Google Developer Student Clubs

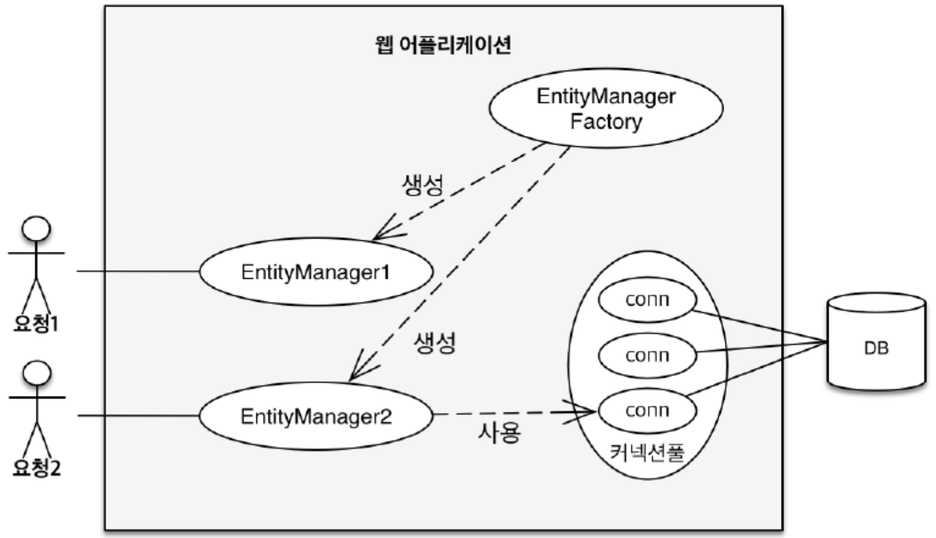
# Backend 6th Study

# Contents



# 영속성 컨텍스트

## 엔티티 매니저 팩토리와 엔티티 매니저



# 영속성 컨텍스트

JPA를 이해하는데 가장 중요한 용어

엔티티를 영구 저장하는 환경이라는 뜻

```
EntityManager.persist(entity);
```

## 엔티티의 생명주기

# 엔티티?: 테이블에 대응하는 하나의 클래스

**비영속 (new/transient)**
영속성 컨텍스트와 전혀 관계가 없는 **새로운** 상태

**영속 (managed)**
영속성 컨텍스트에 **관리**되는 상태

**준영속 (detached)**
영속성 컨텍스트에 저장되었다가 **분리**된 상태

**삭제 (removed)**
**삭제**된 상태

## 엔티티의 생명주기

# 엔티티?: 테이블에 대응하는 하나의 클래스

**비영속 (new/transient)**
영속성 컨텍스트와 전혀 관계가 없는 **새로운** 상태

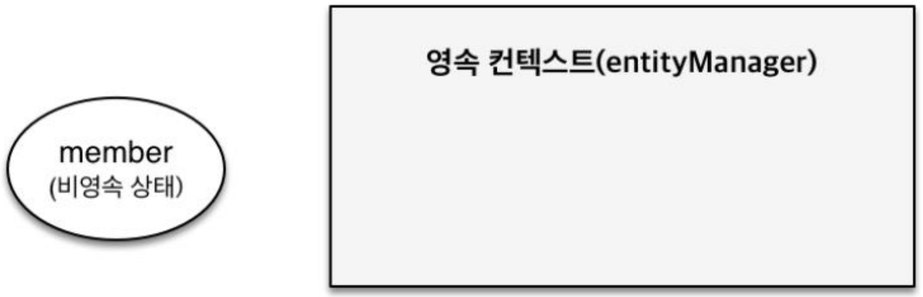
**영속 (managed)**
영속성 컨텍스트에 **관리**되는 상태

**준영속 (detached)**
영속성 컨텍스트에 저장되었다가 **분리**된 상태

**삭제 (removed)**
**삭제**된 상태

# 엔티티의 생명주기

## 비영속




```
//객체를 생성한 상태(비영속)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");
```

# 엔티티의 생명주기

## 영속


```
//객체를 생성한 상태(비영속)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");

EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

//객체를 저장한 상태(영속)
em.persist(member);
```

# 영속성 컨텍스트의 이점

1차 캐시

동일성(identity) 보장

트랜잭션을 지원하는 쓰기 지연
(transactional write-behind)

변경 감지(Dirty Checking)

지연 로딩(Lazy Loading)

