

# [JPA] JPA 소개

[JPA] JPA 소개

[JPA] 영속성 컨텍스트 #1

[JPA] 영속성 컨텍스트 #2

[JPA] 연관관계 매핑 기초 #1

(연관관계의 필요성, 단방향 연관관계)

[JPA] 연관관계 매핑 기초 #2

(양방향 연관관계와 연관관계의 주인)

# [JPA] JPA 소개

## 객체와 관계형 데이터베이스의 차이

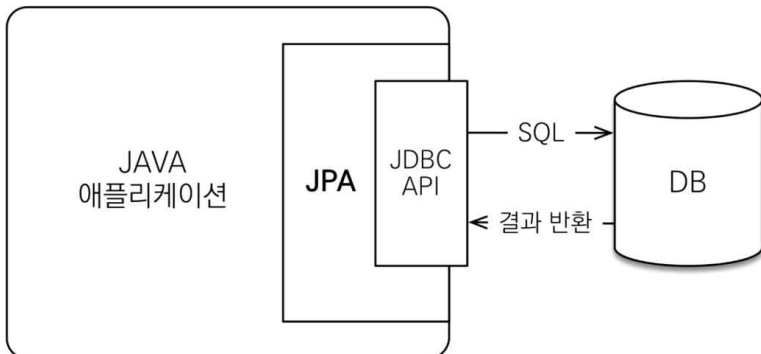
1. 상속 : 객체는 상속관계가 있지만, 관계형 데이터베이스는 상속 관계가 없다.
2. 연관관계
  - 객체는 reference(참조)를 가지고 있다. (예를들어, 연관된 객체를 getter로 가져올 수 있음)
  - 관계형 데이터베이스는 PK(Primary Key)나 FK(Foreign Key)로 join을 해서 필요한 데이터를 찾을 수 있다.
3. 데이터 타입
4. 데이터 식별 방법

● JPA : Java Persistence API / 자바 진영의 ORM 기술 표준

### ORM

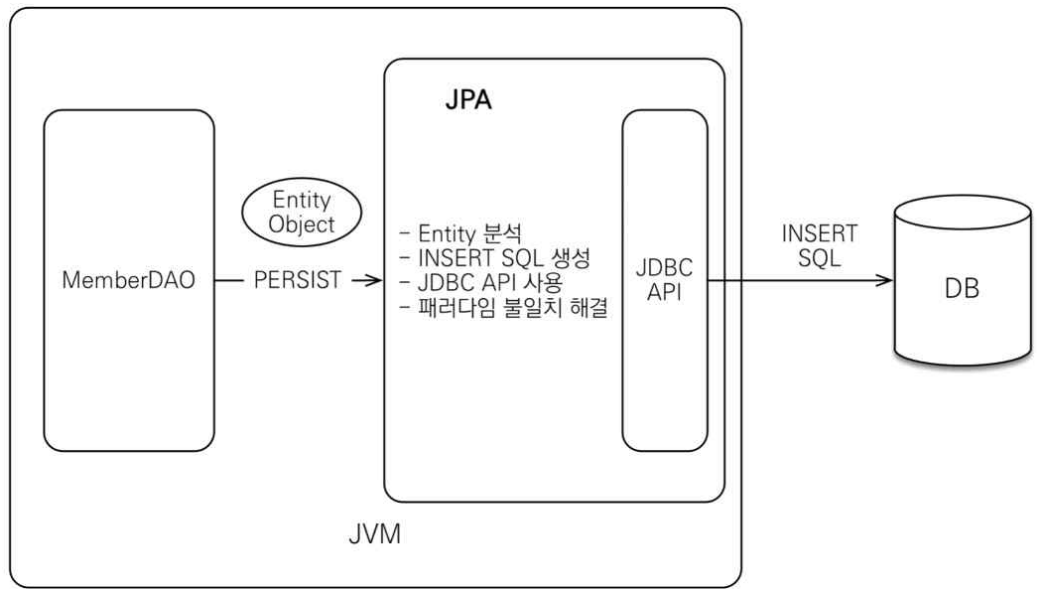
- Object-relational mapping(객체 관계 매핑)
- 객체는 객체대로 설계
- 관계형 데이터베이스는 관계형 데이터베이스대로 설계
- ORM 프레임워크가 중간에서 매핑
- 대중적인 언어에서 대부분 ORM 기술이 존재

● JPA는 애플리케이션과 JDBC 사이에서 동작



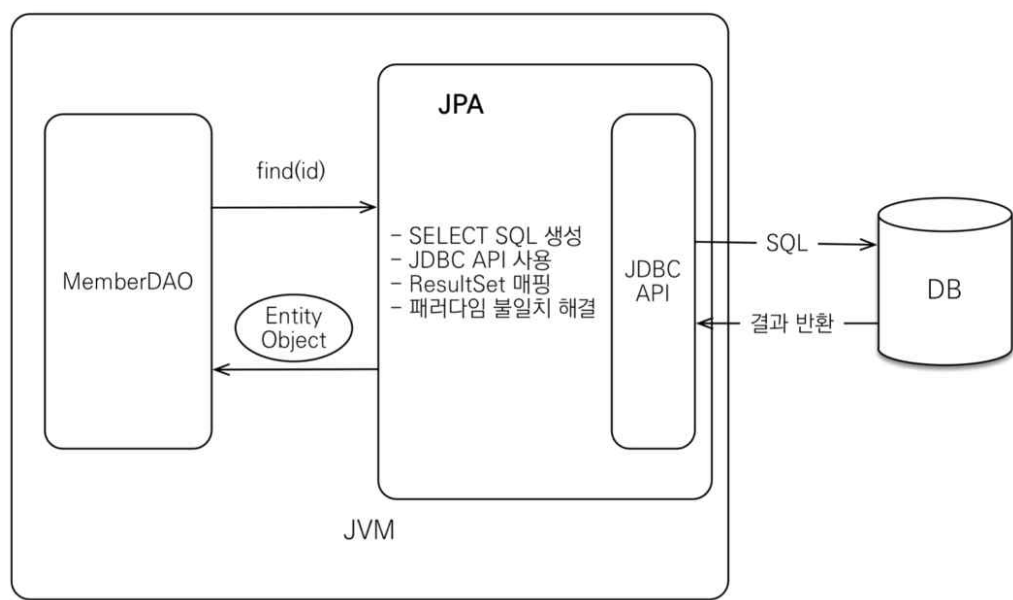
JAVA 애플리케이션에서 JPA에게 명령하면, JPA는 JDBC API를 사용해서 SQL을 만들어서 DB에 보낸다.

