 제약조건 만으로는 외래 키가 없는 `team -> member` 와 같은 관계는 필수적 관계로 만들 수 없다.

이러한 종류의 비즈니스 규칙은 보통 **애플리케이션 계층(Application Layer)에서 로직으로 해결**한다.

예를 들면 애플리케이션 계층에서 다음과 같은 쿼리를 하나의 트랜잭션으로 수행하는 것이다.

```
-- 팀 생성과 첫 멤버 등록을 하나의 트랜잭션으로
START TRANSACTION;

INSERT INTO team(name) VALUES ('플랫폼팀');
SET @team_id := LAST_INSERT_ID();

INSERT INTO member(name, team_id) VALUES ('션', @team_id);

COMMIT;
```

> ✨ **데이터베이스 고급 기능**
> 데이터베이스에 따라 트리거(Trigger)나 같은 기능을 사용해 이런 복잡한 규칙을 구현할 수도 있다. 하지만 이는 데이터베이스에 비즈니스 로직이 깊게 관여하게 만들어 애플리케이션과의 의존성을 높이고 관리를 어렵게 할 수 있으므로, 신중하게 사용해야 한다. 대부분의 현대적인 개발 방식에서는 애플리케이션에서 트랜잭션으로 처리하는 것을 더 선호한다.

ERD와 비즈니스 규칙의 중요성

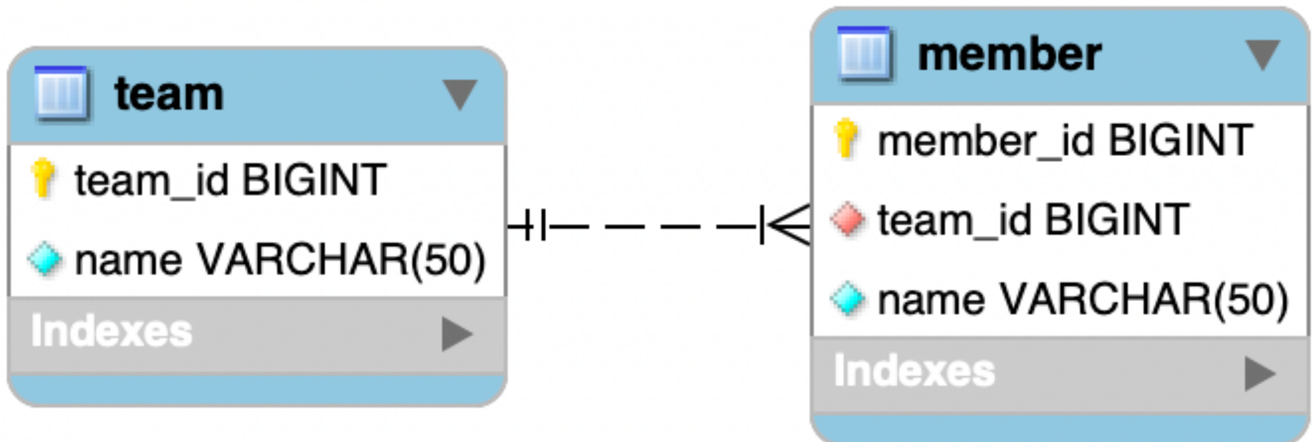
여기서 우리가 반드시 기억해야 할 중요한 점이 있다. 데이터베이스 제약조건으로 구현하기 어렵다고 해서, ERD(개체-관계 다이어그램)에 원래의 비즈니스 규칙을 표현하지 않으면 안 된다는 것이다.

- **ERD**: 데이터베이스의 물리적 제약을 넘어, **원래의 비즈니스 규칙을 그대로 표현**해야 한다. 만약 '팀은 반드시 한 명 이상의 회원을 가져야 한다'는 규칙이 있다면, ERD에는 그 필수 참여 관계가 명확하게 그려져야 한다.
- **데이터베이스 DDL**: ERD에 표현된 규칙 중, 데이터베이스가 제약조건으로 강제할 수 있는 부분까지만 구현한다.
- **애플리케이션 코드**: DDL로 구현하지 못한 나머지 비즈니스 규칙을 책임지고 구현하여 데이터의 무결성을 완성한다.

예를 들어, 다음과 같은 비즈니스 규칙이 있다고 가정해보자.

1. 모든 회원은 반드시 하나의 팀에 소속되어야 한다. (**필수 참여**)
2. 팀은 생성될 때 반드시 최소 한 명 이상의 회원을 가져야 한다. (**필수 참여**)

ERD 표현 (Crow's Foot Notation 기준)



- `team` 쪽의 `||`: `member`는 `team`에 반드시 참여해야 한다. (필수 참여).
- `member` 쪽의 `|<`: `team`은 `member`를 반드시 가져야 한다. (필수 참여).

