

JPA

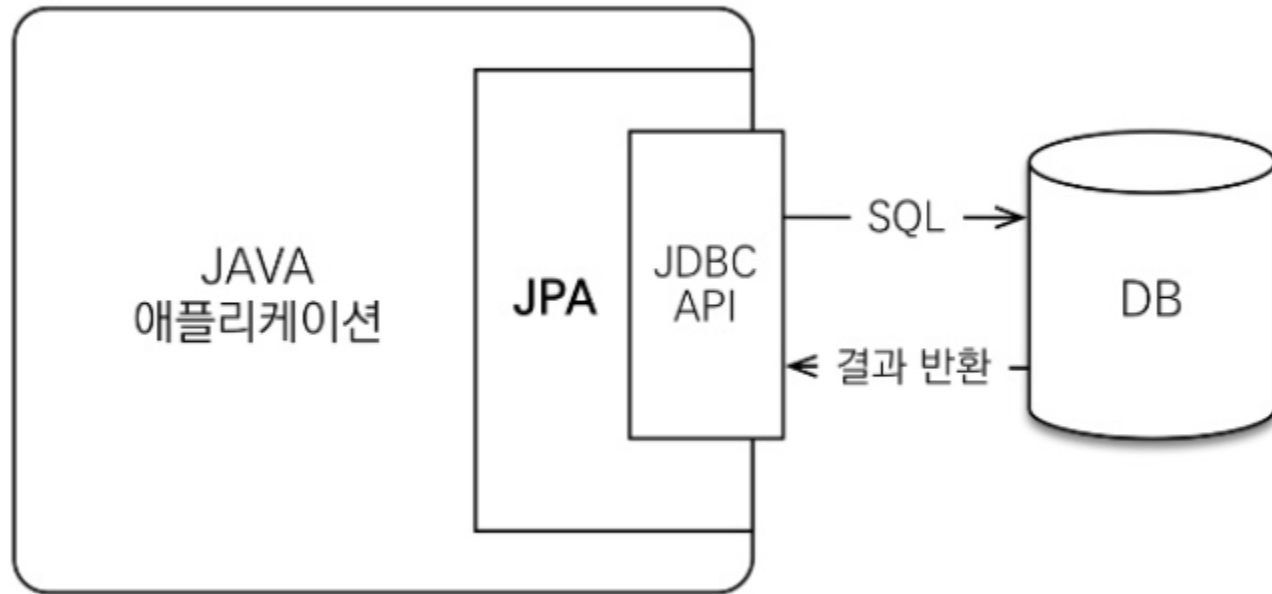
# JPA?

- Java Persistence API
- 자바 진영의 ORM 기술 표준

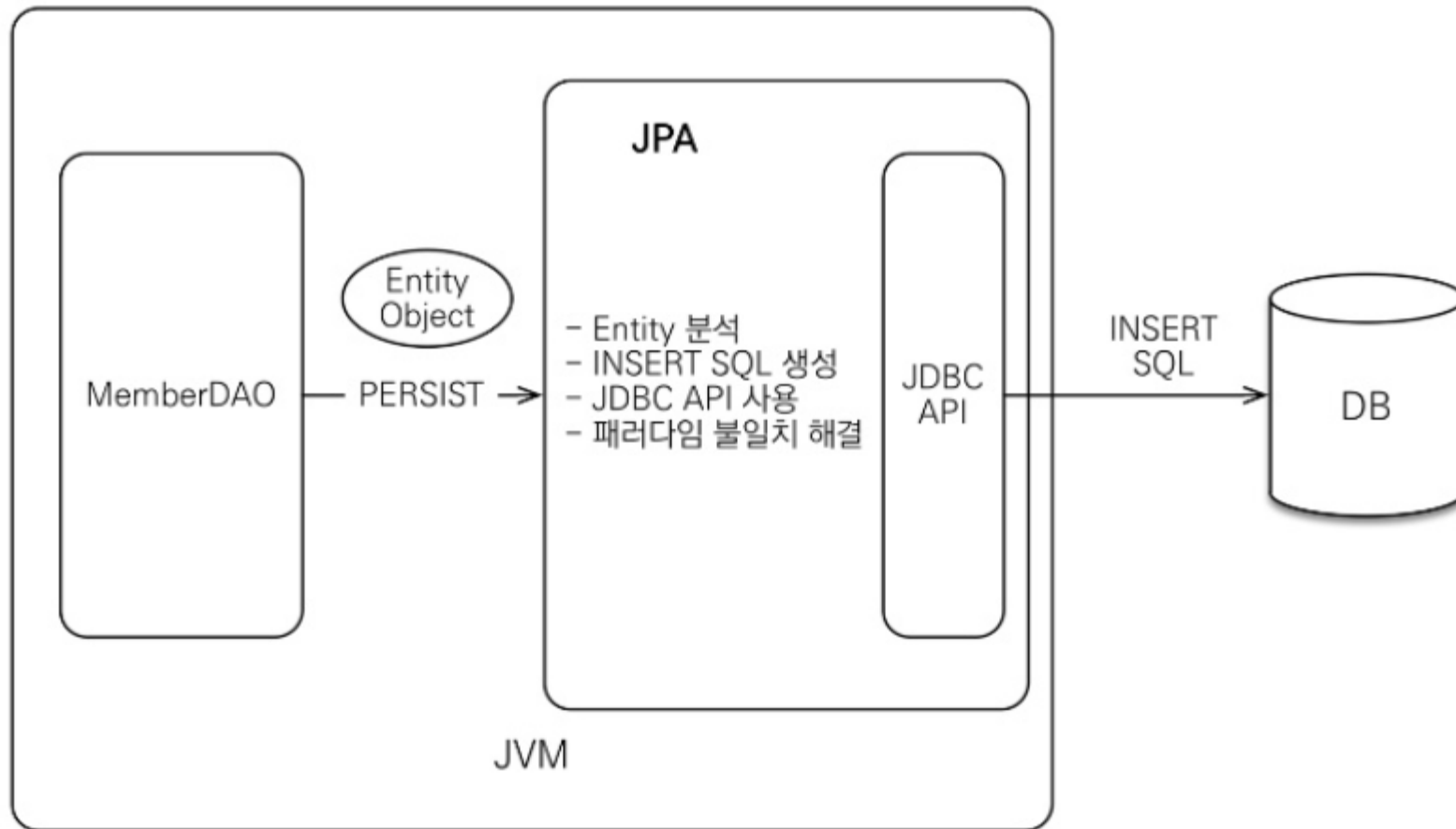
# ORM?

- Object-relational mapping
- 객체 관계 매핑
- 객체는 객체대로 설계
- 관계형 데이터베이스는 관계형 데이터베이스대로 설계
- ORM 프레임워크가 중간에서 매핑
- 대중적인 언어에는 대부분 ORM 기술이 존재

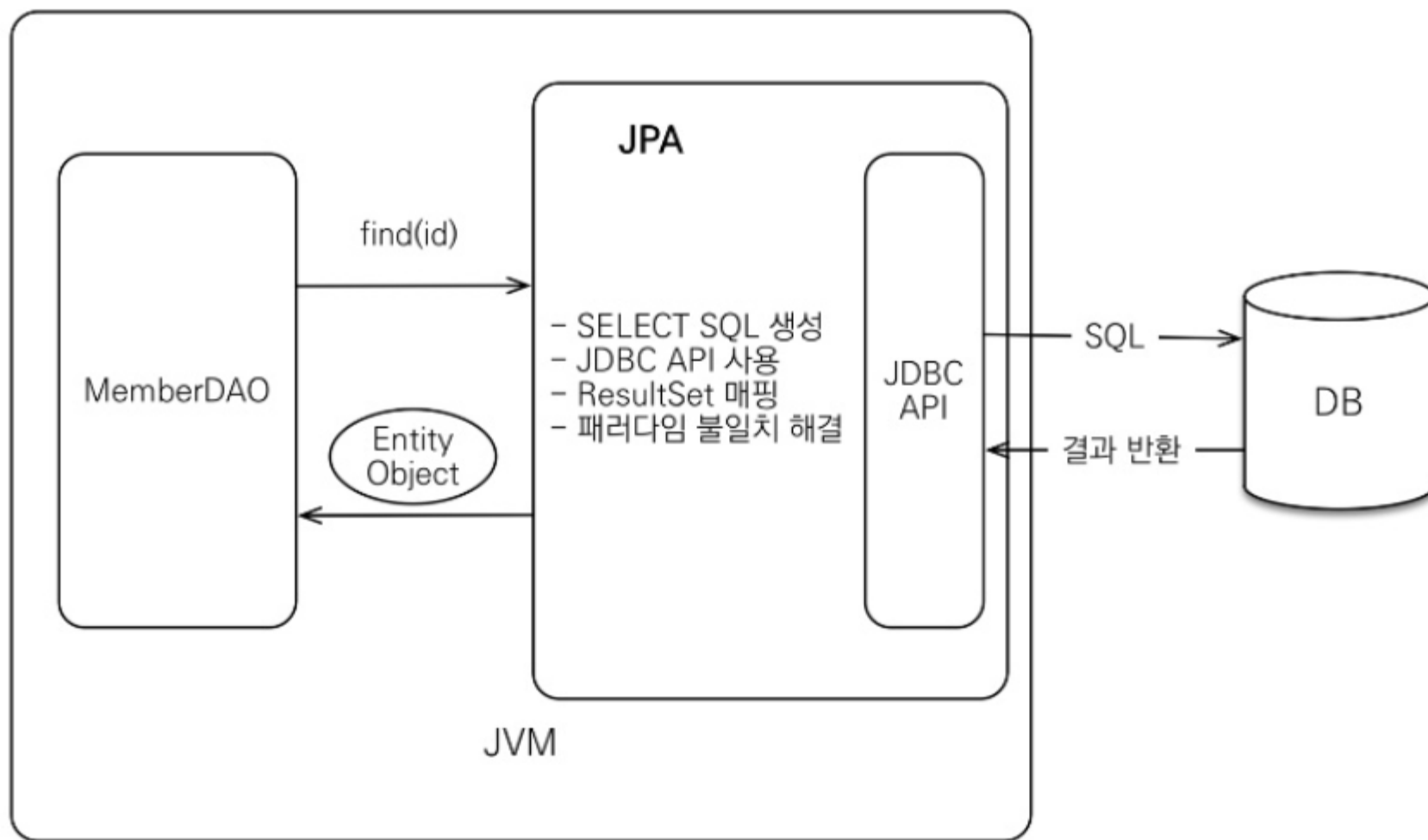
# JPA는 애플리케이션과 JDBC사이에서 동작



# JPA 동작 - 저장



# JPA동작 - 조회



# JPA 소개

하이버네이트(오픈 소스)