● JPA 동작(저장)



- JPA에게 Member객체를 넘기면 JPA가 객체를 분석
- INSERT query 생성 (JPA가 query를 만듬!)
- JPA가 JDBC API를 사용해서 INSERT query를 DB에 보냄
- 패러다임의 불일치 해결

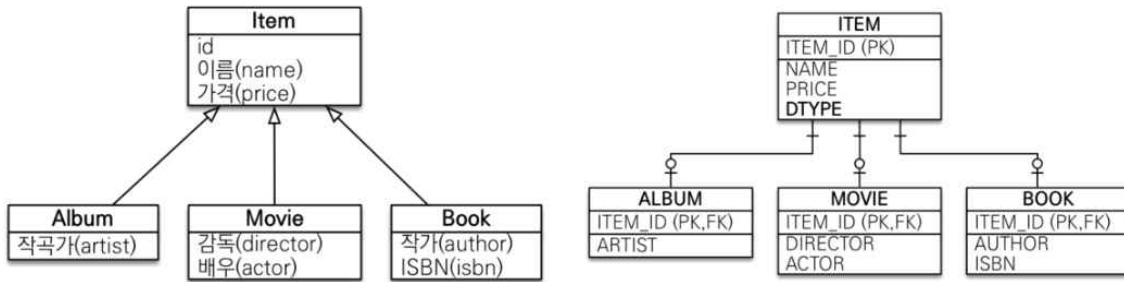
● JPA 동작(조회)



- JPA에게 PK값으로 find 요청
- JPA는 SELECT query 생성

- JDBC API를 통해 DB에 보내고 결과를 받음
- ResultSet 매핑
- 패러다임 불일치 해결

## ● JPA와 상속



[객체 상속 관계]

[Table 슈퍼타입 서브타입 관계]

### ① 저장

`jpa.persist(album);` 을 명령하면 JPA가 알아서 처리해준다.  
JPA는 `persist` 명령을 받고, ITEM table, ALBUM table 두 군데 모두 INSERT 쿼리를 날린다.

### ② 조회

`jpa.find(Album.class, albumId);`라는 명령을 하면 JPA가 알아서 처리해준다.  
JPA는 Item과 Album을 join해서 찾아옴.

## ● 지연 로딩과 즉시 로딩

- ① 지연 로딩: 객체가 실제 사용될 때 로딩
- ② 즉시 로딩: JOIN SQL로 한번에 연관된 객체까지 미리 조회

### 지연 로딩

```
Member member = memberDAO.find(memberId);  
Team team = member.getTeam();  
String teamName = team.getName();
```

```
SELECT * FROM MEMBER  
SELECT * FROM TEAM
```

### 즉시 로딩

```
Member member = memberDAO.find(memberId);  
Team team = member.getTeam();  
String teamName = team.getName();
```

```
SELECT M.*, T.*  
FROM MEMBER  
JOIN TEAM ...
```

지연 로딩은 member를 조회 했을 때, member만 가져온다.

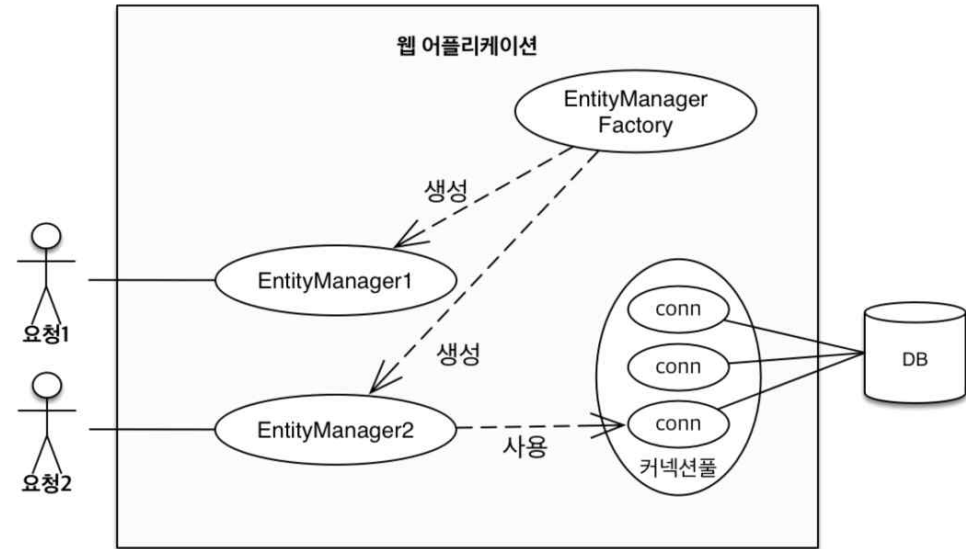
그리고 3번째 줄에서 team.getName을 했을 때, jpa가 db에 team에 대한 query를 날려서 가져온다. 즉,필요한 시점에 쿼리를 날려 값을 가져온다.

즉시 로딩은 member를 조회 했을 때, member와 연관된 객체까지 모두 가져온다.

따라서 member를 가져왔을 때 member만 쓴다고 하면 지연 로딩이 좋고,  
member를 가져왔을 때 member와 연관 객체를 같이 쓰는 경우엔 즉시 로딩이 좋다.

# [JPA] 영속성 컨텍스트 #1

## ● 먼저 알고 가기 : 엔티티 매니저 팩토리 와 엔티티 매니저



### <시나리오>

1. 새로운 고객의 요청이 올때마다 엔티티 매니저 팩토리는 엔티티 매니저를 생성함
2. 엔티티 매니저는 내부적으로 데이터베이스 커넥션을 사용해서 DB를 사용

### 엔티티 매니저 팩토리

- 엔티티 매니저를 만드는 공장
- 엔티티 매니저 팩토리는 생성하는 비용이 커서 한 개만 만들어 애플리케이션 전체에서 공유
- 여러 스레드가 동시에 접근해도 안전

### 엔티티 매니저

- 엔티티의 CRUD 등 엔티티와 관련된 모든 일을 처리
- 엔티티 매니저는 여러 스레드가 동시에 접근하면 동시성 문제가 발생하므로 스레드 간에 절대 공유하면 안됨

## ● 영속성 컨텍스트란!?

```
EntityManger.persist(member);
```

이 코드는 member 엔티티를 저장한다고 했었다. 하지만 정확하게 얘기하면 데이터베이스에 저장하는게 아니라 엔티티 매니저를 사용해서 회원 엔티티를 영속성 컨텍스트에 저장하는 코드다.

영속성 컨텍스트는 엔티티 매니저를 생성할 때 하나만 만들어진다. 엔티티 매니저를 통해 영속성 컨텍스트에 접근할 수 있고, 영속성 컨텍스트를 관리할 수 있다.

## ● 엔티티의 생명주기

### ① 엔티티의 4가지 상태

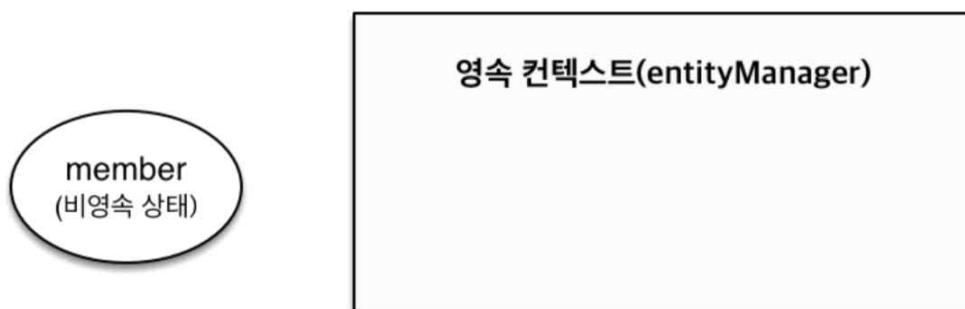
- 비영속(new/transient): 영속성 컨텍스트와 전혀 관계가 없는 새로운 상태
- 영속(managed): 영속성 컨텍스트에 관리되는 상태
- 준영속(detached): 영속성 컨텍스트에 저장되었다가 분리된 상태
- 삭제(removed): 삭제된 상태

이제 각 상태에 대해 자세히 알아보자.