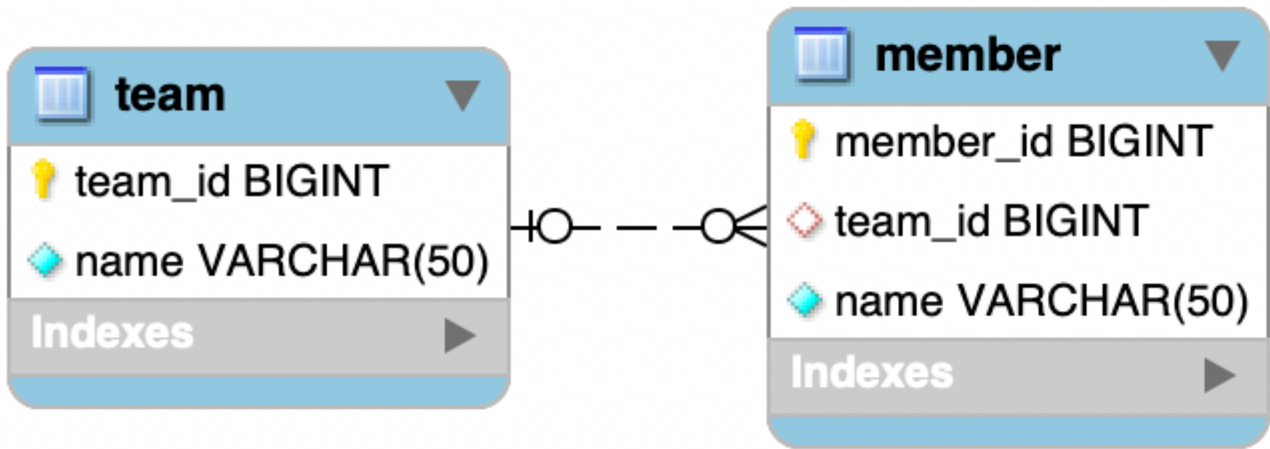
DDL 구현

- `member.team_id`는 `NOT NULL`로 설정한다. (규칙 1 구현)
- `team` 테이블에는 FK가 없으므로 제약조건으로 강제할 수 없다. 따라서 애플리케이션에서 구현해야 한다. (규칙 2 구현)

결론적으로, 데이터베이스 설계는 단순히 테이블과 컬럼을 만드는 작업이 아니다. 비즈니스 규칙을 명확히 이해하고, 그 규칙을 ERD에 올바르게 표현한 뒤, 데이터베이스 제약조건과 애플리케이션 코드가 각자의 역할을 분담하여 전체 데이터의 무결성을 지켜나가는 과정이다.

일대다(1:N) 다대일(N:1) 관계 - 외래 키 위치1

일대다(1:N) 다대일(N:1) 관계는 가장 흔하게 볼 수 있는 관계다. 실무에서 만나는 대부분의 관계는 일대다 또는 다대일 관계라고 봐도 무방하다. 우리 회사의 `팀(Team)`과 `회원(Member)`을 예로 들어보자.



- 하나의 팀에는 여러 명의 회원이 소속될 수 있다.
- 한 명의 회원은 반드시 하나의 팀에만 소속된다.
- 여기서는 둘 다 선택적 관계를 사용하겠다.

이 관계를 어느 테이블 입장에서 바라보느냐에 따라 관계의 카디널리티가 달라진다.

- `팀` 입장에서 `회원` 을 바라보면, 하나의 팀이 여러 회원을 가지므로 **일대다(1:N)** 관계다.
- `회원` 입장에서 `팀` 을 바라보면, 여러 회원이 하나의 팀에 속하므로 **다대일(N:1)** 관계다.

결국 같은 관계를 어느 쪽에서 바라보느냐의 차이일 뿐이다.
논리적 모델링에서 중요한 점은 이 관계를 데이터베이스 테이블로 구현하는 방법이다.

외래 키(Foreign Key)는 어디에 위치해야 할까?

관계형 데이터베이스는 외래 키를 사용해서 관계를 표현한다.
그렇다면 일대다 또는 그 반대인 다대일 관계에서 외래 키는 어디에 위치해야 할까?
결론부터 말하면, **외래 키는 항상 '다(N)' 쪽에 존재해야 한다.** 즉, `회원(N)` 테이블이 `팀(1)` 테이블의 기본 키를 외래 키로 가져야 한다.

만약 일(1) 쪽인 팀(`team`) 테이블에 외래 키를 둔다고 상상해보자.
'개발팀'에 '김개발'과 '박개발'이 소속되어 있다고 가정하고, `팀` 테이블에 외래 키를 두는 잘못된 설계를 살펴보자.