# 영속성 컨텍스트의 이점

## 엔티티 수정

### 변경 감지

```
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();
transaction.begin();  // [트랜잭션] 시작

// 영속 엔티티 조회
Member memberA = em.find(Member.class, "memberA");

// 영속 엔티티 데이터 수정
memberA.setUsername("hi");
memberA.setAge(10);

//em.update(member) 이런 코드가 있어야 하지 않을까?

transaction.commit(); // [트랜잭션] 커밋
```

# 엔티티 수정
## 변경 감지

```
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();
transaction.begin();  // [트랜잭션] 시작

// 영속 엔티티 조회
Member memberA = em.find(Member.class, "memberA");

// 영속 엔티티 데이터 수정
memberA.setUsername("hi");
memberA.setAge(10);

//em.update(member) 이런 코드가 있어야 하지 않을까?

transaction.commit(); // [트랜잭션] 커밋
```

# 플러시

## 영속성 컨텍스트의 변경내용을 데이터베이스에 반영

## 플러시

변경 감지

수정된 엔티티 쓰기 지연 SQL 저장소에 등록

쓰기 지연 SQL 저장소의 쿼리를 데이터베이스에 전송
(등록, 수정, 삭제 쿼리)

# 플러시

**em.flush() - 직접 호출**

**트랜잭션 커밋 - 플러시 자동 호출**

**JPQL 쿼리 실행 - 플러시 자동 호출**

# 플러시

영속성 컨텍스트를 비우지 않음

영속성 컨텍스트의 변경내용을 데이터베이스에 동기화

트랜잭션이라는 작업 단위가 중요 -> 커밋 직전에만 동기화 하면 됨

## 엔티티 매핑 소개

- 객체와 테이블 매핑: **@Entity, @Table**
- 필드와 컬럼 매핑: **@Column**
- 기본 키 매핑: **@Id**
- 연관관계 매핑: **@ManyToOne,@JoinColumn**

## @Entity

- @Entity가 붙은 클래스는 JPA가 관리, 엔티티라 한다.
- JPA를 사용해서 테이블과 매핑할 클래스는 **@Entity** 필수
- **주의**
  - **기본 생성자 필수**(파라미터가 없는 public 또는 protected 생성자)
  - final 클래스, enum, interface, inner 클래스 사용X
  - 저장할 필드에 final 사용 X

## 데이터베이스 스키마 자동 생성

- DDL을 애플리케이션 실행 시점에 자동 생성
- 테이블 중심 -> 객체 중심
- 데이터베이스 방언을 활용해서 데이터베이스에 맞는 적절한 DDL 생성
- 이렇게 **생성된 DDL은 개발 장비에서만 사용**
- 생성된 DDL은 운영서버에서는 사용하지 않거나, 적절히 다듬은 후 사용

# 데이터베이스 스키마 자동 생성 - 속성

## @Column

| 속성 | 설명 | 기본값 |
|---|---|---|
| name | 필드와 매핑할 테이블의 컬럼 이름 | 객체의 필드 이름 |
| insertable, updatable | 등록, 변경 가능 여부 | TRUE |
| nullable(DDL) | null 값의 허용 여부를 설정한다. false로 설정하면 DDL 생성 시에 not null 제약조건이 붙는다. | |
| unique(DDL) | @Table의 uniqueConstraints와 같지만 한 컬럼에 간단히 유니크 제약조건을 걸 때 사용한다. | |
| columnDefinition (DDL) | 데이터베이스 컬럼 정보를 직접 줄 수 있다.<br>ex) varchar(100) default 'EMPTY' | 필드의 자바 타입과 방언 정보를 사용해 |
| length(DDL) | 문자 길이 제약조건, String 타입에만 사용한다. | 255 |

# @Column

| 속성 | 설명 | 기본값 |
|---|---|---|
| name | 필드와 매핑할 테이블의 컬럼 이름 | 객체의 필드 이름 |
| insertable, updatable | 등록, 변경 가능 여부 | TRUE |
| nullable(DDL) | null 값의 허용 여부를 설정한다. false로 설정하면 DDL 생성 시에 not null 제약조건이 붙는다. | |
| unique(DDL) | @Table의 uniqueConstraints와 같지만 한 컬럼에 간단히 유니크 제약조건을 걸 때 사용한다. | |
| columnDefinition (DDL) | 데이터베이스 컬럼 정보를 직접 줄 수 있다.<br>ex) varchar(100) default 'EMPTY' | 필드의 자바 타입과 방언 정보를 사용해 |
| length(DDL) | 문자 길이 제약조건, String 타입에만 사용한다. | 255 |