### ▶ 비영속(new/transient)

엔티티 객체가 생성된 순수 객체 상태

아직 영속성 컨텍스트나 데이터베이스와는 전혀 관계가 없는 상태



## ▶ 영속(managed)

엔티티 매니저를 통해 엔티티를 영속성 컨텍스트에 저장하면, 영속성 컨텍스트가 엔티티를 관리하므로 영속 상태가 된다.



```
EntityManager.persist(member)
```

위 코드를 수행하면 member 엔티티는 영속성 컨텍스트에 의해 관리되는 영속 상태가 됨  
위 코드가 수행된다고 데이터베이스에 저장되는 것은 아님. (트랜잭션이 commit하는 시점에 영속성 컨텍스트에 있는 엔티티들에 대한 쿼리가 날라간다)

## ▶ 준영속(detached)

영속성 컨텍스트가 관리하던 영속 상태의 엔티티를 영속성 컨텍스트가 관리하지 않으면 준영속 상태가 됨

엔티티를 준영속 상태로 만들려면?

- EntityManager.detach(entity) : 엔티티를 준영속 상태로 만듦
- EntityManager.close() : 영속성 컨텍스트를 닫음
- EntityManager.clear() : 영속성 컨텍스트를 초기화

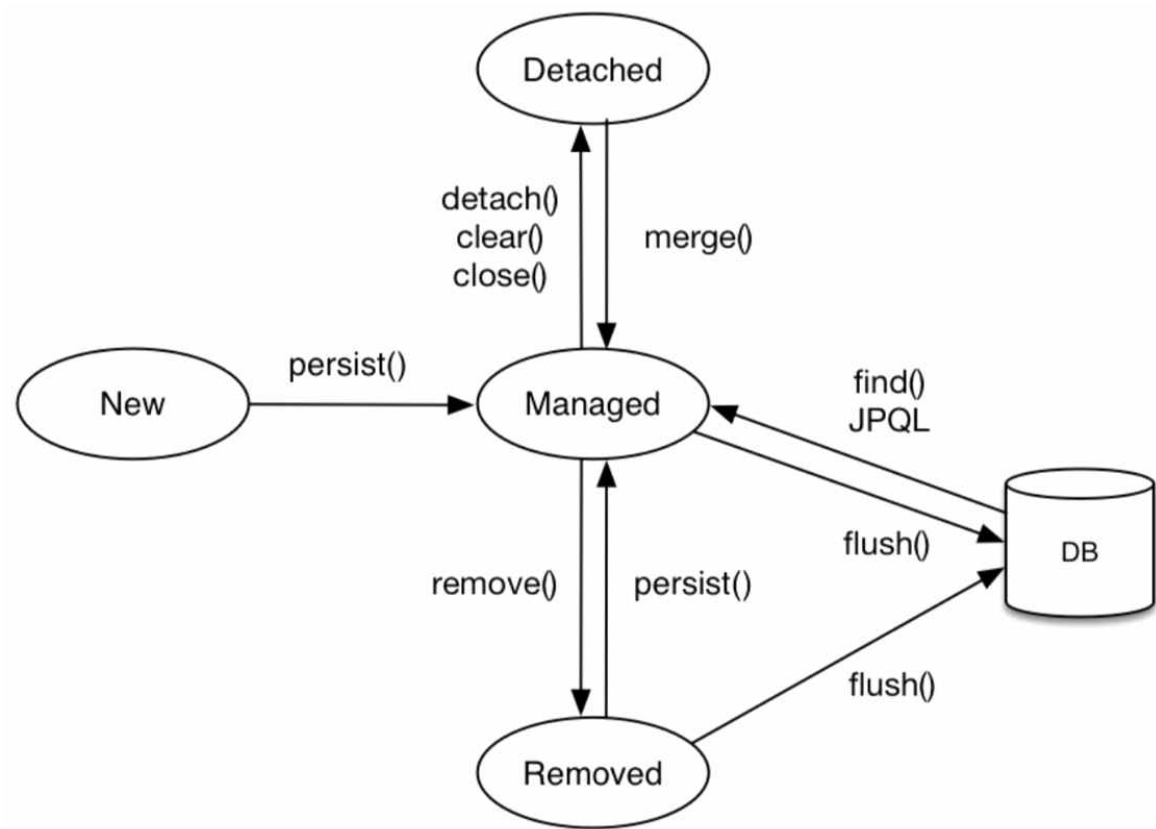


## ▶ 삭제

엔티티를 영속성 컨텍스트와 데이터베이스에서 삭제

```
EntityManager.remove(entity)
```

## 그림으로 보는 엔티티 생명주기



# [JPA] 영속성 컨텍스트 #2

## 1. 영속성 컨텍스트의 특징

- 영속성 컨텍스트와 식별자 값

영속성 컨텍스트는 엔티티를 식별자 값(@Id로 테이블의 기본 키와 매핑한 값)으로 구분한다. 따라서 영속 상태는 식별자 값이 반드시 있어야 한다!

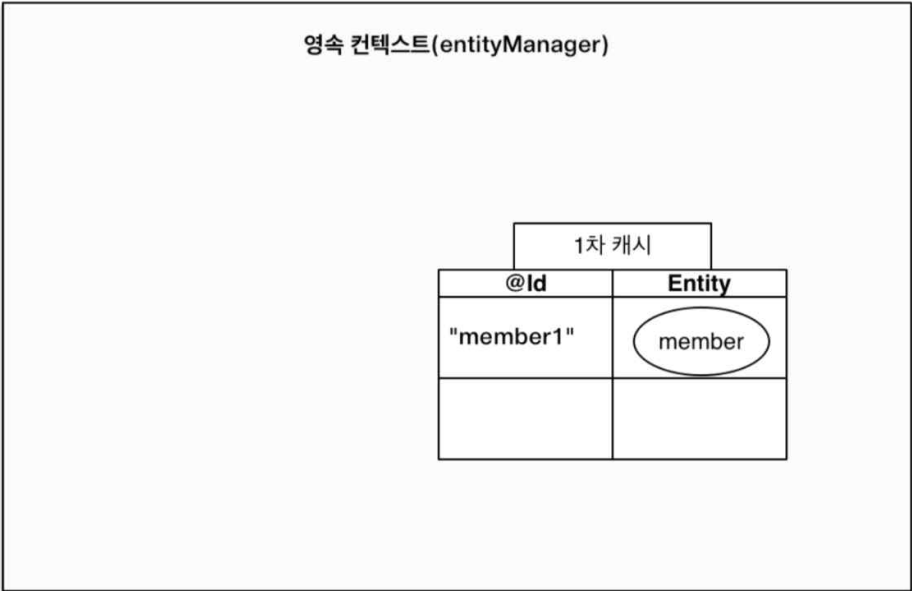
- 영속성 컨텍스트와 데이터베이스 저장

JPA는 보통 트랜잭션을 커밋하는 순간 영속성 컨텍스트에 새로 저장된 엔티티를 데이터베이스에 반영한다. 이를 플러시(flush) 라고 한다.

- 영속성 컨텍스트가 엔티티를 관리하면 얻게되는 장점

- 1차 캐시
- 동일성(identity) 보장
- 트랜잭션을 지원하는 쓰기 지연(transactional write-behind)
- 변경 감지(Dirty Checking)
- 지연 로딩(Lazy Loading)

### ① 엔티티 조회, 1차 캐시



### 1차 캐시란?

영속성 컨텍스트는 내부에 캐시를 가지고 있는데 이것을 **1차 캐시**라 한다. 영속 상태의 엔티티는 모두 이곳에 저장된다. 쉽게 말해 영속성 컨텍스트 내부에 Map이 하나 있는데 (1차 캐시), 키는 @Id로 매핑한 식별자고 값은 엔티티 인스턴스다.