'일(1)' 쪽에 외래 키를 두면 발생하는 문제1 - 여러 행에 나누어 보관

하나의 팀은 하나의 행에 저장해야 한다. 따라서 다음과 같이 여러 행을 저장하는 것은 말이 안 된다.

```
DROP TABLE IF EXISTS team_bad1;
```

```sql
CREATE TABLE team_bad1 (
    team_id BIGINT NOT NULL,
    name VARCHAR(50) NOT NULL,
    member_id BIGINT NULL,
    PRIMARY KEY (team_id)
);

-- 데이터 삽입
INSERT INTO team_bad1(team_id, name, member_id) VALUES (1, '개발팀', 1); -- 김개발
(ID:1)

-- 오류 발생
INSERT INTO team_bad1(team_id, name, member_id) VALUES (1, '개발팀', 2); -- 박개발
(ID:2)
```

[기대하는 결과]

team_id	name	member_id
1	개발팀	1(김개발)
1(???)	개발팀	2(박개발)

[실행 결과]

```
Error Code: 1062. Duplicate entry '1' for key 'team_bad1.PRIMARY'
```

- 팀 테이블에 같은 팀을 여러 행 저장해서는 안된다.
- 두 번째 회원을 저장하는 순간 오류가 발생한다.

'일(1)' 쪽에 외래 키를 두면 발생하는 문제2 - 하나의 컬럼에 보관

그렇다면 하나의 컬럼 안에 여러 `member_id` 를 보관하면 어떨까?

예를 들어 `member_ids` 라는 컬럼 하나에 '1,2'와 같은 문자열을 넣는 방식은 어떨까?

```sql
DROP TABLE IF EXISTS team_bad2;
CREATE TABLE team_bad2 (
    team_id BIGINT NOT NULL,
    name VARCHAR(50) NOT NULL,
```

```
    member_ids VARCHAR(255) NULL, -- 쉼표로 구분된 회원 ID 목록
    PRIMARY KEY (team_id)
);

-- 데이터 삽입
INSERT INTO team_bad2(team_id, name, member_ids) VALUES (1, '개발팀', '1,2');
```

[저장된 결과]

team_id	name	member_ids
1	개발팀	1, 2

이 방식은 관계형 데이터베이스의 가장 기본 원칙을 위배하는, 매우 심각한 문제를 가진 설계다.

1. **제1정규형(1NF) 위반**: 관계형 데이터베이스의 컬럼은 **원자성(Atomicity)**을 가져야 한다. 즉, 컬럼 하나에는 단 하나의 값만 저장되어야 한다는 원칙이다. '1, 2'처럼 여러 값을 하나의 문자열로 욱여넣는 것은 이 원칙을 정면으로 위배한다. 정규형은 뒤에서 설명한다.

2. **데이터 검색의 어려움**: `'박개발(ID:2)' 회원이 소속된 팀을 찾아라` 라는 간단한 요구사항을 처리하기가 매우 까다롭다.

   ```
   SELECT name FROM team_bad2 WHERE member_ids LIKE '%2%';
   ```

   이런 쿼리는 인덱스를 전혀 활용할 수 없어 테이블의 모든 데이터를 하나씩 다 확인하는 '풀 테이블 스캔(Full Table Scan)' 방식으로 동작한다. 데이터가 많아질수록 성능은 끔찍하게 저하된다.

3. **데이터 수정의 복잡성**: '개발팀'에 '최개발(ID:3)'을 추가하려면 어떻게 해야 할까? 애플리케이션에서 `'1,2'` 를 읽어와서 문자열을 파싱하고, `,3` 을 붙여서 `'1,2,3'` 으로 다시 업데이트해야 한다. 반대로 회원을 팀에서 제외하는 로직은 더욱 복잡하다. 이런 로직은 오류가 발생하기 매우 쉽다.

4. **참조 무결성 제약 불가**: `member_ids` 컬럼은 문자열(`VARCHAR`) 타입이므로 `member` 테이블의 `member_id` (`BIGINT`)를 참조하는 **외래 키를 설정할 수 없다.** 데이터베이스 레벨에서 데이터의 정합성을 보장할 방법이 사라지는 것이다. 예를 들어 존재하지 않는 회원 ID인 `999` 를 `member_ids` 에 `'1,2,999'` 와 같이 추가해도 데이터베이스는 막지 못한다.

결론적으로, 컬럼 하나에 여러 값을 저장하는 방식은 관계형 데이터베이스의 장점을 모두 포기하는 최악의 설계이므로

절대로 사용해서는 안 된다.

'일(1)' 쪽에 외래 키를 두면 발생하는 문제3 - 여러 컬럼에 나누어 보관

그렇다면 컬럼을 나누어 보관하면 어떨까? `member_id1`, `member_id2` ... 처럼 말이다.

```
DROP TABLE IF EXISTS team_bad3;
CREATE TABLE team_bad3 (
    team_id BIGINT NOT NULL AUTO_INCREMENT,
    name VARCHAR(50) NOT NULL,
    member_id_1 BIGINT NULL, -- 첫 번째 팀원
    member_id_2 BIGINT NULL, -- 두 번째 팀원
    member_id_3 BIGINT NULL, -- 세 번째 팀원 ... 언제까지 추가할 것인가?
    PRIMARY KEY (team_id)
);

-- 데이터 삽입
INSERT INTO team_bad3(name, member_id_1, member_id_2)
VALUES ('개발팀', 1, 2); -- 김개발(ID:1), 박개발(ID:2);
```

[저장된 결과]

team_id	name	member_id_1	member_id_2	member_id_3
1	개발팀	1	2	

이 설계는 다음과 같은 심각한 문제들을 야기한다.