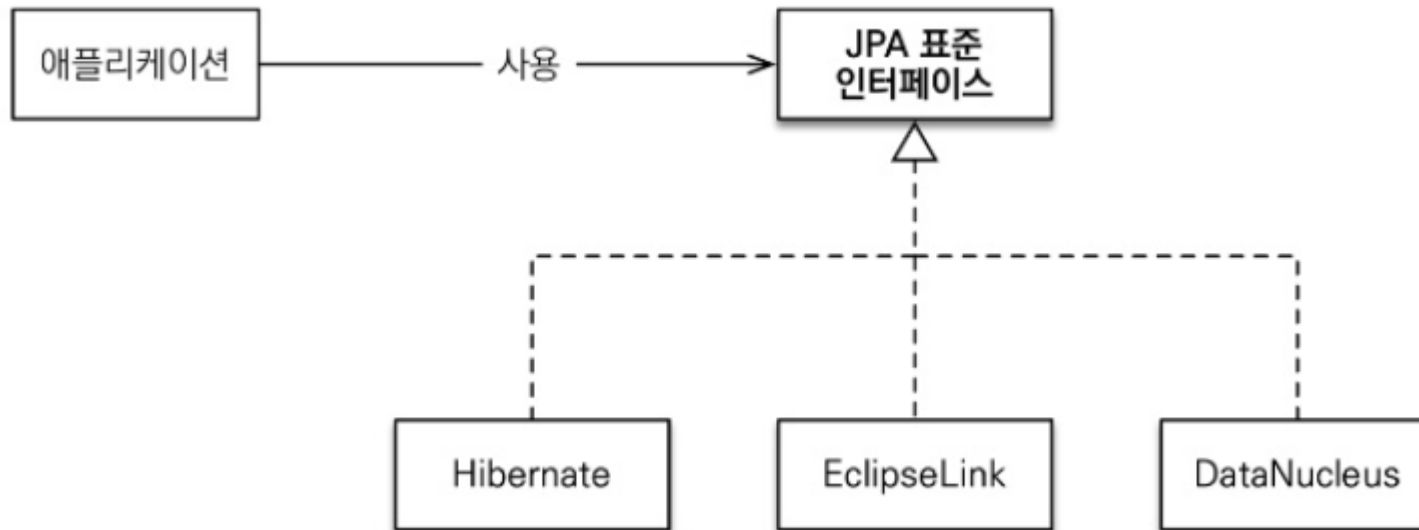


EJB - 엔티티 빈(자바 표준)

JPA(자바 표준)

# JPA는 표준 명세

- JPA는 인터페이스의 모음
- JPA 2.1 표준 명세를 구현한 3가지 구현체
- 하이버네이트, EclipseLink, DataNucleus





# JPA를 사용하는 이유

- SQL 중심적인 개발에서 객체 중심으로 개발
- 생산성
- 유지보수
- 패러다임의 불일치 해결
- 성능
- 데이터 접근 추상화와 벤더 독립성
- 표준

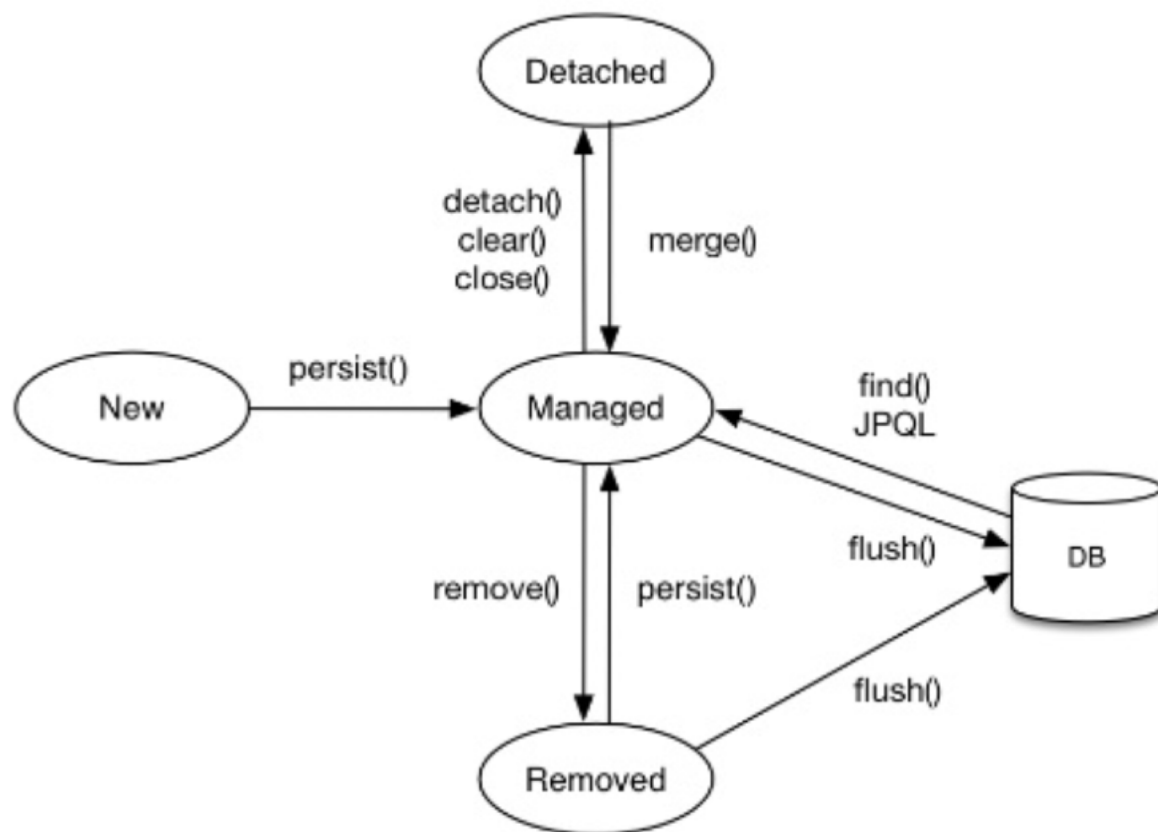
# 영속성 컨텍스트 (Persistence context)

- 엔티티를 영구 저장하는 환경
- 논리적인 개념에 가깝다. 영속성 컨텍스트는 엔티티 매니저를 생성할 때 하나 만들어진다. 그리고 엔티티 매니저는 영속성 컨텍스트에 접근할 수 있고, 영속성 컨텍스트를 관리할 수 있다.

# 엔티티 생명주기

- 비영속(new/transient) : 영속성 컨텍스트와 전혀 관계가 없는 상태
- 영속(managed) : 영속성 컨텍스트에 저장된 상태
- 준영속(detached) : 영속성 컨텍스트에 저장되었다가 분리된 상태
- 삭제(removed) : 삭제된 상태

# 엔티티의 생명주기



# 영속성 컨텍스트 특징

- 영속성 컨텍스트와 식별자 값. : 영속성 컨텍스트는 엔티티를 식별자 값으로 구분한다. 영속상태는 식별자 값이 반드시 있어야 한다.
- 영속성 컨텍스트와 데이터 베이스 저장 : JPA는 보통 트랜잭션이 커밋하는 순간 영속성 컨텍스트에 새로 저장된 엔티티를 데이터메이스에 반영한다. 이를 flush라고 한다.
- 영속성 컨텍스트가 엔티티를 관리할 때의 장점 : 1차 캐시, 동일성 보장, 트랜잭션을 지원하는 쓰기 지연, 변경감지, 지연 로딩

# 엔티티 조회

- 영속성 컨텍스트는 내부에 캐시를 가지고 있는데, 이를 1차 캐시라 한다. 영속상태의 엔티티는 모두 이곳에 저장된다.
- 1차 캐시의 키는 식별자 값이다. 이미 1차캐시에 있을 경우 db에서 조회하지 않고 캐시에서 조회하게 된다.

# 영속 엔티티의 동일성 보장

- 같은 id의 값을 가진 엔티티를 조회하면 == 를 이용했을 경우 참이 나온다. 이를 동일성이라한다. 이는 실제 인스턴스가 같다는 것을 의미한다.