★ service → ContextService.Class

```
@Service
@Transactional
public class ContextService {
    @Autowired
    EntityManager em;

    public Member memberInsert(){
        Member member = new Member();
        member.setMemberId("member_1");
        member.setName("김형민");
        em.persist(member);

        Member m = em.find(Member.class, "member_1");
        return m;
    }
}
```

☆ test → service → ContextTest.Class

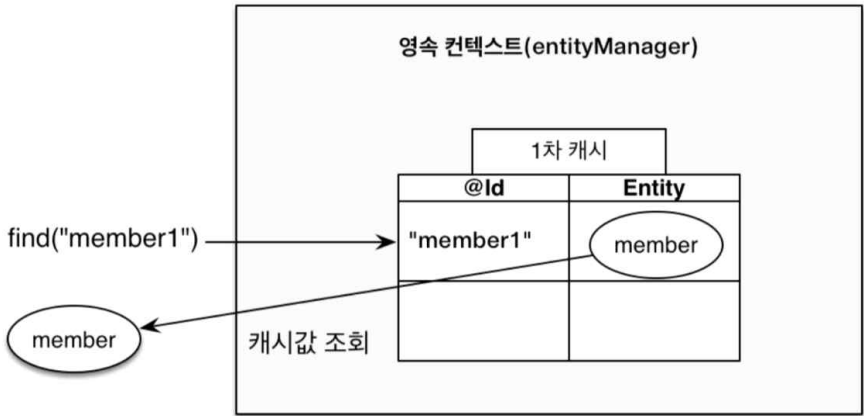
```
@SpringBootTest
public class ContextTest {
    @Autowired
    EntityManager em;

    @Autowired
    ContextService contextService;
}
```

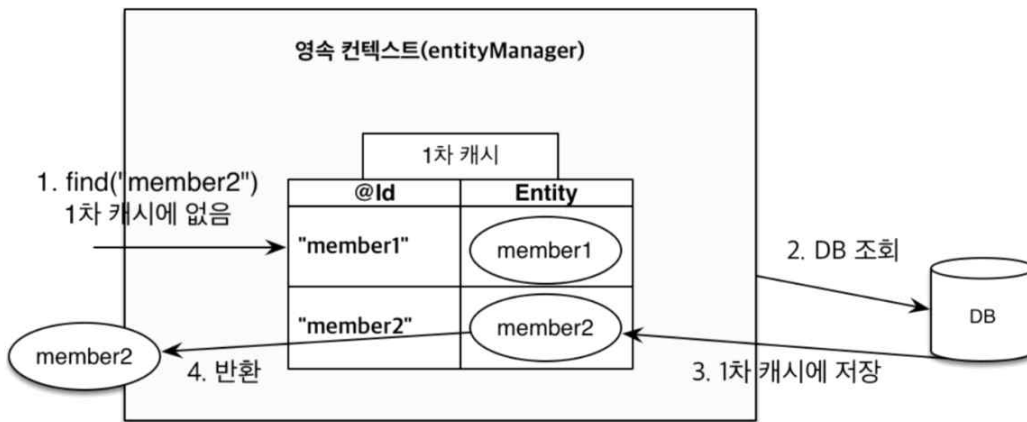
```
@Test
@DisplayName("1차 캐시 테스트")
void firstCash(){
    Member m = contextService.memberInsert();
    System.out.println("=====" + m);
}
}
```

- 엔티티를 조회하는 find 메서드가 실행되면
1. 1차 캐시에서 식별자 값으로 엔티티를 찾는다.
  2. 만약 찾는 엔티티가 1차 캐시에 있으면 데이터베이스를 조회하지 않고 메모리에 있는 1차 캐시에서 엔티티를 조회
  3. 1차 캐시에 찾는 엔티티가 없으면 데이터베이스에서 조회

아래 그림은 위 코드를 실행했을 때 일어나는 상황



em.find()를 호출했는데 엔티티가 1차 캐시에 없으면 엔티티 매니저는 데이터베이스를 조회해서 엔티티를 생성한다. 그리고 1차 캐시에 저장한 후에 영속 상태의 엔티티를 반환한다.



## ② 영속 엔티티의 동일성 보장

☆ test → service → ContextTest.Class

```
@Test
@DisplayName("데이터 영속성 보장 테스트")
void persistContextText(){
    Member a = em.find(Member.class, "member1");
    Member b = em.find(Member.class, "member1");
    System.out.println(a.equals(b));
}
```

현재 member1이 영속성 컨텍스트에 존재하는 상황에서  
위 코드의 a.equals(b) 가 참일까 거짓일까?  
정답은 참이다.

member1을 두 번 find해서 찾은 객체인데 왜 같을까?  
영속성 컨텍스트는 1차 캐시에 있는 같은 엔티티 인스턴스를 반환하기 때문에 참이다!  
따라서 영속성 컨텍스트는 성능상 이점과 엔티티의 동일성을 보장한다

만약 영속성 컨텍스트에 member1이 없는데 위 코드를 실행한다면 결과가 같을까?  
그렇다.

처음 member1에 대한 find 요청 시 엔티티 매니저는 데이터베이스에서 member1을 조회해서 1차 캐시에 저장하고 반환한다. 두번째 find 요청시에는 영속성 컨텍스트가 1차 캐시에 있는 member1 엔티티 인스턴스를 반환하므로 같은 결과가 된다.

### ③ 트랜잭션을 지원하는 쓰기 지연

★ service → ContextService.Class

```
@Transactional
public void transactionTest(){
    Member member1 = new Member();
    member1.setMemberId("member2");
    member1.setName("Hong");

    Member member2 = new Member();
    member2.setMemberId("member3");
    member2.setName("Kim");

    em.persist(member1);
    em.persist(member2);

    em.flush();
}
```

☆ test → service → ContextTest.Class

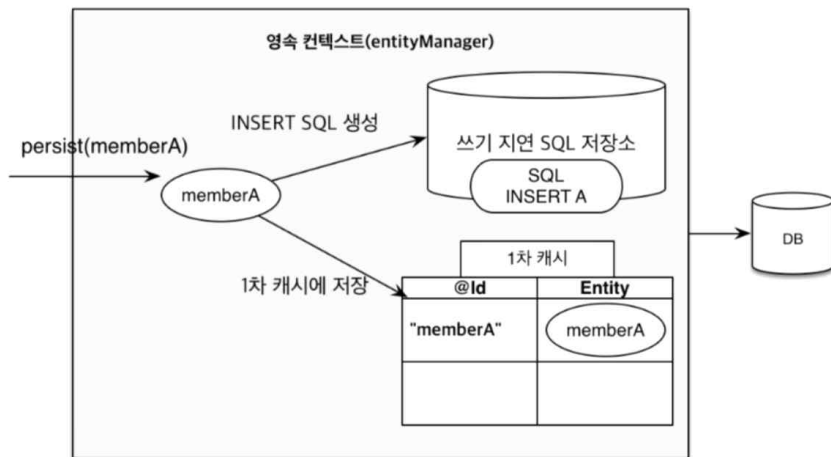
```
@Test
@DisplayName("Transaction Commit 테스트")
void transactionCommitTest(){
    contextService.transactionTest();
}
```

위는 엔티티 매니저를 사용해서 엔티티를 영속성 컨텍스트에 등록하는 코드다.