1. **확장성의 부재**: 팀원이 3명에서 4명으로 늘어나면 어떻게 할 것인가? `ALTER TABLE`을 사용해 `member_id_4` 컬럼을 추가해야만 한다. 팀원 수의 제한이 없는 비즈니스 요구사항을 전혀 반영하지 못한다. 이는 팀원 수에 따라 테이블 구조가 계속 바뀌어야 하는 끔찍한 설계가 된다.

2. **공간 낭비**: 대부분의 팀이 2~3명인데, 최대 팀원 수를 10명으로 가정하고 `member_id_10`까지 컬럼을 만들어 두면, 나머지 `NULL` 컬럼들이 엄청난 공간을 낭비하게 된다.

3. **데이터 검색의 어려움**: "'박개발(ID:2)' 회원이 어느 팀 소속인가?"를 찾으려면 어떻게 해야 할까?

```
SELECT name FROM team_bad3
```

```
WHERE member_id_1 = 2 OR member_id_2 = 2 OR member_id_3 = 2;
```

이런 쿼리는 작성하기도 어렵고, 모든 `member_id` 컬럼을 확인해야 하므로 성능도 매우 나쁘다. 인덱스를 활용하기도 까다롭다.

'다(N)'쪽에 외래 키 두기

반면, '다(N)' 쪽인 `회원` 테이블에 `team_id`라는 컬럼 하나만 추가하면 모든 문제가 간단히 해결된다.

실제 데이터가 테이블에 어떻게 저장되는지를 보면 왜 '다(N)' 쪽에 외래 키를 두는 것이 유일하게 올바른 방법인지 명확하게 이해할 수 있다. 먼저 각 테이블에 데이터가 어떻게 저장되는지 확인해 보자.

`team` 테이블 (1 쪽)

팀 정보는 각 팀마다 단 하나의 행만 차지한다. 매우 깔끔하고 중복이 없다.

team_id (PK)	name
1	개발팀
2	디자인팀

`member` 테이블 (N 쪽)

각 회원은 고유한 행을 가지며, `team_id` 컬럼에 자신이 속한 팀의 ID **'단 하나'**만 저장한다.

member_id (PK)	team_id (FK)	name
1	1	김개발
2	1	박개발
3	2	최디자인
4	NULL	이기획

각 회원은 자신이 속한 팀의 `team_id`만 저장하면 된다. 10명의 회원이든 100명의 회원이든 각자 자신의 행에 소속 팀 ID 하나만 가지면 되므로 데이터 구조가 매우 안정적이고 확장성 있게 유지된다.

이 구조의 핵심을 보자.

- **원자성 준수**: `member` 테이블의 `team_id` 컬럼에는 언제나 **값 하나**만 들어간다. '김개발'의 소속팀은 `1` 이라는

값 하나로 명확하게 표현된다. 이것이 관계형 데이터베이스의 가장 기본 원칙이다.

- **유연성**: 팀에 소속되지 않은 '이기획' 같은 회원도 `NULL`을 허용함으로써 표현할 수 있다.
- **확장성**: '개발팀'에 신입사원 100명이 들어와도 `team` 테이블은 그대로이고, `member` 테이블에 100개의 행이 추가될 뿐이다. 테이블의 구조는 전혀 바꿀 필요가 없다.

이것이 바로 **관계의 주인을 '다(N)' 쪽으로 정하고, 외래 키를 '다(N)' 쪽 테이블에 두는 이유**다.

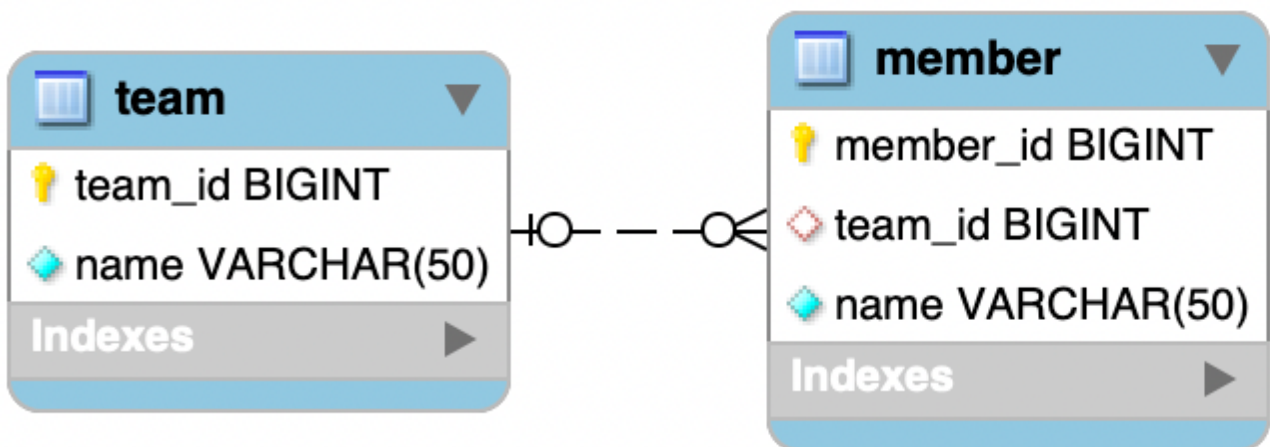
부모 테이블과 자식 테이블

외래 키 관계를 이야기할 때, 우리는 두 테이블을 **부모(Parent)**와 **자식(Child)**으로 표현한다.