# 트랜잭션 지연쓰기

- `em.persist(member1); em.persist(member2);` 와 같이 여러번 영속성을 부여한 다음 트랜잭션을 커밋할 때 한꺼번에 sql이 실행된다. 이를 트랜잭션을 지원하는 쓰기 지연이라고 말한다.



# 엔티티 수정

- 엔티티가 수정된 것을 감지하여 db내용을 수정하는 것을 변경감지라한다.

# 준영속

- 영속 상태의 엔티티를 준영속 상태로 만드는 방법
  - `em.detach(entity)` : 특정 엔티티를 준영속 상태로 전환한다. 1차 캐시부터 쓰기 지연 sql저장소까지 해당 엔티티를 관리하기 위한 모든 정보가 제거된다.
  - `em.clear()` : 영속성 컨텍스트를 초기화 한다.
  - `em.close()` : 영속성 컨텍스트를 종료한다.
- 준영속 상태의 특징
  - 거의 비영속 상태에 가깝다.
  - 식별자 값을 가지고 있다.
  - 지연 로딩을 할 수 없다.

# 병합

- 준영속 상태의 엔티티를 다시 영속상태로 변경할 경우 merge()를 사용한다.

ex: `Member member = em.merge(member);`

# 생산성 - JPA와 CRUD

- `jpa.persist(member);` //저장
- `Member member = jpa.find(memberId);` // 조회
- `member.setName("변경할 이름");` // 수정
- `jpa.remove(member);` // 삭제

# 유지보수 - 필드 변경시 모든 SQL 수정

```
public class Member {
```

```
    private String memberId;
```

```
    private String name;
```

```
    private String tel;
```

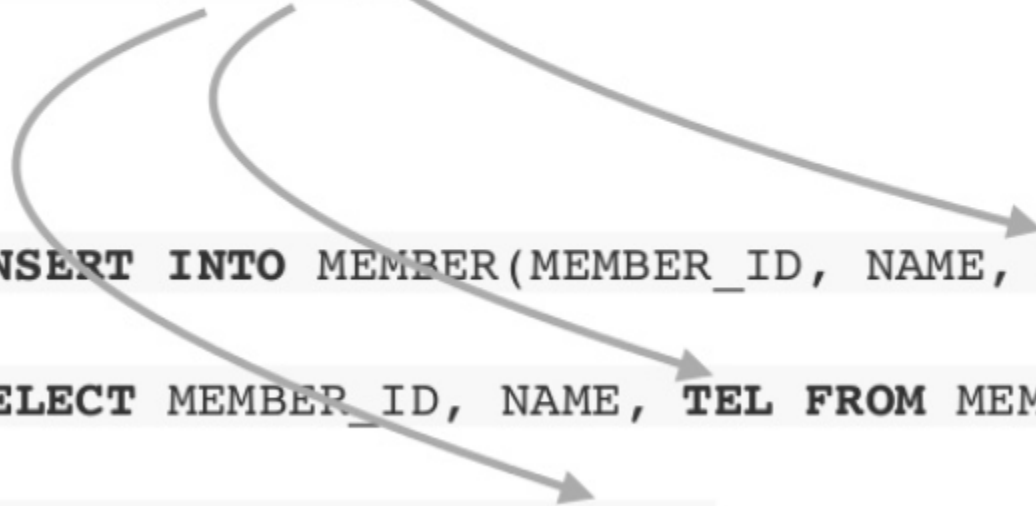
```
    ...
```

```
}
```

**INSERT INTO MEMBER(MEMBER\_ID, NAME, TEL) VALUES ...**

**SELECT MEMBER\_ID, NAME, TEL FROM MEMBER M**

**UPDATE MEMBER SET ... TEL=?**



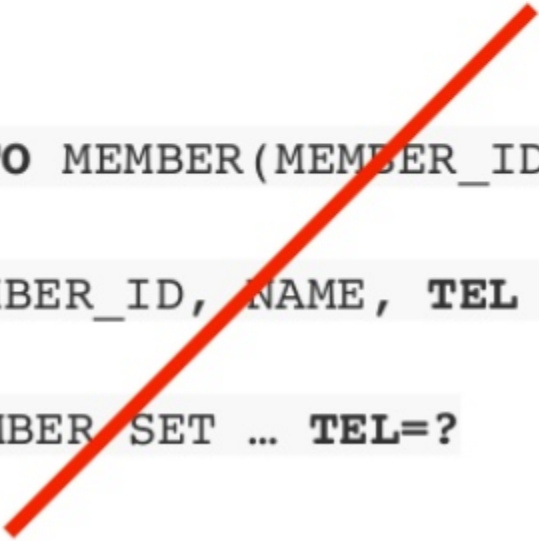
## 유지보수 - 필드만 추가하면 됨, SQL은 JPA가 알아서 처리

```
public class Member {  
  
    private String memberId;  
    private String name;  
    private String tel;  
    ...  
}
```

**INSERT INTO MEMBER(MEMBER\_ID, NAME, TEL) VALUES ...**

**SELECT MEMBER\_ID, NAME, TEL FROM MEMBER M**

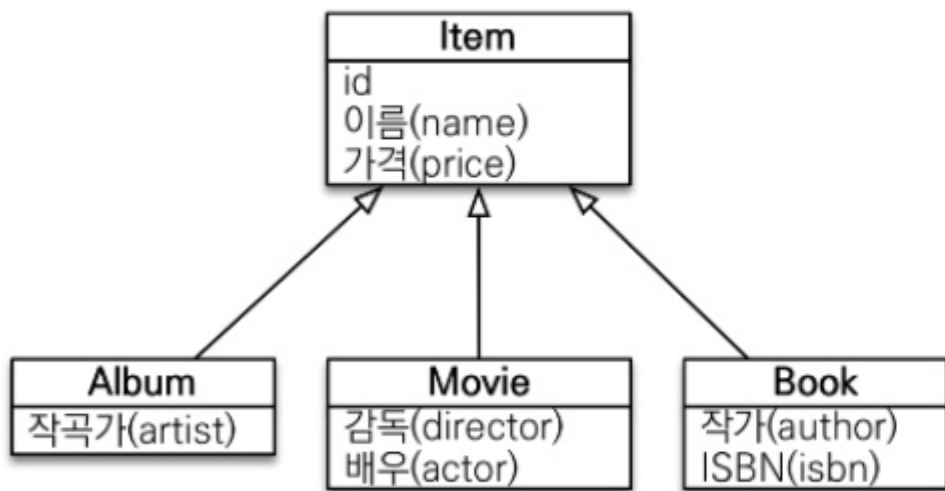
**UPDATE MEMBER SET ... TEL=?**



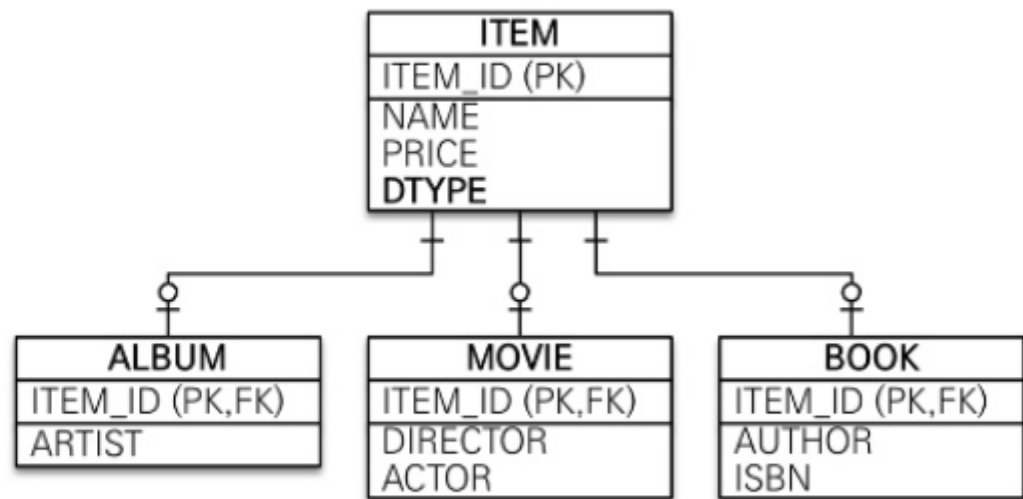
# JPA와 패러다임의 불일치 해결

- JPA와 상속
- JPA와 연관관계
- JPA와 객체 그래프 탐색
- JPA와 비교하기

# JPA와 상속



[객체 상속 관계]



[Table 슈퍼타입 서브타입 관계]



# JPA와 상속

```
jpa.persist(album);
```

개발자가 할 일

```
INSERT INTO ITEM ...  
INSERT INTO ALBUM ...
```

나머진 JPA가 처리

# JPA와 상속

```
Album album = jpa.find(Album.class, albumId);
```

개발자가 할 일

```
SELECT I.*, A.*  
FROM ITEM I  
JOIN ALBUM A ON I.ITEM_ID = A.ITEM_ID
```

나머진 JPA가 처리

# JPA와 연관관계, 객체 그래프 탐색

```
member.setTeam(team);  
jpa.persist(member);
```

연관관계 저장

```
Member member = jpa.find(Member.class, memberId);  
Team team = member.getTeam();
```