# JPA

## 기본 키 매핑 어노테이션

- @Id
- @GeneratedValue

```
@Id @GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

## 기본 키 매핑 방법

- 직접 할당: **@Id만 사용**
- 자동 생성(**@GeneratedValue**)
  - **IDENTITY**: 데이터베이스에 위임, MYSQL
  - **SEQUENCE**: 데이터베이스 시퀀스 오브젝트 사용, ORACLE
    - @SequenceGenerator 필요
  - **TABLE**: 키 생성용 테이블 사용, 모든 DB에서 사용
    - @TableGenerator 필요
  - **AUTO**: 방언에 따라 자동 지정, 기본값

Google Developer Student Clubs

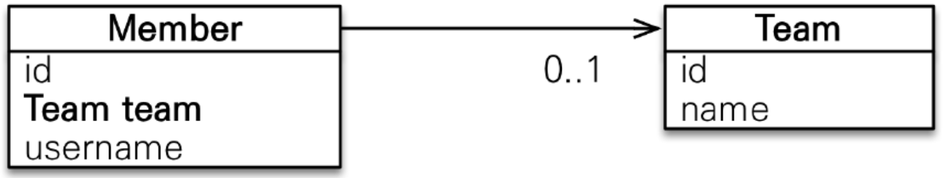
# 쉬는 시간

# 연관 관계

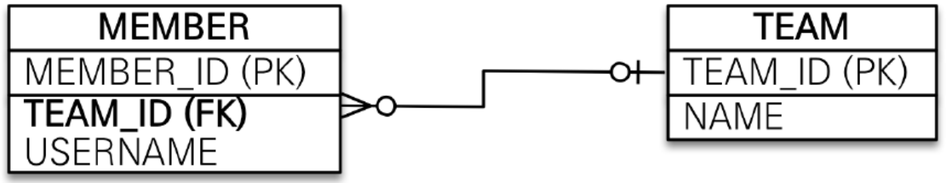
## 객체 지향 모델링
(객체 연관관계 사용)

---

[객체 연관관계]




[테이블 연관관계]



# 연관 관계

## 객체 지향 모델링

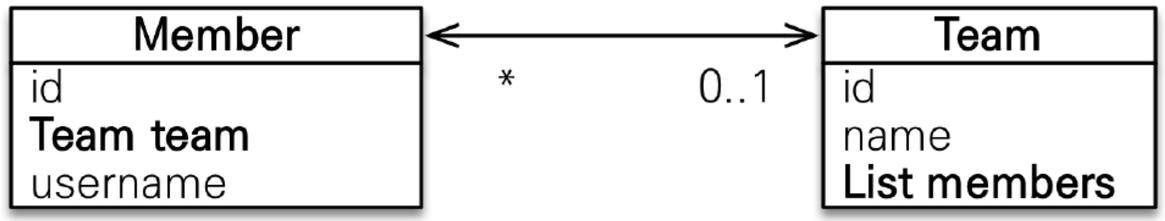
(객체의 참조와 테이블의 외래 키를 매핑)

```java
@Entity
public class Member {

    @Id @GeneratedValue
    private Long id;

    @Column(name = "USERNAME")
    private String name;
    private int age;

//    @Column(name = "TEAM_ID")
//    private Long teamId;

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

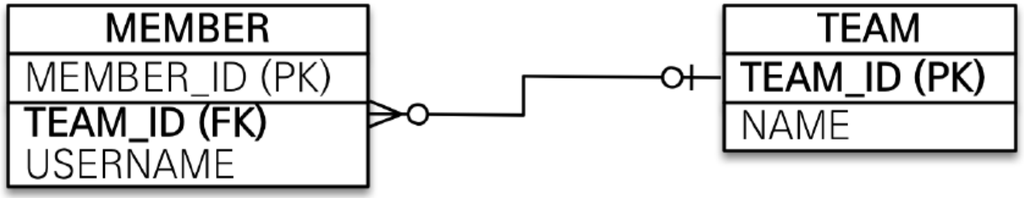
# 연관 관계

## 양방향 매핑

[양방향 객체 연관관계]




[테이블 연관관계]