- **부모 테이블**: 관계에서 '일(1)'에 해당하는 테이블로, 다른 테이블에 의해 참조되는 쪽이다. **기본 키(Primary Key)**를 가지고 있다. (`team` 테이블)
- **자식 테이블**: 관계에서 '다(N)'에 해당하는 테이블로, 다른 테이블을 참조하는 쪽이다. **외래 키(Foreign Key)**를 가지고 있다. (`member` 테이블)

자식은 부모에게 의존한다. 즉, `member`는 `team`에 의존한다. 존재하지 않는 팀(`team_id`가 없는)에 회원이 소속될 수는 없다. 이처럼 데이터베이스는 외래 키 제약조건을 통해 부모 테이블에 존재하지 않는 데이터가 자식 테이블에 입력되는 것을 막아 **참조 무결성(Referential Integrity)**을 지켜준다.

지금까지 설명한 개념을 반영하여 올바른 일대다, 다대일 관계 테이블을 설계해 보자.



테이블 생성

`team` 테이블 생성 (관계에서 '1' 쪽, 부모 테이블)

관계를 맺기 위해서는 참조의 대상이 되는 부모 테이블이 먼저 존재해야 한다.

```
```

```sql
DROP TABLE IF EXISTS member;
DROP TABLE IF EXISTS team;

CREATE TABLE team (
    team_id   BIGINT       NOT NULL AUTO_INCREMENT,
    name      VARCHAR(50)  NOT NULL,
    PRIMARY KEY (team_id),
    UNIQUE KEY uq_name (name)
);
```

`member` **테이블 생성 (관계에서 'N' 쪽, 자식 테이블)**

이제 `team`을 참조하는 `member` 테이블을 만들자.

```sql
DROP TABLE IF EXISTS member;

CREATE TABLE member (
    member_id   BIGINT       NOT NULL AUTO_INCREMENT,
    team_id     BIGINT       NULL, -- 팀에 소속되지 않은 회원이 있을 수 있다면 NULL 허용
    name        VARCHAR(50)  NOT NULL,
    PRIMARY KEY (member_id),
    CONSTRAINT fk_member_team FOREIGN KEY (team_id)
        REFERENCES team (team_id)
);
```

- `CONSTRAINT fk_member_team`: 외래 키 제약조건에 `fk_member_team` 이라는 명시적인 이름을 부여했다. 이렇게 하면 오류 발생 시 원인을 파악하기 쉽다.
- `FOREIGN KEY (team_id) REFERENCES team (team_id)`: `member` 테이블의 `team_id`가 `team` 테이블의 `team_id`를 참조하도록 설정한다. 이로써 두 테이블 간의 관계가 완성된다.

데이터 삽입 및 결과 확인

데이터가 잘 입력되었는지 각 테이블을 조회해서 확인해 보자.

```sql
-- 팀 데이터 삽입
INSERT INTO team(name) VALUES ('개발팀'), ('디자인팀');

-- 김개발, 박개발 -> 개발팀 (team_id=1)
INSERT INTO member(team_id, name)
VALUES (1, '김개발'),
       (1, '박개발');
```

```sql
-- 최디자인 -> 디자인팀 (team_id=2)
INSERT INTO member(team_id, name)
VALUES (2, '최디자인');

-- 이기획 -> 팀 없음 (team_id=NULL)
INSERT INTO member(team_id, name)
VALUES (NULL, '이기획');
```

각 테이블의 데이터를 조회해 보자.

`team` **테이블 조회**

```sql
SELECT * FROM team;
```

[실행 결과]

team_id	name
1	개발팀
2	디자인팀

`member` **테이블 조회**

```sql
SELECT * FROM member;
```

[실행 결과]

member_id (PK)	team_id (FK)	name
1	1	김개발
2	1	박개발
3	2	최디자인
4	NULL	이기획

결과를 보면 `member` 테이블의 `team_id` 컬럼이 외래 키 역할을 하면서 각 회원이 어떤 팀에 속해있는지 명확하게 보여준다. '김개발'과 '박개발'은 `team_id` 가 1인 '개발팀'에, '최디자인'은 `team_id` 가 2인 '디자인팀'에 소속된 것을 확인할 수 있다.

이처럼 '다(N)' 쪽인 `member` 테이블에 외래 키를 두는 것이 일대다 관계를 표현하는 올바른 방법이다.

일대다(1:N) 다대일(N:1) 관계 - 외래 키 위치2

정리 - 일대다 관계의 외래 키 위치

지금까지 학습한 내용을 간단하게 정리해보자.