객체 그래프 탐색

# 신뢰할 수 있는 엔티티 계층

```
class MemberService {  
    ...  
    public void process() {  
        Member member = memberDAO.find(memberId);  
        member.getTeam(); //자유로운 객체 그래프 탐색  
        member.getOrder().getDelivery();  
    }  
}
```

# JPA와 비교하기

```
String memberId = "100";  
Member member1 = jpa.find(Member.class, memberId);  
Member member2 = jpa.find(Member.class, memberId);  
  
member1 == member2; //같다.
```

동일한 트랜잭션에서 조회한 엔티티는 같음을 보장

# Spring Data JPA

- JPA + Spring

# Query Method

- find...By : findBoardByTitle
- read...By : readBoardByTitle
- query...By : queryBoardByTitle
- get...By : getBoardByTitle
- count...By : countBoardByTitle

# Query Method – find by 특정 컬럼 처리

- Collection findBy + 속성이름(속성타입)



# Query Method - like 구문처리

- 단순 like : Like
- 키워드 + '%' : StartingWith
- '%' + 키워드 : EndingWith
- '%' + 키워드 + '%' : Cotaining

# Query Method - and 혹은 or 조건 처

- 2개 이상 속성 검색할 경우 And or Or를 사용  
ex) findByTitleContainingOrContentContaining

# Query Method - 부등호 처리

- >와 < 부등호 처리는 GreaterThan 과 LessThan을 이용
  - public Collection<Board>  
findByTitleContainingAndBnoGreaterThan(String keyword, Long num);

# Query Method - order by 처리

- OrderBy + 속성 + Asc or Desc를 이용

```
public Collection<Board> findByBnoGreaterThanOrderByBnoDesc(Long  
bno);
```

# Query Method – 페이징 처리와 정렬

- 모든 쿼리 메소드의 마지막 파라미터로 페이지 처리를 할 수 있는 Pageable 인터페이스와 정렬을 처리하는 Sort 인터페이스를 사용할 수 있다.
- org.springframework.data.domain.Pageable 인터페이스 구현한 PageRequest 클래스.
- boot 2.0 에서 new PageRequest()는 deprecated 되어서 PageRequest.of() 를 사용.
- bno > 0 order by bno desc 조건  

```
public List<Board> findByBnoGreaterThanOrderByBnoDesc(Long bno, Pageable paging);
```
- 파라미터에 Pageable 적용되고, 리턴타입이 Collection<> 대신 Slice<>, Page<>, List<> 로 이용.

# Query Method - 페이징 생성자 설정

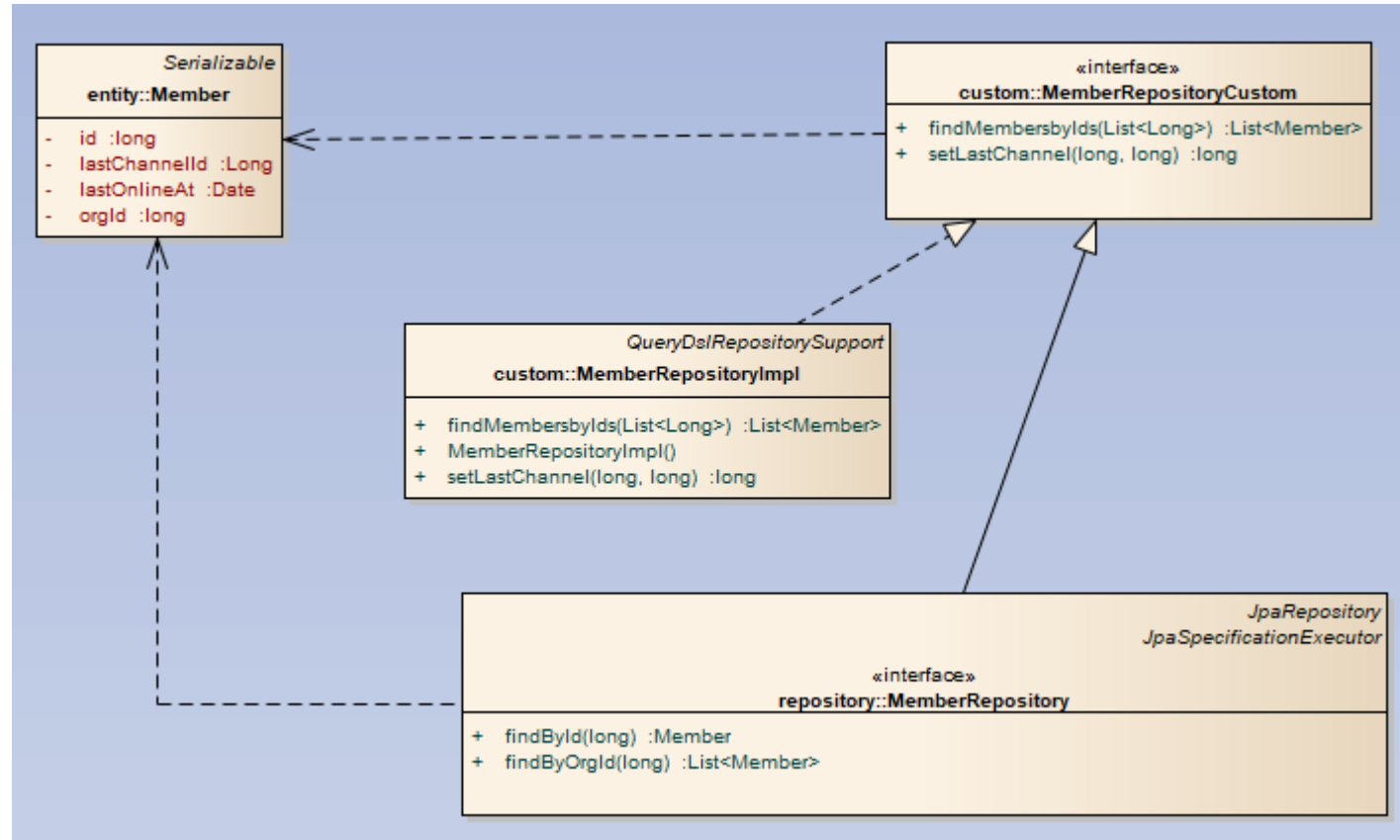
- `final Pageable paging = new PageRequest(0, 10);` // 페이지 번호 (0부터 시작), 페이지당 데이터수

## Query Method - 페이징 생성자에 정렬 객체 전달 가능

`PageRequest(int page, int size, SortDirection direction, String... props);` // 페이지 번호, 페이지당 데이터 수, 정렬 방향, 속성(칼럼) 들

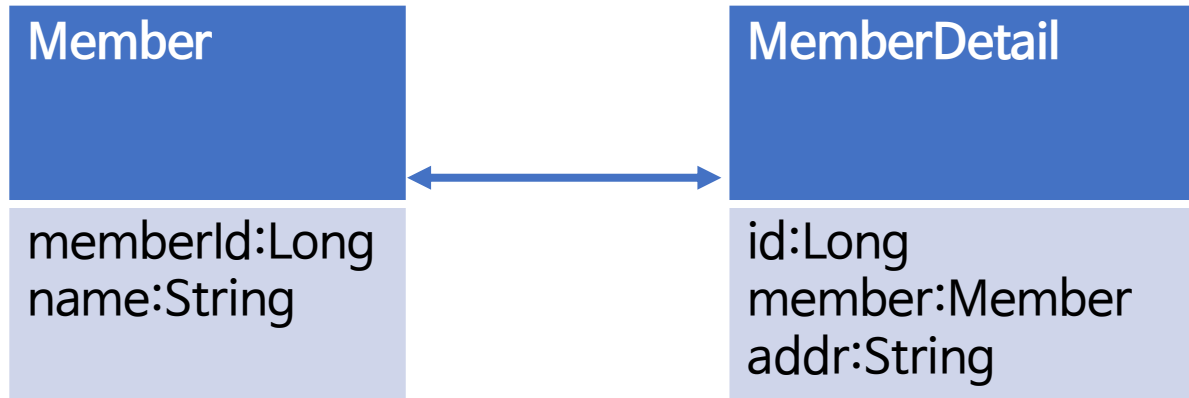
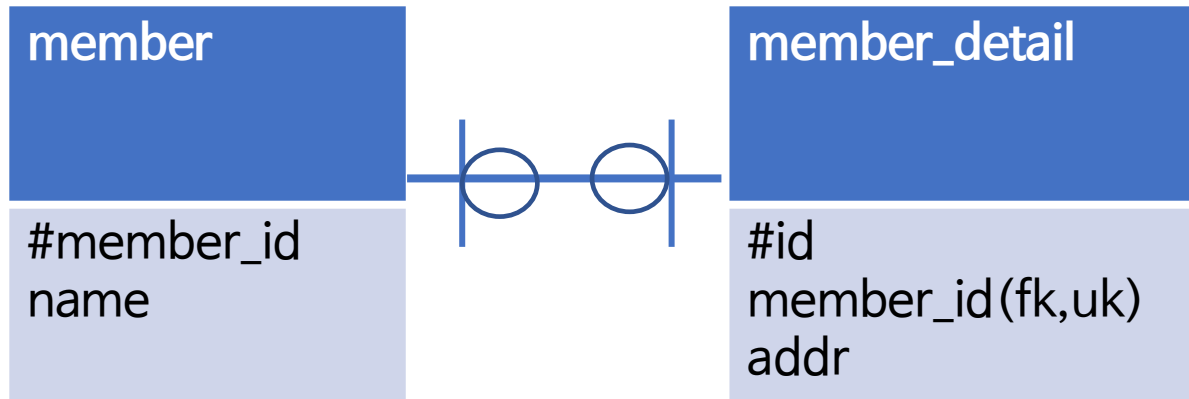
`new PageRequest(0, 10, Sort.Direction.ASC, "bno");`