엔티티 매니저는 트랜잭션을 커밋하기 직전까지 데이터베이스에 엔티티를 저장하지 않고 내부 쿼리 저장소에 INSERT SQL을 모아둔다. 그리고 트랜잭션을 커밋할 때 모아둔 쿼리를 데이터베이스에 보내서 저장시킨다.

이를 트랜잭션을 지원하는 쓰기 지연이라 한다.

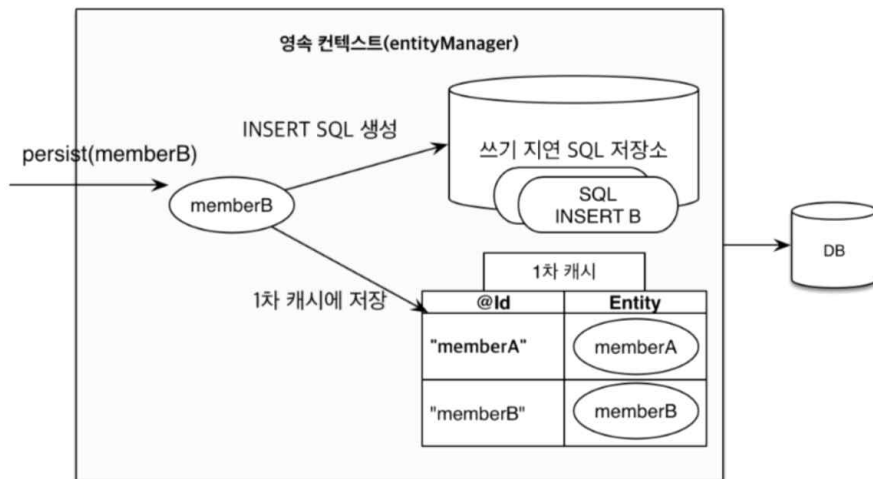
코드가 실행되는 과정을 그림으로 알아보자!



위는 `em.persist(memberA)`가 실행되면 일어나는 일에 대한 그림이다.

`em.persist(memberA)`가 실행되면, 영속성 컨텍스트는 1차 캐시에 `memberA`에 대한 엔티티를 저장하면서 동시에 JPA가 이 entity를 분석해서 쓰기 지연 SQL 저장소에 INSERT 쿼리를 저장한다.

(DB에 아직 INSERT 쿼리를 보내지 않는다!!)



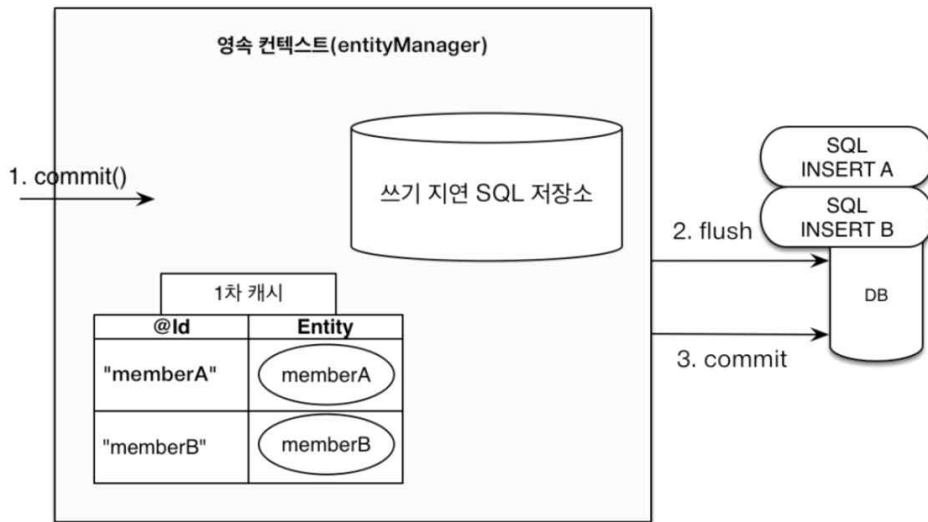
위는 `em.persist(memberB)`가 실행되면 일어나는 일에 대한 그림이다.

`em.persist(memberB)`가 실행되면, `memberA`와 같은 작업이 실행된다.

(DB에 아직 INSERT 쿼리를 보내지 않는다!!)

그럼 쓰기 지연 SQL 저장소에 저장된 쿼리들이 DB에 날라가는 시점은 언제인가?

그 시점은 코드에서 보이는 트랜잭션을 커밋하는 순간이다.



트랜잭션을 commit하면, 엔티티 매니저는 영속성 컨텍스트를 플러시(flush) 한다. 플러시는 영속성 컨텍스트의 변경 내용을 데이터베이스에 동기화하는 작업인데 이때 등록, 수정, 삭제한 엔티티를 데이터베이스에 반영한다.

#### ④ 변경 감지(Dirty Checking)

SQL을 사용하면 수정 쿼리를 직접 작성해야 한다. 그런데 프로젝트가 점점 커지고 요구 사항이 늘어나면서 수정 쿼리도 점점 추가된다.

이런 방식의 문제점은 수정 쿼리가 많아지는 것은 물론이고 비즈니스 로직을 분석하기 위해 SQL을 계속 확인해야 한다. 결국 직접적이든 간접적이든 비즈니스 로직이 SQL에 의존하게 된다.

★ service → ContextService.Class

@Transactional

```
public void dirtyCheckingTest(){
    // 영속 엔티티 조회
    Member memberA = em.find(Member.class, "member1");

    // 영속 엔티티 데이터 수정
    memberA.setName("hi~~~");
}
```



☆ test → service → ContextTest.Class

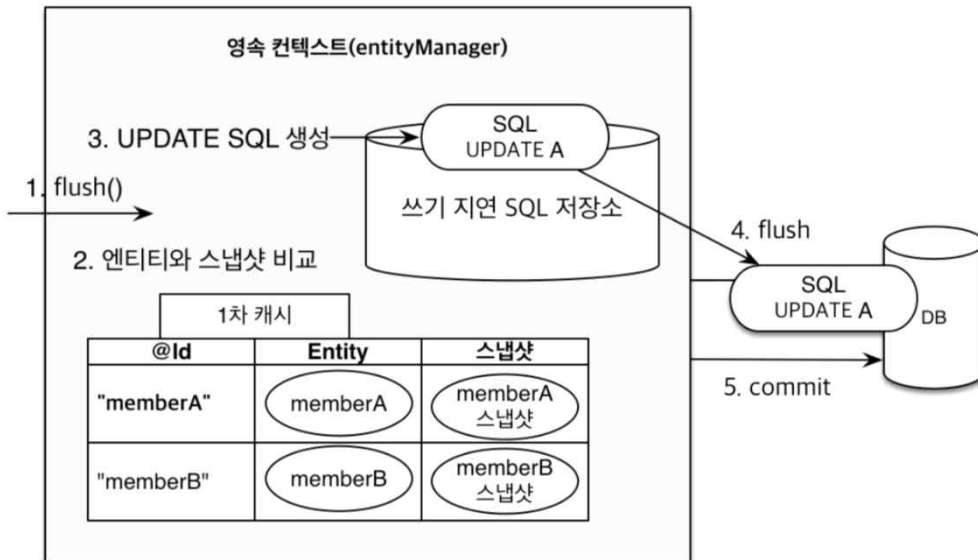
```
@Test
@DisplayName("Dirty Checking 테스트")
void dirtyCheckingTest(){
    contextService.dirtyCheckingTest();
}
```

위는 JPA의 엔티티 수정 코드다.

em.update() 혹은, em.save() 같은 메서드가 없이 member1에 대한 setter로만 변경했는데 어떻게 데이터베이스의 엔티티가 수정되는 걸까?