`team` 테이블 (1 쪽)에 외래 키를 두는 방법
'개발팀' 행 안에 '김개발(1)'과 '박개발(2)'의 정보를 모두 담아야 한다. 방법이 없으니 문자열로 억지로 저장한 모습이다.

[team 테이블]

team_id (PK)	name	member_ids
1	개발팀	`1:김개발, 2:박개발`
2	디자인팀	3:최디자인

- `team` 테이블에서 '개발팀(`team_id:1`)'은 자신의 팀에 속한 여러 회원을 알아야 한다.
- `team` 테이블에서 '개발팀'은 하나의 행에만 존재할 수 있다. '개발팀'이라는 하나의 행에 여러 회원들의 ID를 억지로 보관할 수는 있지만 앞서 설명한 수 많은 부작용이 발생한다.

`member` 테이블 (N 쪽)에 외래 키를 두는 방법
각 회원은 고유한 행을 가지며, `team_id` 컬럼에 자신이 속한 팀의 ID **'단 하나'**만 저장하면 된다.

[member 테이블]

member_id (PK)	name	team_id (FK)
1	김개발	1

2	박개발	1
3	최디자인	2
4	이기획	NULL

- `member` 테이블에서 김개발(`member_id:1`) 회원은 하나의 팀에만 속하면 된다. 따라서 `team_id` 컬럼에 자신이 속한 팀의 ID **'단 하나'**만 저장하면 된다.

'다(N)' 쪽의 개체(회원)는 '일(1)' 쪽의 개체(팀)를 단 하나만 참조하기 때문에, `member` 테이블에 `team_id`라는 단일 컬럼을 두는 것으로 관계가 완벽하게 표현된다.
다시 정리하자면 이처럼 '다(N)' 쪽에 외래 키를 두는 것이 일대다 관계를 표현하는 올바른 방법이다.

관계형 데이터베이스의 법칙과 외래 키의 위치

이번에는 이론적인 관점에서 외래 키가 '다(N)'에 위치해야 하는 이유를 알아보자.

관계형 데이터베이스에서 테이블의 행(row)은 다음과 같은 법칙을 가진다.

1. **기본 키(PK)는 반드시 유일하다.**
   - 한 테이블 안에서 동일한 PK를 가진 행은 존재할 수 없다.
2. **외래 키(FK)는 단일 행만 참조한다.**
   - 외래 키 값 하나는 상대 테이블의 정확히 한 행만 가리킨다.
3. **컬럼은 원자적(atomic)이어야 한다.**
   - 하나의 컬럼에 여러 값을 넣을 수 없다(제1정규형).

관계의 법칙: 왜 외래 키는 '다(N)'에만 존재할 수 있는가?
앞서 살펴본 3가지 기본 제약은 관계형 데이터베이스가 데이터를 일관되고 안정적으로 유지하기 위한 최소한의 약속이다. 이 약속들을 기반으로 왜 외래 키가 반드시 '다(N)' 쪽에 위치해야만 하는지 증명해 보자.

'일(1)' 쪽 테이블의 딜레마
만약 '일(1)' 쪽인 `team` 테이블에 외래 키를 두어 소속된 회원을 표현한다고 가정해 보자.
'개발팀'은 여러 명의 회원을 가져야 한다는 비즈니스 요구사항이 있다. 하지만 데이터베이스의 제약 때문에 `team` 테이블은 이 요구사항을 담아낼 수 없다.

[문제 상황1]

team_id	name	member_id
1	개발팀	1(김개발)
1(???) ➔ 불가능	개발팀	2(박개발)

1. `team` 테이블의 기본 키(`team_id`)는 유일해야 하므로(제약 1), '개발팀'은 **단 하나의 행**으로만 존재해야 한다. 여러 행에 걸쳐 '개발팀'을 저장할 수 없다.

[문제 상황2]

team_id	name	member_id
1	개발팀	1(김개발), 2(박개발) ➔ 불가능

2. 하나의 행에 있는 컬럼은 원자적이어야 하므로(제약 3), `member_id` 와 같은 컬럼에는 **단 하나의 값**만 저장할 수 있다. 여러 회원의 ID를 하나의 컬럼에 넣는 것은 불가능하다.

이 두 가지 제약이 충돌하면서 논리적인 모순이 발생한다. '개발팀'이라는 **하나의 행**에 여러 회원의 정보를 담아야 하는데, 컬럼에는 **하나의 값**만 넣을 수 있다. 이는 마치 하나의 상자에 물건을 하나만 넣을 수 있는데, 여러 개의 물건을 넣으라는 요구와 같다. 이 문제를 해결할 방법은 관계형 데이터베이스의 규칙 안에서는 존재하지 않는다.