# Spring Data + QueryDSL





# 엔티티 매핑(Entity Mapping) - 1 : 1, 단방향 , 대상테이블에 외래키



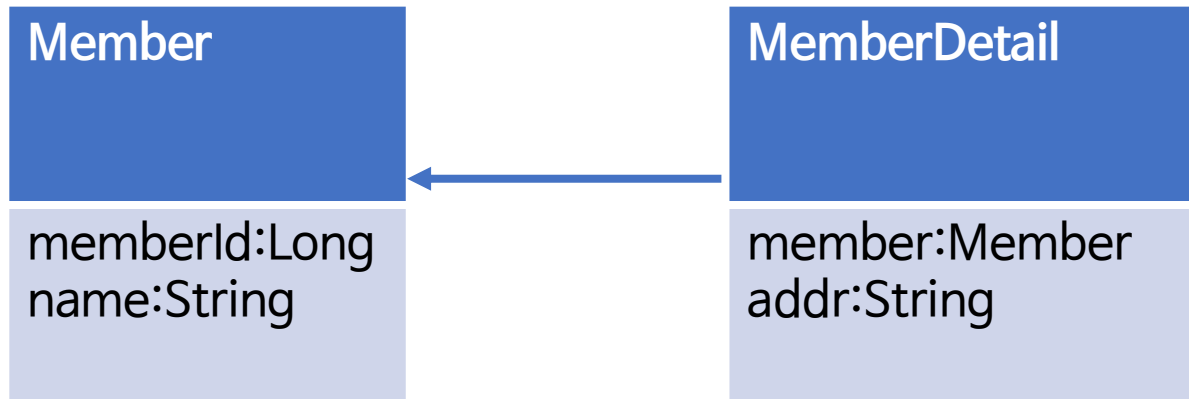
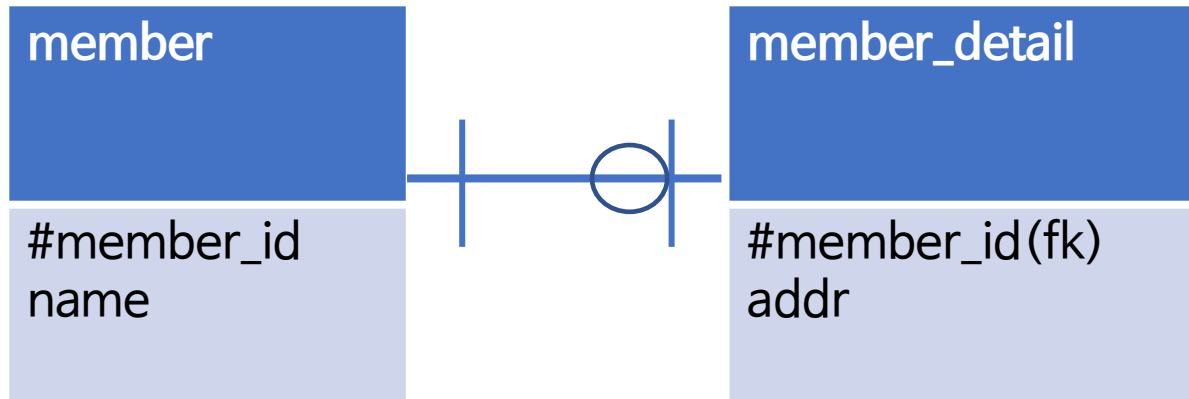
```
@Entity
class Member{
    @Id @GeneratedValue
    private Long memberId;
    private String name;
    @OneToOne(mappedBy="member")
    private MemberDetail memberDetail;
}

@Entity
class MemberDetail{
    @Id
    @GeneratedValue
    private Long id;
    private String addr;
    @OneToOne
    @JoinColumn(name="member_id")
    private Member member;
}
```

## 엔티티 매핑(Entity Mapping) - 1 : 1, 단방향 , 대상테이블에 외래키

- MemberDetail 테이블쪽에 Member테이블의 주키가 내려오는 경우로 JPA에서는 1:1관계에서 대상테이블에 외래키가 있는 경우는 지원하지 않는다. 1:N 단방향 매핑은 JPA2.0 부터 가능하다.  
1:1, 단방향 매핑에서 대상 테이블에 외래키가 있는 것을 지원하지 않으므로 양방향으로 만들고 Member쪽에 mappedBy를 사용하여 MemberDetail을 Owner로 만들어야 한다.

## 엔티티 매핑(Entity Mapping) - 1 : 1 식별관계, 단방향 , 대상테이블에 외래키



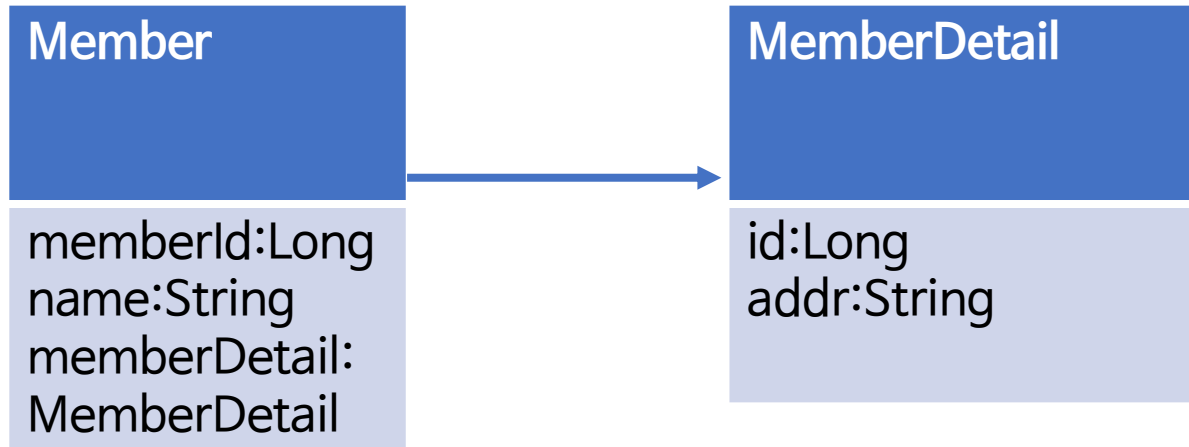
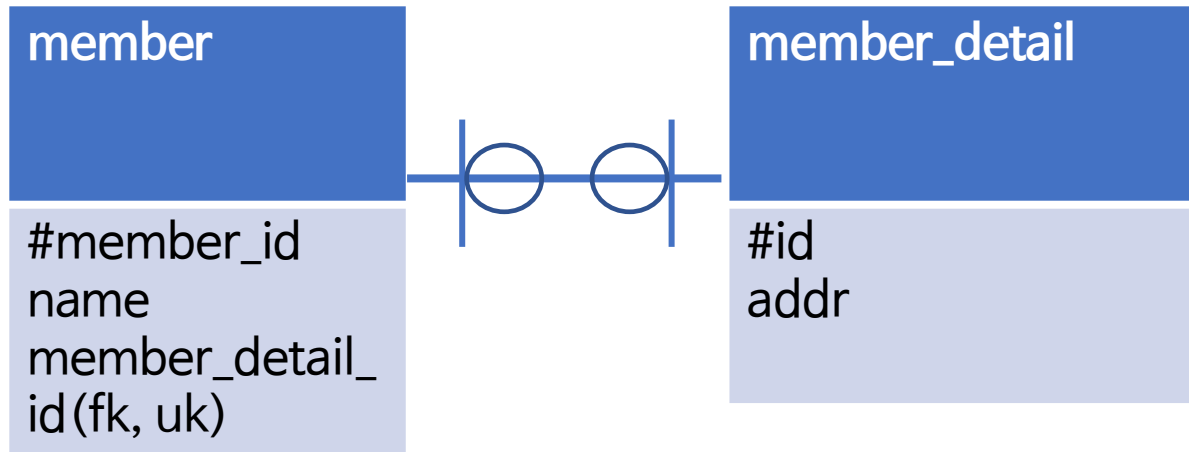
```
@Entity
class Member{
    @Id @GeneratedValue
    private Long memberId;
    private String name;
}

@Entity
class MemberDetail{
    @Id
    private Long id;
    private String addr;
    @OneToOne
    @MapsId
    @JoinColumn(name="member_id")
    private Member member;
}
```

## 엔티티 매핑(Entity Mapping) - 1 : 1, 단방향 , 대상테이블에 외래키

- 주테이블의 PK가 자식테이블에 식별자(PK) 형태로 외래키로 내려가는 경우이며, 부모테이블의 기본키를 자식 테이블에서도 기본키로 사용하는 경우이며 자식테이블의 기본키로는 부모테이블 에서 내려오는 외래키만을 사용하고, 식별자가 하나인 경우에는 @MapId를 사용한다.