그 이유는 JPA가 변경 감지(dirty checking) 기능이 있기 때문이다.

변경 감지는 엔티티의 변경사항을 데이터베이스에 자동으로 반영하는 기능이다.



위 그림을 단계별로 살펴보자.

(JPA는 엔티티를 영속성 컨텍스트에 보관할 때, 최초 상태를 복사해서 저장해두는데 이것을 스냅샷이라 한다)

1. 트랜잭션을 커밋하면 엔티티 매니저 내부에서 먼저 플러시(flush())가 호출된다.
2. 엔티티와 스냅샷을 비교해서 변경된 엔티티를 찾는다.
3. 변경된 엔티티가 있으면 수정 쿼리를 생성해서 쓰기 지연 SQL 저장소에 보낸다.
4. 쓰기 지연 저장소의 SQL을 데이터베이스에 보낸다.
5. 데이터베이스 트랜잭션을 커밋한다.

**변경 감지**는 영속성 컨텍스트가 관리하는 영속 상태의 엔티티에만 적용된다  
비영속, 준영속처럼 영속성 컨텍스트의 관리를 받지 못하는 엔티티는 값을 변경해도  
데이터베이스에 반영되지 않는다

그럼 변경 감지로 인해 실행된 UPDATE SQL의 쿼리는 변경된 부분만 수정 쿼리가 생성  
될까?

아니다.

JPA의 기본 전략은 엔티티의 모든 필드를 업데이트 한다. 모든 필드를 업데이트하면 데  
이터베이스에 보내는 데이터 전송량이 증가하는 단점이 있지만, 다음과 같은 장점으로  
인해 모든 필드를 업데이트 한다.

모든 필드를 사용하면 수정 쿼리가 항상 같다. 따라서 애플리케이션 로딩 시점에 수정  
쿼리를 미리 생성해두고 재사용할 수 있다.

데이터베이스에 동일한 쿼리를 보내면 데이터베이스는 이전에 한 번 파싱된 쿼리를 재사  
용할 수 있다.

## ⑤ 엔티티 삭제

```
Member member = em.find(Member.class, "member1");  
em.remove(member);
```

em.remove()에 삭제 대상 엔티티를 넘겨주면 엔티티를 삭제한다.

물론 엔티티를 즉시 삭제하는 것이 아니라, 엔티티 등록과 비슷하게 삭제 쿼리를 쓰기 지연  
SQL 저장소에 등록한다.

이후 트랜잭션을 커밋해서 플러시를 호출하면 실제 데이터베이스에 삭제 쿼리를 전달한다.

참고: em.remove(member)를 호출하는 순간 member는 영속성 컨텍스트에서 제거된다.

## 플러시는!

- 영속성 컨텍스트를 비우지는 않음!!
- 영속성 컨텍스트의 변경내용을 데이터베이스에 동기화
- 트랜잭션이라는 작업 단위가 중요 -> 커밋 직전에만 동기화하면 됨

## ⑥ 기타 상태

- 엔티티를 준영속 상태로 전환: detach()
- 영속성 컨텍스트 초기화: clear()
- 영속성 컨텍스트 종료: close() -- 영속성 컨텍스트를 종료하면 해당 영속성 컨텍스트가 관리하던 영속 상태의 엔티티가 모두 준영속 상태가 된다.

# [JPA] 연관관계 매핑 기초 #1

## (연관관계의 필요성, 단방향 연관관계)

### 1. 연관관계가 필요한 이유

<시나리오>

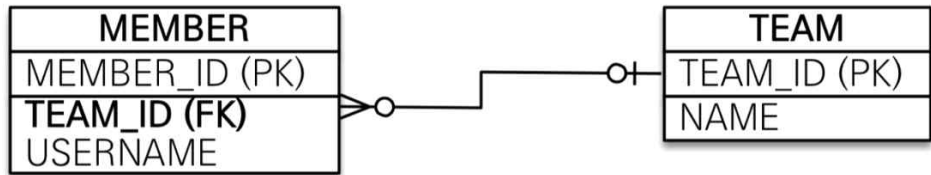
- 회원과 팀이 있다.
- 회원은 하나의 팀에만 소속될 수 있다.
- 회원과 팀은 다대일 관계다.

객체 테이블에 맞추어 모델링(참조 대신 외래 키를 그대로 사용)

[객체 연관관계]



[테이블 연관관계]



아래는 회원과 팀 클래스입니다.

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Member {
    @Id
    @Column(nullable = false, unique = true)
```

```
private String memberId;
private String name;
}

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Team {
    @Id
    @Column(nullable = false, unique = true)
    private String teamId;
    private String teamName;
}
```

아래는 팀과 회원을 저장하는 코드입니다.

```
@Service
public class RelationTestService {
    @Autowired
    EntityManager em;

    @Transactional
    public void insertMemberAndTeam(){
        Member member = new Member();
        member.setMemberId("member01");
        member.setName("김형민");
        member.setTeamId("team01");
        em.persist(member);

        Team team = new Team();
        team.setTeamId("team01");
        team.setTeamName("우리컴");
        em.persist(team);
    }
}
```

```

}

@SpringBootTest
public class RelationTest {
    @Autowired
    RelationTestService relationTestService;

    @Test
    @DisplayName("MemberAndTeam Insert Test")
    public void insertMemberAndTeam(){
        relationTestService.insertMemberAndTeam();
    }
}

```

이 상황에서 만약 회원의 팀을 찾으려면 어떻게 할까요?  
 방법은 다음과 같습니다.