# 연관 관계

## 양방향 매핑

(Team 엔티티는 컬렉션 추가)

```
@Entity
public class Team {

  @Id @GeneratedValue
  private Long id;

  private String name;

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

# 연관 관계

## 연관관계의 주인과 mappedBy

- mappedBy = JPA의 멘탈붕괴 난이도
- mappedBy는 처음에는 이해하기 어렵다.
- 객체와 테이블간에 연관관계를 맺는 차이를 이해해야 한다.

# 연관 관계

객체와 테이블이 관계를 맺는 차이

---

- **객체 연관관계 = 2개**
  - 회원 -> 팀 연관관계 1개(단방향)
  - 팀 -> 회원 연관관계 1개(단방향)
- **테이블 연관관계 = 1개**
  - 회원 <-> 팀의 연관관계 1개(양방향)

# 연관 관계

## 객체의 양방향 관계

- 객체의 **양방향 관계는 사실 양방향 관계가 아니라 서로 다른 단방향 관계 2개다.**
- 객체를 양방향으로 참조하려면 **단방향 연관관계를 2개** 만들어야 한다.
- A -> B (a.getB())
- B -> A (b.getA())

```
class A {
    B b;
}

class B {
    A a;
}
```

# 연관 관계

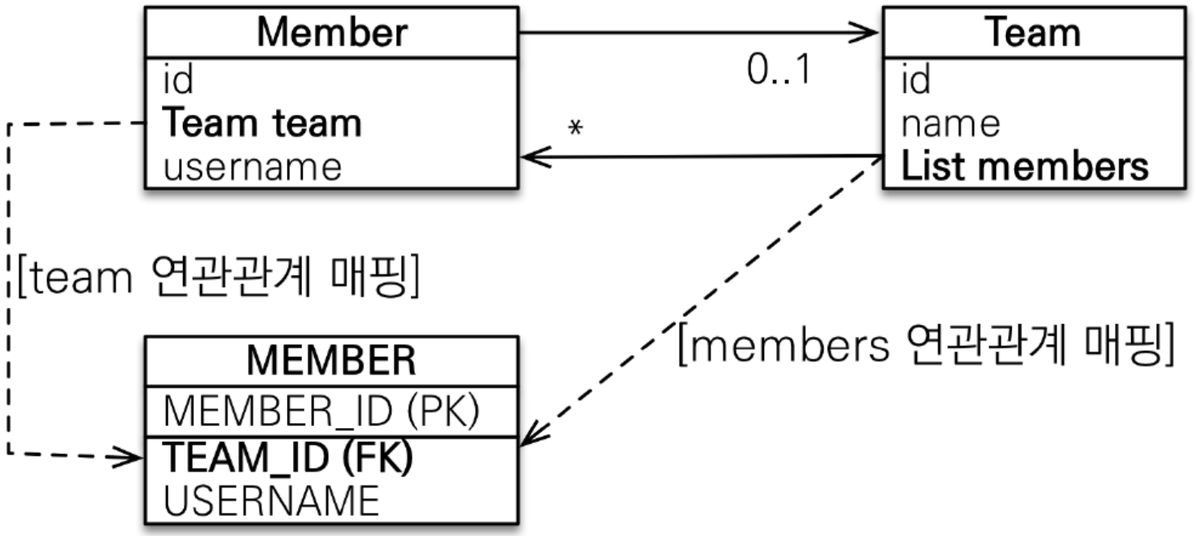
## 테이블의 양방향 연관관계

- 테이블은 **외래 키 하나**로 두 테이블의 연관관계를 관리
- MEMBER.TEAM_ID 외래 키 하나로 양방향 연관관계 가짐
  (양쪽으로 조인할 수 있다.)

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

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