## 엔티티 매핑(Entity Mapping) - 1 : 1, 단방향, 주 테이블에 외래키



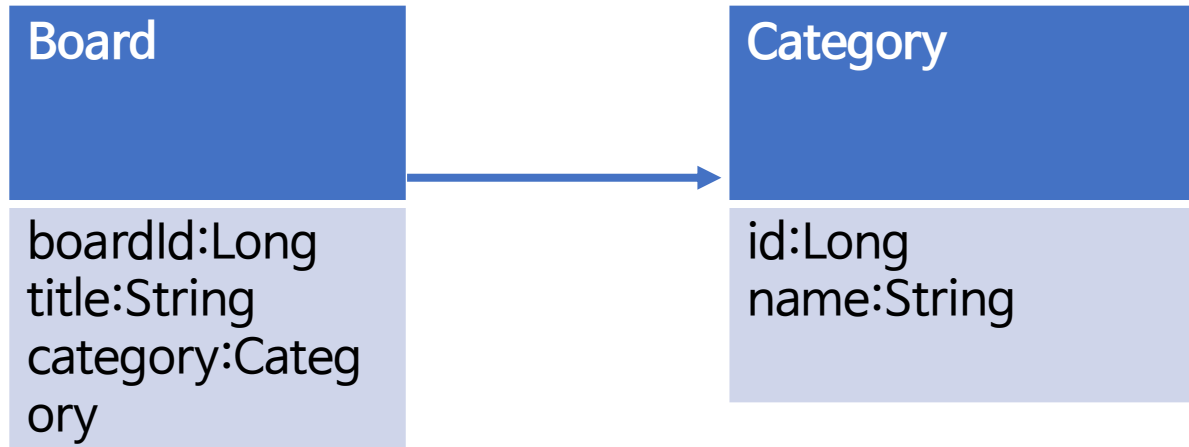
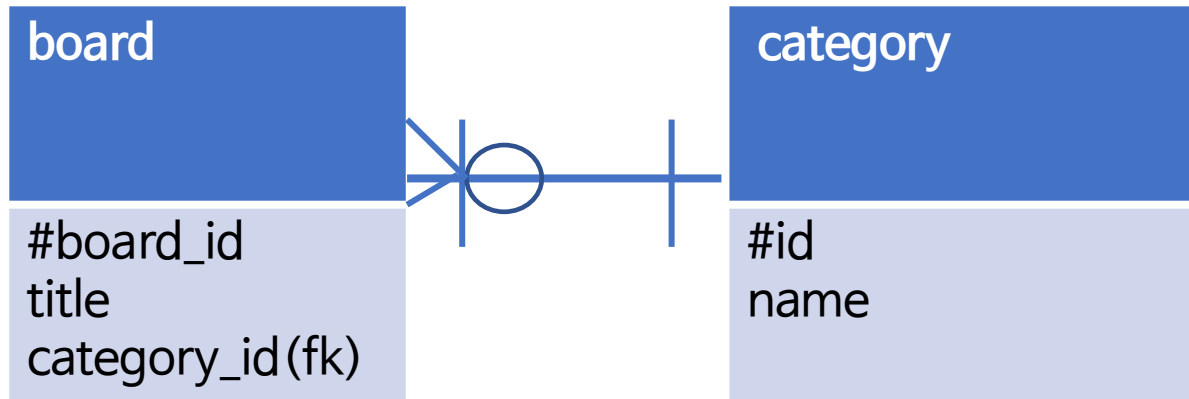
```
@Entity
class Member{
    @Id @GeneratedValue
    private Long memberId;
    private String name;
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "member_detail_id")
    private MemberDetail memberDetail ;
}

@Entity
class MemberDetail{
    @Id
    private Long id;
    private String addr;
}
```

## 엔티티 매핑(Entity Mapping) - 1 : 1, 단방향 , 주 테이블에 외래키

- 1:1 관계에서는 주 테이블이나 대상 테이블 모두 외래키를 가질 수 있다  
아래는 주테이블인 Member 테이블에 외래키가 있고 주테이블에서 대상 테이블을 참조하는 단방향 관계이다.

## 엔티티 매핑 (Entity Mapping) - N:1 단방향



```
@Entity
class Board{
    @Id @GeneratedValue
    private Long boardId;
    private String title;
    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category ;
}

@Entity
class Category{
    @Id
    private Long id;
    private String name;
}
```

# @OnDelete

@ManyToOne

@OnDelete(action=OnDeleteAction.CASCADE)

private Dept dept;

@OnDelete를 사용하여 ON DELETE CASCADE 제약조건을 생성 할 수 있다. JPA에서 생성하는 테이블 생성 코드(CREATE TABLE EMP...) 으에서 deptno 칼럼 정의하는 부분에 on delete cascade 옵션을 추가해 준다.



# @ForeignKey

```
@ManyToOne @JoinColumn(name="deptno")  
@ForeignKey(name="fk_dept_emp")  
private Dept dept;
```

- @ForeignKey를 사용하여 외래키를 지정할 수 있다.

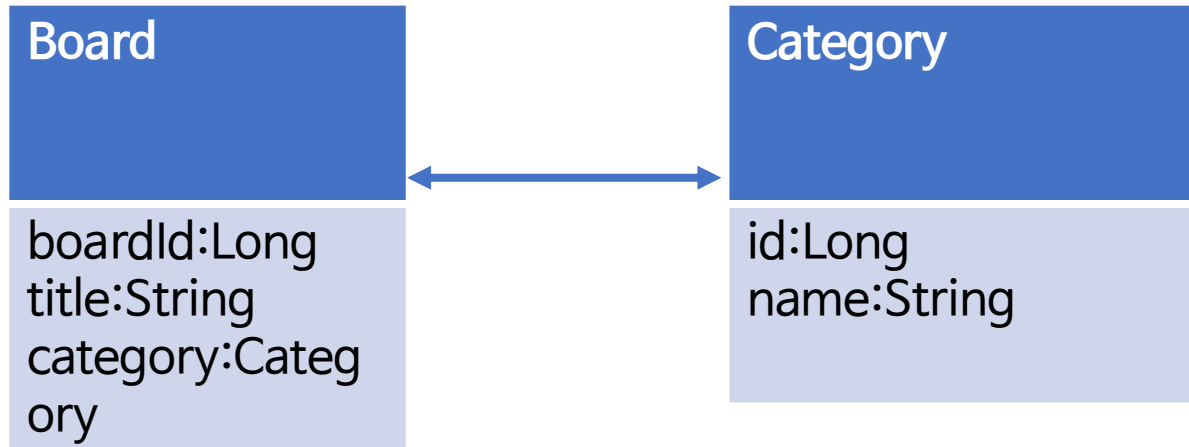
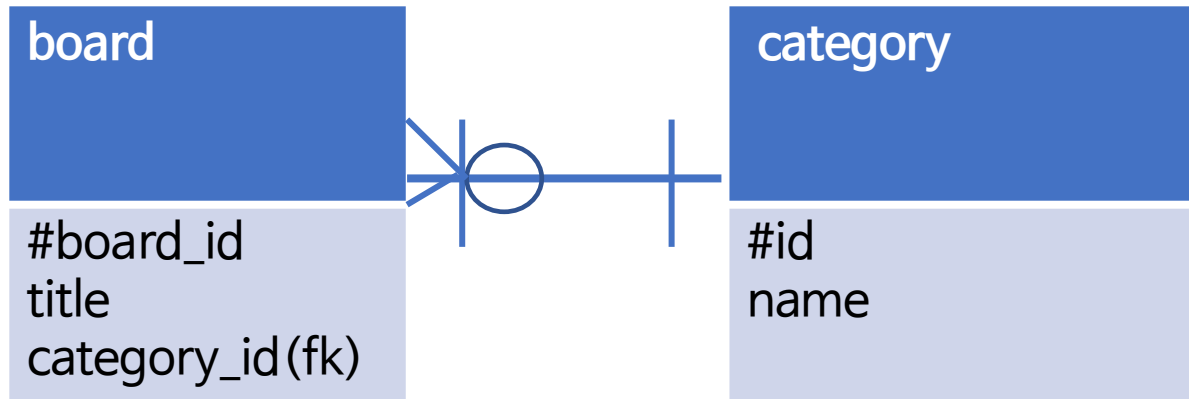
# @ManyToOne 속성 1/2

- optional : false인 경우 연관되는 데이터가 항상 있어야 한다.
- fetch : 데이터 패치 전략, 즉시로딩(FetchType.EAGER), 지연로딩(FetchType.LAZY) 값이 있으며 추천 방법은 모든 연관관계에 지연 로딩을 사용하고 상황에 따라 꼭 필요한 곳에만 즉시 로딩을 사용하는 것이 좋다. 지연 로딩은 어떤 객체의 필요한 속성정보가 있을 경우 사용하기 직전에 로딩한다는 의미이고 즉시로딩은 객체가 참조 될 때 그 즉시 로딩한다는 의미이다. 기본값 : @OneToMany : FetchType.LAZY
  - @ManyToOne : FetchType.EAGER
  - @ManyToMany : FetchType.LAZY
  - @OneToOne : FetchType.EAGER

# @ManyToOne 속성 2/2

- cascade : 영속성전이를 뜻하는데 어떤 객체를 영속성 객체로 만들 때 연관된 객체도 같이 영
- 속성 객체로 만든다. CascadeType은 여러 종류가 있다.
  - ALL : 부모의 영속성 변화가 자식에게 모두 전이 시킨다. (부모가 영속화되면 자식도 영속 화 되고, 부모가 저장되면 자식도 저장되고, 부모가 삭제되면 자식도 삭제된다.)
  - PERSIST: 부모가 영속화 될 때 자식도 영속화 된다.
  - MERGE : 트랜잭션이 종료되고 detach 상태에서 연관 엔티티를 추가하거나 변경된 이후에 부모 엔티티가 merge()를 수행하게 되면 변경사항이 적용된다. 연관 엔티티의 추가 및 수정 모두 반영된다.
  - REMOVE : 부모를 삭제할 때 연관된 자식 엔티티도 삭제된다.
  - REFRESH : 부모 엔티티가 REFRESH(DB에서 값을 다시 읽음)되면 연관도 자식 엔티티도 REFRESH 된다.
  - DETACH : 부모 엔티티가 detach를 수행하면 연관된 엔티티도 detach 상태가 되어 영속 성 컨텍스트를 빠져나와 변경 사항이 반영되지 않는다.