```

@Test
@DisplayName("ID : member01의 팀이름 찾기 테스트")
public void member01TeamNameSearchTest(){
    Member member = em.find(Member.class, "member01");
    Team team = em.find(Team.class, member.getTeamId());
    System.out.println("Team name ----- " + team.getTeamName());
}

```

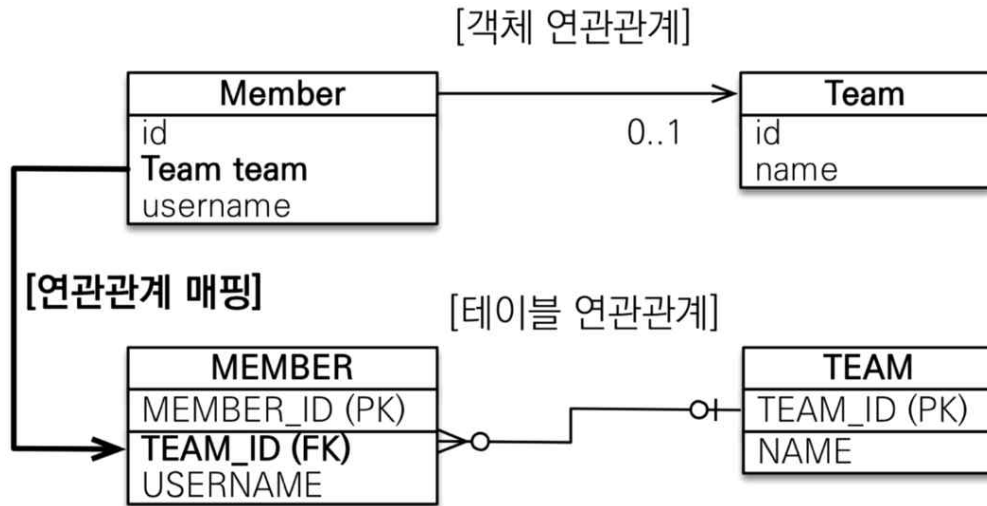
연관관계가 없다면 위와 같이, member의 team을 가져오는데 계속 꺼내와야합니다.  
 member를 꺼내고, id를 가져와서 다시 팀을 가져오는... 회원의 팀을 가져오기 위해 많은  
 비용이 듭니다.  
 그리고 객체 지향스럽지 않은 코드가 됩니다.

객체를 테이블에 맞추어 데이터 중심으로 모델링하면, 협력 관계를 만들 수 없습니다.

테이블은 외래 키로 조인을 사용해서 연관된 테이블을 찾습니다.  
 객체 참조를 사용해서 연관된 객체를 찾습니다.  
 테이블과 객체 사이에는 이런 큰 간격이 있습니다.

## 2. 단방향 연관관계

회원과 팀의 관계를 통해 다대일 단방향 관계를 알아보겠습니다.



### ● 객체 연관 관계

+ 회원 객체(Member)는 Member.team 필드(멤버 변수)로 팀 객체와 연관관계를 맺습니다. 회원 객체와 팀 객체는 단방향 관계입니다. 회원은 Member.team 필드를 통해 팀을 알 수 있지만, 팀 객체로 소속된 회원들을 알 수 없기 때문입니다. member → team 의 조회는 member.getTeam()으로 가능하지만, team → member 를 접근하는 필드는 없습니다.

### ● 테이블 연관관계

+ 회원 테이블은 TEAM\_ID 외래 키로 팀 테이블과 연관관계를 맺습니다.

회원 테이블과 팀 테이블은 양방향 관계입니다. 회원 테이블의 TEAM\_ID 외래 키를 통해서 회원과 팀을 조인할 수 있고, 반대로 팀과 회원도 조인할 수 있습니다. 예를 들어, MEMBER 테이블의 TEAM\_ID 외래 키 하나로 MEMBER JOIN TEAM 과 TEAM JOIN MEMBER 둘 다 가능합니다.

회원과 팀을 조인하는 SQL

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

팀과 회원을 조인하는 SQL

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

● 객체 연관관계와 테이블 연관관계의 가장 큰 차이

참조를 통한 연관관계는 언제나 단방향입니다. 객체간에 연관관계를 양방향으로 만드려 싶으면 반대쪽에도 필드를 추가해서 참조를 보관해야 합니다. 이렇게 양쪽에서 서로 참조하는 것을 양방향 연관관계라고 합니다.

하지만 정확히 이야기하면 이것은 양방향 관계가 아니라 서로 다른 단방향 관계 2개입니다. 반면에 테이블은 외래 키 하나로 양방향으로 조인할 수 있습니다.

● 객체 연관관계 vs 테이블 연관관계 정리

- + 객체는 참조(주소)로 연관관계를 맺습니다.
- + 테이블은 외래 키로 연관관계를 맺습니다.
- + 참조를 사용하는 객체의 연관관계는 단방향입니다.
- + 외래 키를 사용하는 테이블의 연관관계는 양방향입니다. (A JOIN B가 가능하면 반대로 B JOIN A도 가능)



## 연관관계 사용 테이블 재정의

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Member {
    @Id
    @Column(nullable = false, unique = true)
    private String memberId;
    private String name;

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

## 연관관계 사용 Create/Read/Update Test

```
--- Service
@Transactional
public void insertMemberAndTeamRelation(){
    Team team = new Team();
    team.setTeamId("team02");
    team.setTeamName("족구 팀");
    em.persist(team);

    Member member = new Member();
    member.setMemberId("member02");
    member.setName("강호동");
    member.setTeam(team);
    em.persist(member);
}
```

```

--- Test
@Test
@DisplayName("[Relation]MemberAndTeam Insert Test")
public void insertMemberAndTeamRelation() {
    relationTestService.insertMemberAndTeamRelation();
}

@Test
@DisplayName("ID : 단방향 연관관계 후 member01의 팀이름 찾기 및 수정")
public void member01TeamNameRelationSearchTest() {
    Team team = new Team();
    team.setTeamId("team03");
    team.setTeamName("씨름팀");

    Member member = em.find(Member.class, "member02");
    member.setTeam(team);
    System.out.println(member.getTeam().getTeamName());
}

```

## @ManyToOne, @JoinColumn

@ManyToOne : 다대일(N:1) 관계라는 매핑 정보입니다. 위에서 봤던 회원과 팀은 다대일 관계입니다. 연관관계를 매핑할 때 이렇게 다중성을 나타내는 어노테이션을 필수로 사용해야 합니다.

속성	기능	기본값
optional	false로 설정하면 연관된 엔티티가 항상 있어야 한다	true
fetch	글로벌 패치 전략을 설정한다. (자세한 내용은 추후에)	@ManyToOne=FetchType.EAGER, @OneToMany=FetchType.LAZY
cascade	영속성 전이 기능을 사용한다.(자세한 내용은 추후에)	
targetEntity	연관된 엔티티의 타입 정보를 설정한다. 이 기능은 거의 사용하지 않음	

@JoinColumn: 외래 키를 매핑할 때 사용합니다.

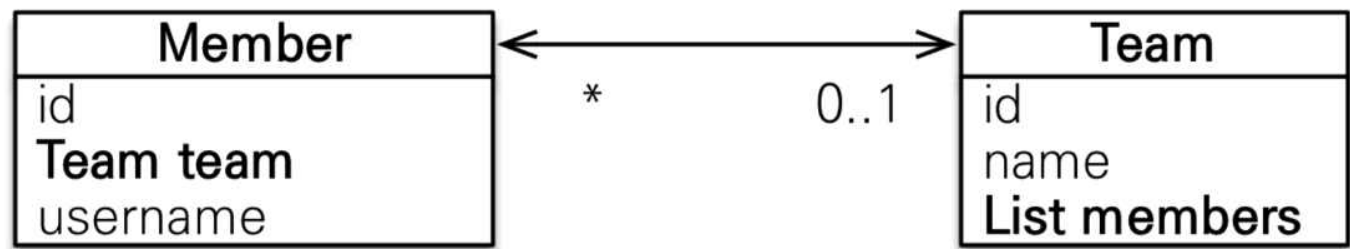
속성	기능	기본값
name	매핑할 외래 키 이름	"필드명" + "_" + "참조하는 테이블의 기본 키 컬럼명"
referencedColumnName	외래 키가 참조하는 대상 테이블의 컬럼명	참조하는 테이블의 기본 키 컬럼명

# [JPA] 연관관계 매핑 기초 #2

## (양방향 연관관계와 연관관계의 주인)

이번에는 팀에서 회원으로 접근하는 관계를 접근하는 관계를 추가해서, 양방향 연관관계로 매핑을 해보겠습니다.

[양방향 객체 연관관계]

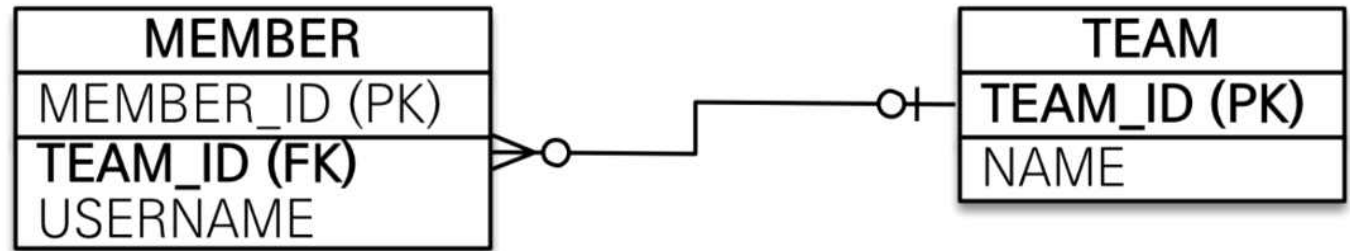


먼저 객체 연관관계를 살펴보겠습니다.  
위 그림을 보면, 회원과 팀은 다대일 관계입니다. 반대로 팀에서 회원은 일대다 관계입니다. 일대다 관계는 여러 건과 연관관계를 맺을 수 있으므로 컬렉션을 사용해야 합니다.

객체 연관관계는 다음과 같습니다.

- 회원 → 팀 (Member.team)
- 팀 → 회원 (Team.members)

[테이블 연관관계]



내가 소속된 팀 이름을 알고 싶으면 TEAM 테이블의 TEAM\_ID로 MEMBER 테이블에 조인하면 됩니다.

팀에 소속된 멤버들을 알고 싶으면 MEMBER의 TEAM\_ID로 TEAM 테이블에 join하면 됩니다.

따라서 데이터베이스 테이블은 외래 키 하나로 양방향으로 조회할 수 있습니다.

## 1. 양방향 연관관계 매핑

이제 코드를 통해 양방향 관계를 매핑하는 것을 알아보시다.

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Member {
    @Id
    @Column(nullable = false, unique = true)
    private String memberId;
    private String name;

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

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Team {
    @Id
    @Column(nullable = false, unique = true)
    private String teamId;
    private String teamName;

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