'다(N)' 쪽 테이블의 명쾌한 해답

반대로 '다(N)' 쪽인 `member` 테이블에 외래 키(`team_id`)를 두는 경우를 보자. 모든 것이 명쾌하게 해결된다.

member_id (PK)	team_id (FK)	name
1	1	김개발
2	1	박개발
3	2	최디자인
4	NULL	이기획

1. '김개발'이라는 회원은 오직 **하나의 팀**에만 소속된다.
2. 따라서 `member` 테이블의 '김개발' 행에 있는 `team_id` 컬럼에는 **단 하나의 값**만 저장하면 된다. 이는 컬럼의 원자성(제약 3)을 완벽하게 만족한다.
3. 이 `team_id` 값은 `team` 테이블의 유일한 하나의 행을 정확히 가리킨다. 즉, 외래 키가 단일 행만 참조한다는 제약(제약 2)을 만족한다.

4. 물론 `member` 테이블의 기본 키(`member_id`)도 유일성을 유지하며(제약 1), `team` 테이블의 기본 키도 마찬가지다.

'박개발' 회원도, '최디자인' 회원도 각자의 행에서 자신이 속한 팀의 ID 하나만 `team_id` 컬럼에 저장하면 그만이다. 어떤 제약과도 충돌하지 않으며, 모든 규칙을 완벽하게 준수한다.

결론적으로, '일대다' 관계에서 외래 키를 '다(N)' 쪽에 두는 것은 선택의 문제가 아니라, **관계형 데이터베이스의 기본 제약을 모두 만족시키는 유일하고 올바른 방법**이다. 이것이 관계형 데이터베이스가 관계를 표현하는 근본적인 원리이다.

일대다(1:N) 다대일(N:1) 관계 - 조인과 뻥튀기

일대다와 다대일은 같은 관계지만, 어떤 테이블을 기준으로 조인(JOIN)하느냐에 따라 결과의 형태가 완전히 달라진다.
조인을 했을 때 기준 테이블의 행 수보다 결과 행의 수가 더 늘어날 수 있는데 이것을 데이터 뻥튀기라고 한다.
이 '데이터 뻥튀기' 현상을 이해하는 것은 SQL 쿼리 작성의 핵심이다.

> **조인과 뻥튀기 현상에 대한 자세한 내용은 기본편을 참고하자.**
> 이 내용은 기본편에서 많은 시간을 들여서 매우 자세히 설명했다.
> 중요한 내용이므로 복습 차원에서 간략하게 설명하겠다.
> 원리와 자세한 내용은 실전 데이터베이스 기본편 ➔ 3. 조인2 - 외부 조인과 기타 조인 ➔ 조인의 특징을 참고하자.

앞서 팀, 회원 데이터 예제를 그대로 사용하겠다.

다대일(N:1) 조인: 데이터가 뻥튀기되지 않는다.

'다(N)' 쪽인 `member` 테이블을 기준으로 '일(1)' 쪽인 `team` 테이블을 조인해 보자. 모든 회원의 이름과 그들이 속한 팀의 이름을 조회하는 쿼리다.

먼저 `member` 테이블만 조회해서 기준이 되는 데이터의 수를 확인하자.

```
SELECT * FROM member;
```

[실행 결과]

member_id	team_id	name
1	1	김개발
2	1	박개발
3	2	최디자인
4	NULL	이기획

총 4개의 행이 있다. 이제 조인을 사용해서 이 `member` 테이블에 `team` 정보를 붙여보자.

```
SELECT
    m.name AS member_name,
    t.name AS team_name
FROM
    member m
LEFT JOIN
    team t ON m.team_id = t.team_id;
```

- '이기획'처럼 `team` 이 없는 회원도 확인하려면 LEFT JOIN을 사용하면 된다.

[실행 결과]

member_name	team_name
김개발	개발팀
박개발	개발팀
최디자인	디자인팀
이기획	NULL

결과를 보면, 조인 전 `member` 테이블의 행 수(4개)와 정확히 일치하는 4개의 행이 출력되었다. 각 회원은 단 하나의 팀에만 속하기 때문에, `member` 를 기준으로 `team` 정보를 가져와도 결과 행의 수가 늘어나지 않는다. 매우 예측 가능하고 안정적인 조인이다.

일대다(1:N) 조인: 데이터가 뻥튀기된다!

이번에는 반대로, '일(1)' 쪽인 `team` 테이블을 기준으로 '다(N)' 쪽인 `member` 테이블을 조인해 보자.

먼저 기준이 되는 `team` 테이블의 데이터를 확인하자.

```
SELECT * FROM team;
```

[실행 결과]

team_id	name
1	개발팀
2	디자인팀

총 2개의 행이 있다. 이제 여기에 소속된 `member` 정보를 붙여보자.

```
SELECT
    t.name AS team_name,
    m.name AS member_name
FROM
    team t
JOIN
    member m ON t.team_id = m.team_id;
```

- 소속 팀이 없는 '이기획' 회원은 제외했다. 참고로 이 쿼리에 RIGHT JOIN을 사용하면 이기획 회원도 조회할 수 있다.

[실행 결과]

team_name	member_name
개발팀	김개발
개발팀	박개발
디자인팀	최디자인

분명 `team` 테이블에는 행이 2개뿐이었지만, 조인 결과는 3개의 행이 나왔다. '개발팀'은 2명의 팀원을 가지고 있으므로, 조인 과정에서 '개발팀' 행이 2개로 복제되어 각 팀원과 연결되었다. 이것이 바로 '1' 쪽에서 'N' 쪽을 조인할 때 발생하는 **데이터 뻥튀기 현상**이다.