## 엔티티 매핑 (Entity Mapping) - N:1 양방향



```
@Entity
class Board{
    @Id @GeneratedValue
    private Long boardId;
    private String title;
    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category ;
}

@Entity
class Category{
    @Id
    private Long id;
    private String name;
    @OneToMany(mappedBy="board")
    //Set이외 List, Map등 사용가능 하다.
    private Set<Board> boards;
}
```

## 엔티티 매핑(Entity Mapping) - N:1 양방향

- 양방향 관계에서는 @OneToMany쪽에 mappedBy를 표시해서 연관관계의 주인인 아니라고 반드시 표시해야 하는데 이렇게 선언된 필드는 DB 테이블에 컬럼으로는 생성되지 않고 단지 ~~에 의해 매핑된다 라고 해석하면 된다. Category 엔티티의 boards 필드는 DB에 컬럼으로 생성되지 않고 단지 Board쪽의 category 필드에 의해 매핑된다는 의미이다.
- @OneToMany쪽에 mappedBy 속성으로 주인이 아니라고 표시해야 하며 자신의 객체참조를 가리키는 Owner 테이블(외래키 테이블, Board)의 필드명을 지정하면 된다. (Owner쪽은 @ManyToOne이 된다.) 양방향 관계에서 관계 연결처리를 하는 쪽, 외래키가 있는쪽(多)을 Owner라 한다. 다(多) 쪽인 @ManyToOne은 항상 연관관계의 주인(Owner)이 되므로 mappedBy를 설정할 필요 없다.

# 엔티티 매핑(Entity Mapping) - N:1 양방향

- 양방향 관계에서 @ManyToOne은 Owning Side 이며, @OneToMany는 Inverse Side 이다.
- 양방향 관계에서 Inverse Side의 mappedBy는 @OneToOne, @OneToMany 또는 @ManyToMany 에 사용되고 Owning Side의 InverseBy는 @OneToOne, @ManyToOne 또는 @ManyToMany에서 사용하는 속성이다.
- boards 필드는 지연, 즉시 읽기에 따라 로딩되는 시점은 다르지만 Category 엔티티가 DB에서 검색될 때 자동으로 활성화 된다.
- Board엔티티는 주인(OWNER)이며 주인 엔티티에서 일어난 변화만 DB에 반영되며 반대 쪽 엔티티인 Category 엔티티의 변화는 DB에 저장되지 않으며 검색시에만 참조된다.
- 대개 Owner쪽(Board)의 category필드는 외래키로 인덱스로 이용되어 검색속도가 향상 될 수 있다.

## @OneToMany

- N:1 양방향관계에서 one 쪽에서 many 쪽으로의 방향성 또는 1:N 단방향에서 many쪽의 데 이터를 참조하므로 Set, List, Map을 사용할 수 있다.
- List는 순서가 있고 중복을 허락하는 자료구조 이므로 자료들의 순서를 알 수 있고, 정렬이 가능하다. 자료들의 순서(인덱스)를 알기 위해서는 별도 인덱스 칼럼 (@OrderColumn)이 정의 되어야 한다.
- Map은 Key, Value 쌍으로 자료를 저장하는데 key설정을 위해 @MapKey를 사용하며, 실제 데 이블의 칼럼을 지정할 때는 @MapKeyColumn을 사용한다.

## @OneToMany 속성

- targetEntity : 연결을 맺는 상대 엔티티
- cascade : 연관 엔티티의 영속성전이 전략을 설정.
- mappedBy : 양방향 관계에서 주체가 되는 쪽(Many쪽, 외래키가 있는 쪽)의 자신을 가리키는 필드명을 기술하여 주인이 아님을 표시.
- orphanRemoval : 엔티티에서 삭제가 일어난 경우 연관관계에 있는 엔티티도 같이 삭제할지의 여부를 결정한다. Cascade는 JPA 레이어의 정의이고 이 속성은 DB레이어 에서 직접 처리한다. 기본은 false.



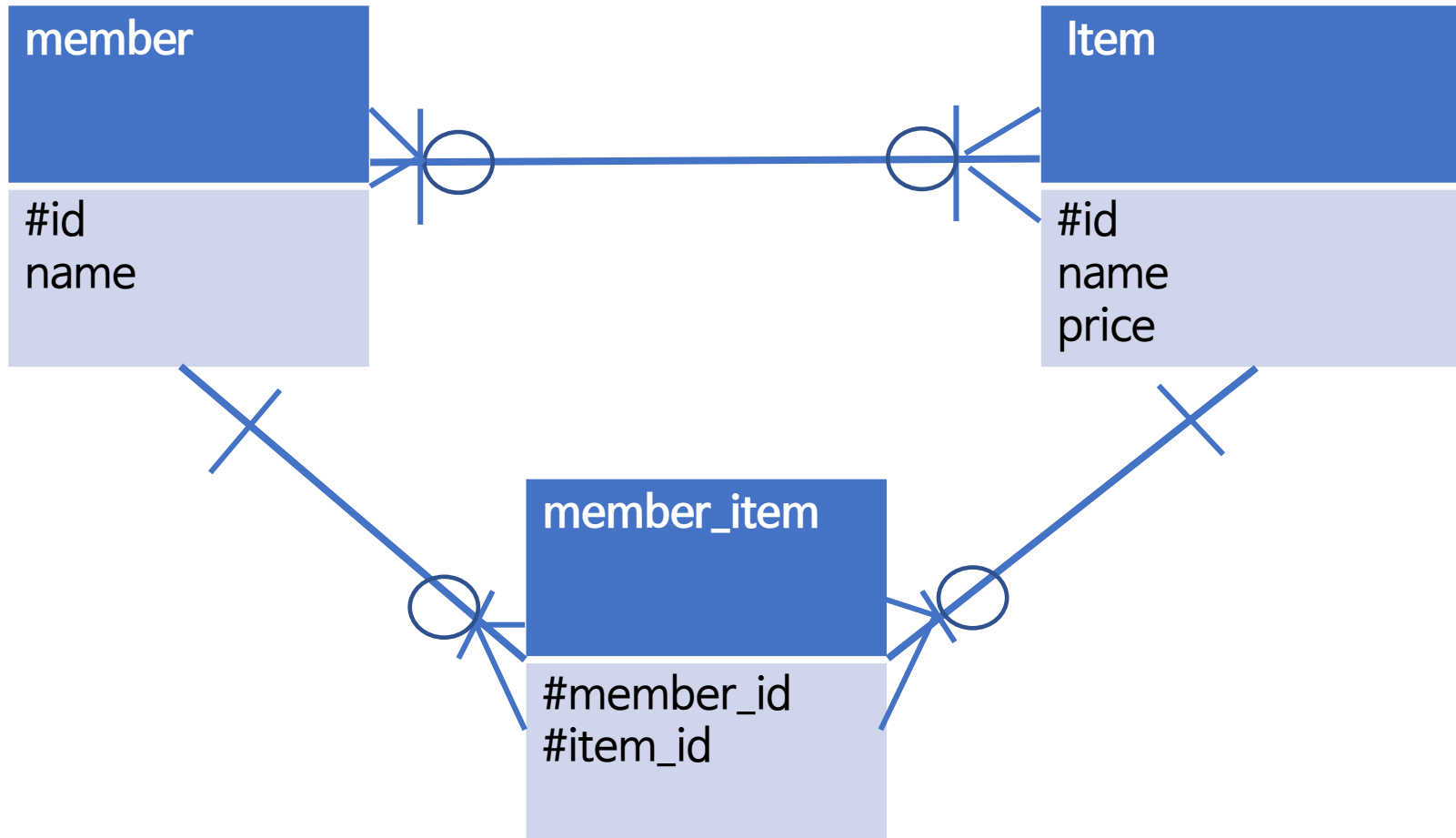
# mappedBy 1/2

- 테이블은 외래키 하나로 두 테이블의 연관관계를 관리하는데 엔티티를 단방향으로 매핑하면 참조를 하나만 사용하고, 양방향 관계로 설정하면 객체의 참조는 양쪽에서 하나씩 둘인데, 외래키는 하나이므로 두 엔티티 중 하나(OWNER)를 정해서 테이블의 외래키를 관리해야 한다. MANY쪽(Board)이 OWNER 이며 mappedBy는 @OneToMany쪽(Category)의 컬렉션 칼럼(boards)에 기술하여 OWNER가 아님을 정의한다. (테이블의 칼럼으로 만들어지지 않는다.)
- mappedBy는 @OneToOne, @OneToMany, @ManyToMany 어노테이션에서 사용할 수 있으며 mappedBy가 없으면 JPA에서 양방향 관계라는 것을 모르고 두 엔티티의 매핑 테이블을 생성한다.