# 연관 관계

둘 중 하나로 외래 키를 관리해야 한다.



# 연관 관계

## 연관관계의 주인(Owner)

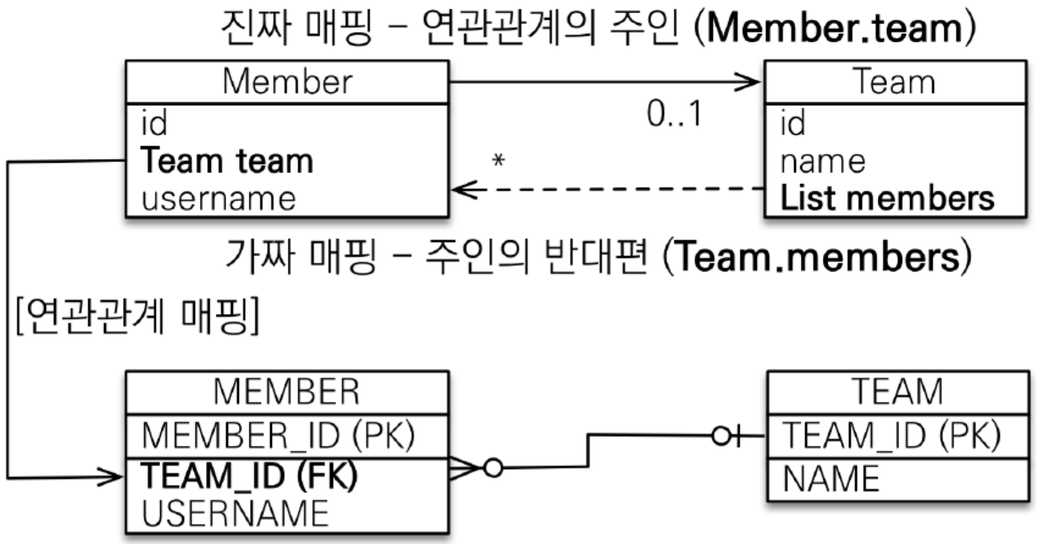
### 양방향 매핑 규칙

- 객체의 두 관계중 하나를 연관관계의 주인으로 지정
- **연관관계의 주인만이 외래 키를 관리(등록, 수정)**
- **주인이 아닌쪽은 읽기만 가능**
- 주인은 mappedBy 속성 사용X
- 주인이 아니면 mappedBy 속성으로 주인 지정

# 연관 관계

## 누구를 주인으로?

- 외래 키가 있는 있는 곳을 주인으로 정해라
- 여기서는 **Member.team**이 연관관계의 주인



# 연관 관계

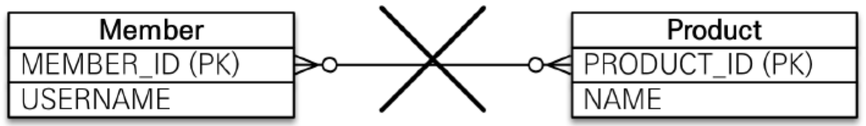
## 다중성

- 다대일: @ManyToOne
- 일대다: @OneToMany
- 일대일: @OneToOne
- 다대다: @ManyToMany

# 연관 관계

## 다대다

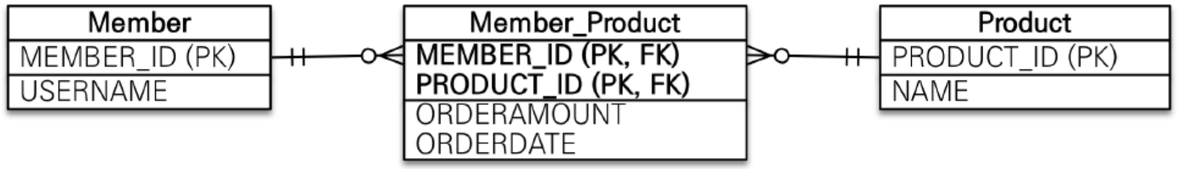
- 관계형 데이터베이스는 정규화된 테이블 2개로 다대다 관계를 표현할 수 없음
- 연결 테이블을 추가해서 일대다, 다대일 관계로 풀어내야함




# 연관 관계

## 다대다 매핑의 한계

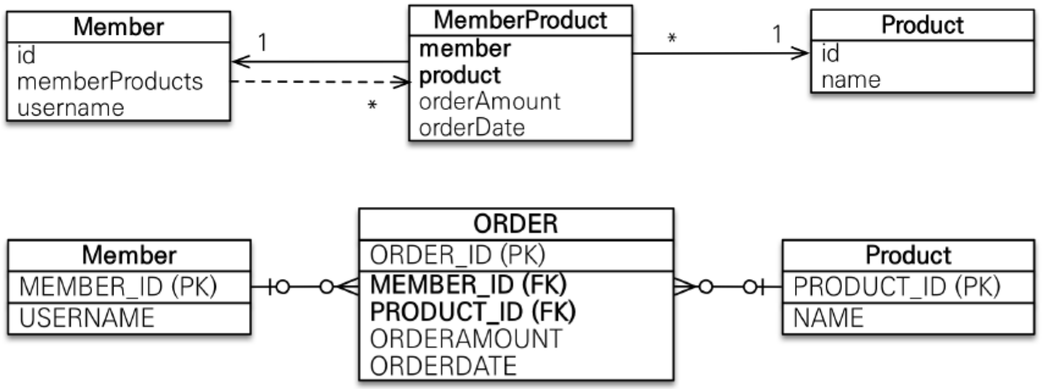
- **편리해 보이지만 실무에서 사용X**
- 연결 테이블이 단순히 연결만 하고 끝나지 않음
- 주문시간, 수량 같은 데이터가 들어올 수 있음



# 연관 관계

## 다대다 한계 극복

- 연결 테이블용 엔티티 추가(연결 테이블을 엔티티로 승격)
- @ManyToMany -> @OneToMany, @ManyToOne



Google Developer Student Clubs

# 수고하셨습니다~