이 현상 자체는 오류가 아니다. 오히려 이런 특징을 활용하여 '특정 팀에 속한 모든 팀원 목록' 같은 기능을 구현하는 것이다. 다만, `GROUP BY` 나 `COUNT` 같은 집계 함수를 사용할 때 이 데이터 뻥튀기를 인지하지 못하면 잘못된 결과를 얻을 수 있으므로 반드시 이해하고 넘어가야 한다.

> **조인과 뻥튀기 간단 정리**
> - 다대일 조인
>   - FK ➜ PK로 조인한다.
>   - 기준 테이블의 데이터보다 더 많이 조회되는 뻥튀기 현상이 발생하지 않는다.
> - 일대다 조인
>   - PK ➜ FK로 조인한다.
>   - 기준 테이블의 데이터보다 더 많이 조회되는 뻥튀기 현상이 발생할 수 있다.
>
> 자세한 내용은 실전 데이터베이스 기본편 ➜ 3. 조인2 - 외부 조인과 기타 조인 ➜ 조인의 특징을 참고하자.

정리

논리적 모델링 - 관계

- 개념적 모델링의 추상적인 관계는 논리적 모델링에서 외래 키를 통해 구체적으로 구현된다.
- 관계형 데이터베이스의 관계는 방향성이 없으며, 외래 키 하나를 통해 양방향으로 데이터를 조회할 수 있다.
- 관계는 한 테이블의 행이 다른 테이블의 행과 몇 개나 연결되는지를 나타내는 **카디널리티(Cardinality)**와 관계 참여 여부를 나타내는 **참여도(Optionality)**라는 두 가지 핵심 속성을 가진다.

참여도

- 참여도는 한 엔티티가 관계에 반드시 참여해야 하는지(필수적) 또는 참여하지 않아도 되는지(선택적)를 나타낸다.
- 논리적 모델링에서는 외래 키 컬럼의 `NULL` 허용 여부(`NOT NULL` 또는 `NULL`)로 참여도를 구현한다.
- 외래 키가 있는 '다(N)' 쪽 테이블의 참여도는 `NULL` 제약조건으로 간단히 제어할 수 있다.
- 반면, 외래 키가 없는 '일(1)' 쪽 테이블의 필수 참여(예: 팀은 최소 1명 이상의 회원을 가져야 한다)는 데이터베이

스의 기본 제약만으로는 강제할 수 없으며, 보통 애플리케이션 계층에서 로직으로 처리한다.

- 데이터베이스 제약으로 구현할 수 없더라도, ERD에는 원래의 비즈니스 규칙을 명확하게 표현해야 한다.

일대다(1:N) 다대일(N:1) 관계 - 외래 키 위치1

- 일대다(1:N) 또는 다대일(N:1) 관계는 실무에서 가장 흔하게 사용되는 관계이다.
- 관계를 표현하기 위한 외래 키는 **반드시 '다(N)' 쪽에 위치해야 한다.**
- '일(1)' 쪽에 외래 키를 두면 제1 정규형 위반, 데이터 검색 및 수정의 어려움, 확장성 부재 등 심각한 문제가 발생하므로 절대 사용해서는 안 된다.
- '다(N)' 쪽에 외래 키를 두면 데이터의 원자성을 준수하면서 유연하고 확장성 있는 구조를 만들 수 있다.
- 두 테이블의 관계에서 외래 키가 있는 테이블을 **자식 테이블**, 외래 키가 없는 테이블을 **부모 테이블**이라 한다.
- 결과적으로 외래 키가 있는 다(N)에 해당하는 테이블이 자식 테이블, 외래 키가 없는 일(1)에 해당하는 테이블이 부모 테이블이 된다.

일대다(1:N) 다대일(N:1) 관계 - 외래 키 위치2

- 외래 키가 '다(N)' 쪽에 위치해야 하는 것은 선택이 아니라, 관계형 데이터베이스의 기본 제약(PK 유일성, FK의 단일 행 참조, 컬럼의 원자성)을 모두 만족시키는 유일하고 올바른 방법이다.
- '일(1)' 쪽에 외래 키를 두면 하나의 행에 여러 개의 값을 저장해야 하는 모순이 발생하여 기본 제약을 위반하게 된다.

일대다(1:N) 다대일(N:1) 관계 - 조인과 뻥튀기

- **일대다(1:N) 조인**: '일(1)' 쪽을 기준으로 '다(N)' 쪽을 조인하면, 기준 테이블의 행이 N개의 데이터만큼 복제되어 결과 행 수가 늘어나는 **데이터 뻥튀기 현상이 발생할 수 있다.**
- **다대일(N:1) 조인**: '다(N)' 쪽을 기준으로 '일(1)' 쪽을 조인하면, 기준 테이블의 행 수가 늘어나는 데이터 **뻥튀기 현상이 발생하지 않는다.**