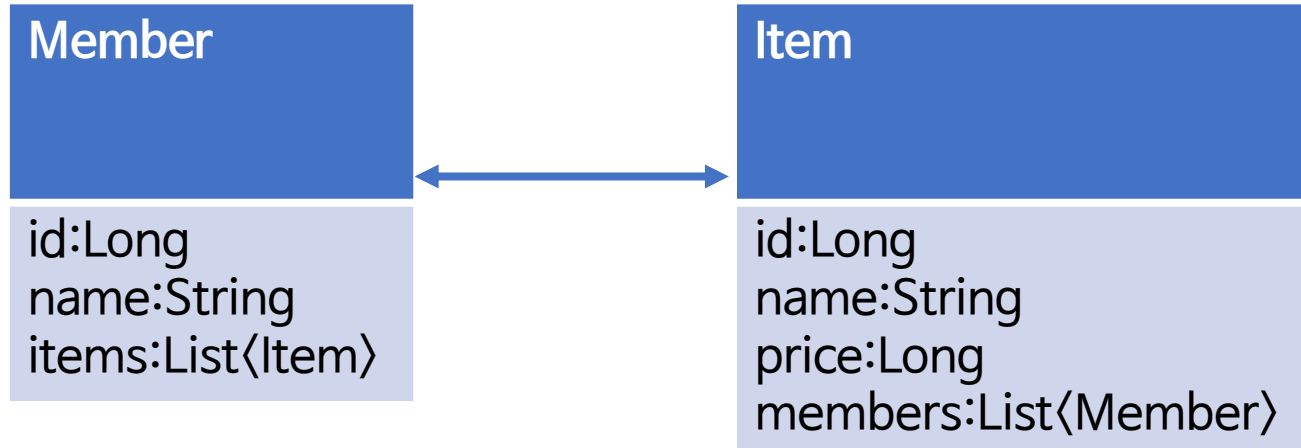
## mappedBy 2/2

- mappedBy의 값은 OWNER쪽 엔티티(Board)의 필드(category)에 대응된다. 즉 Category 엔티티의 boards 컬럼은 DB에 생성되는 컬럼이 아니고 EMP쪽의 dept 필드와 단지 매핑되는 필드임을 정의하는 것이다.
- ManyToOne 양방향 관계에서 Many측에는 mappedBy요소를 사용할 수 없다. (MANY 쪽이 OWNER), OneToOne 양방향 관계 OWNER는 반대쪽(INVERSE SIDE)에 대한 FK를 가지는 쪽이다.
- ManyToMany 양방향 관계는 양쪽 중 아무나 OWNER가 될 수 있다.

# M:N 연관관계



# M:N 연관관계



# M:N 연관관계 1/2

- 회원(Member), 아이템(Item)은 다 : 다 관계이다. 회원은 여러 아이템을 가질 수 있고, 하나의 아이템 역시 여러 회원에게 할당 될 수 있기 때문이다. 보통 관계형 DB에서는 다 : 다 관계는 1 : 다, 다 : 1 로 나누어서 풀게 된다. 그러나 객체에서는 보통 2개의 엔티티 클래스로 다 : 다 관계를 만드는데, 회원에서 아이템을 컬렉션에 넣어 접근 가능하고, 반대로 아이템에서도 회원들을 컬렉션에 넣어 접근하면 양쪽에서 접근이 가능하다.
- 만약 조인 테이블이 추가적인 별도의 속성(필드)를 가져야 한다면 조인 테이블에 대응되는 조인 클래스(엔티티 클래스)도 반드시 만들어야 한다. 아래 예제의 경우 조인 테이블의 추가적인 별도의 속성을 가지지 않으므로 엔티티 클래스는 2개로 작성 되었다.

# M:N 연관관계 2/2

- 다 : 다 매핑을 위해 @ManyToMany 어노테이션을 사용하며 RDB에 테이블로 생성될 때는 다 : 다 인 테이블을 연결시켜주기 위한 조인 테이블이 있어야 한다. 회원(member), 아이템(item) 관계는 다:다 관계인데 이 둘을 연결시켜주기 위한 조인 테이블이 있어야 한다는 것이다. 엔티티 클래스는 두개로 표현하지만 DB에 생성될 때 조인테이블이 생성되며 조인 테이블을 지정할 때는 @JoinTable 어노테이션을 사용한다.
- 다 : 다 양방향 관계에서는 한쪽을 Owning Side, 다른쪽을 Inverse Side라고 하는데 @ManyToMany 어노테이션에 mappedBy속성을 사용하여 Inverse Side 라고 표시를 한다. 즉 연관관계의 주인이 아님을 지정한다. (mappedBy가 없는 곳이 Owning Side)

# Member Entity

```
private List<Item> items;
```

```
.....
```

```
@ManyToMany(cascade = {CascadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH})
```

```
@JoinTable(name = "member_item",
```

```
        joinColumns = @JoinColumn(name = "member_id", referencedColumnName = "id") ,
```

```
        inverseJoinColumns = @JoinColumn(name = "item_id", referencedColumnName = "id") )
```

```
@OrderColumn(name="item_order") //member_item에 item_order칼럼 생성
```

```
public List<Hobby> getItems() {
```

```
    return items;
```

```
}
```

# Item Entity

```
private Set<Member> members;  
@ManyToMany(mappedBy = "items")  
public Set<Member> getMembers() {  
    return members;  
}
```



# 엔티티 설명

- 만약 @JoinTable 어노테이션으로 조인테이블을 지정하지 않으면 기본적인 이름은 주인테이블이름\_반대쪽 테이블을가리키는 주인테이블의필드명 (mappedby에서 기술한 값)이 된다. 또한 조인테이블에 생성되는 Primary Key의 순서도 주인 테이블의 Primary Key에 대한 외래키가 먼저 위치한다.
- 소스 오브젝트의 Primary Key에 대한 외래키는 JoinColumns 속성으로, 타겟 오브젝트의 Primary Key에 대한 외래키는 InverseJoinColumns 속성으로 지정한다.

# 식별자 자동생성 (@GeneratedValue) 1 / 3

- 식별자 자동생성은 @GeneratedValue 어노테이션으로 지정한다.
- 복합키 보다는 대행키(인공키, Artifitial Key) 사용을 권장한다.
- @GeneratedValue의 strategy 속성에 값을 지정해 여러 가지 식별자 자동 생성 전략을 선택할 수 있는데 AUTO, TABLE, SEQUENCE, IDENTITY 값으로 지정한다. 이 값들은 열거형인 GenerationType에 정의되어 있다.
- GenerationType.AUTO : 데이터베이스에 관계없이 식별자를 자동 생성 하라는 의미, DB가 변경되더라도 수정할 필요 없다.

# 식별자 자동생성 (@GeneratedValue) 2/3

- JPA 공급자에 따라 기본설정이 다름
  - Oracle을 사용할 경우 SEQUENCE가 기본
  - MS SQL Server의 경우 IDENTITY가 기본
- 오라클이라면 하이버네이트 글로벌 시퀀스(hibernate\_sequence)를 사용하므로 엔티티에서 두 군데 이상 AUTO로 사용한다면 시퀀스를 별도로 만들고 명시적으로 SequenceGenerator를 기술하여 사용하는 것이 좋다.
- GenerationType.TABLE : 가장 유연하고 이식성이 좋은 방법으로 키를 위한 테이블이 생성되어 있어야 하며, ID 생성 테이블은 두 개의 컬럼을 반드시 가져야 한다. 처음 컬럼은 특정 시퀀스를 식별하는데 사용되는 문자열 타입이고 모든 Generator를 위한 주키다. 두 번째 컬럼은 생성되고 있는 실제 ID 값인 시퀀스를 저장하는 숫자 타입으로 이 컬럼에 저장되는 값은 시퀀스에 할당되었던 마지막 값이 된다. 하나의 테이블에 여러 식별자 값을 저장할 수 있다.

# 식별자 자동생성 (@GeneratedValue) 3/3

- GenerationType.SEQUENCE : DB에서 생성해 놓은 시퀀스를 이용하는 방법으로 오라클, DB2, PostgreSQL, H2 데이터베이스 등에서 사용한다.
- GenerationType.IDENTITY : MySQL, MS-SQL 처럼 DB 자체적으로 식별자 칼럼에 자동증분 속성이 있는 경우에 사용한다. 엔티티의 주 키 컬럼을 위한 정의는 DB 스키마 정의의 일부분으로 정의되어야 한다.

# JPA JPQL

- JPQL (Java Persistence Query Language)은 지속적 엔티티를 저장하는 데 사용되는 메커니즘과 독립적으로 지속적 엔티티에서 검색을 정의하는 데 사용되는데 데이터베이스와 관련된 SQL을 사용하지 않고 오브젝트를 검색하는 언어이다.

# JPQL 기본 문법

select

from

[where]

[group by]

[having]

[orderby]

- jpql을 쉽게 도와주는 도구들 : Criteria 쿼리, QueryDSL

# JPA

참고 -

<http://projects.spring.io/spring-data/>

[http://www.querydsl.com/static/querydsl/4.0.1/reference/ko-KR/html\\_single/](http://www.querydsl.com/static/querydsl/4.0.1/reference/ko-KR/html_single/)

자바 ORM 표준 JPA 프로그래밍 - 저자 : 김영한