```
}
```

팀과 회원은 일대다 관계입니다. 따라서 팀 엔티티에 `List<Member> members`를 추가했습니다. 그리고 일대다 관계를 매핑하기 위해 `@OneToMany` 매핑 정보를 사용했습니다. `@OneToMany`의 `mappedBy` 속성은 양방향 매핑일 때 사용하는데, 반대쪽 매핑의 필드 이름을 값으로 주면 됩니다. 반대쪽 매핑이 `Member.team`이므로 `team`을 값으로 주었습니다.

### 연관관계의 주인

이전까지 `@ManyToOne`, `@OneToMany`에 대해 알아보았습니다. `Team`의 `@OneToMany`에 `mappedBy` 속성은 왜 있을까요?

엄밀히 이야기하면 객체에는 양방향 연관관계라는 것이 없습니다. 서로 다른 단방향 연관관계 2개를 애플리케이션 로직으로 잘 묶어서 양방향인 것처럼 보이게 할 뿐입니다. 반면에 데이터베이스 테이블은 외래 키 하나로 양쪽이 서로 조인할 수 있습니다. 따라서 테이블은 외래 키 하나만으로 양방향 연관관계를 맺습니다.

※ 객체 연관관계 = 2개

- 회원 → 팀 : 연관관계 1개(단방향)
- 팀 → 회원 : 연관관계 1개(단방향)

※ 테이블 연관관계 = 1개

- 회원 ↔ 팀 : 연관관계 1개(양방향)

엔티티를 양방향 연관관계로 설정하면 객체의 참조는 둘인데 외래 키는 하나입니다. 따라서 둘 사이에 차이가 발생합니다. 그렇다면 둘 중 어떤 관계를 사용해서 외래 키를 관리해야 할까요?

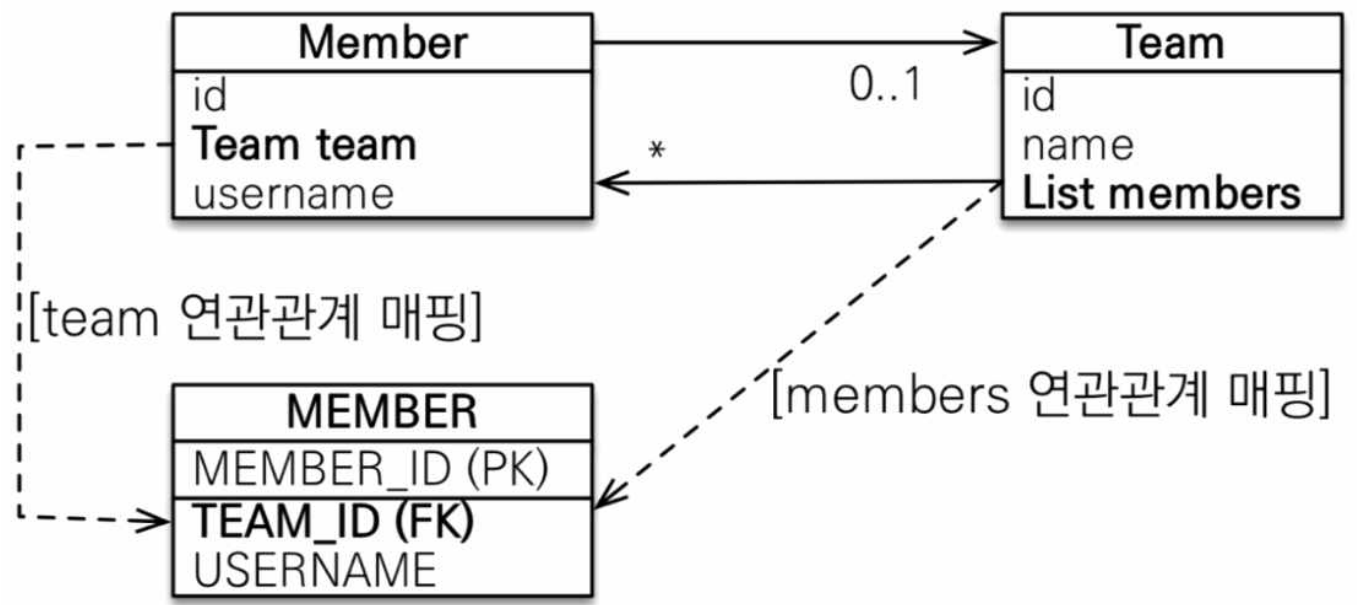
이런 차이로 인해 JPA에서는 두 객체 연관관계 중 하나를 정해서 테이블의 외래키를 관리해야 하는데 이것을 연관관계의 주인이라 합니다.

### 양방향 매핑의 규칙: 연관관계의 주인

양방향 연관관계 매핑 시 지켜야할 규칙이 있는데 두 연관관계 중 하나를 연관관계의 주인

으로 정해야 합니다. 연관관계의 주인만이 데이터베이스 연관관계와 매핑되고 외래 키를 관리(등록, 수정, 삭제)할 수 있습니다. 반면에 주인이 아닌 쪽은 읽기만 할 수 있습니다. 어떤 연관관계를 주인으로 정할지는 mappedBy 속성을 사용하면 됩니다.

주인은 mappedBy 속성을 사용하지 않는다.  
주인이 아니면 mappedBy 속성을 사용해서 속성의 값으로 연관관계의 주인을 지정해야 한다.

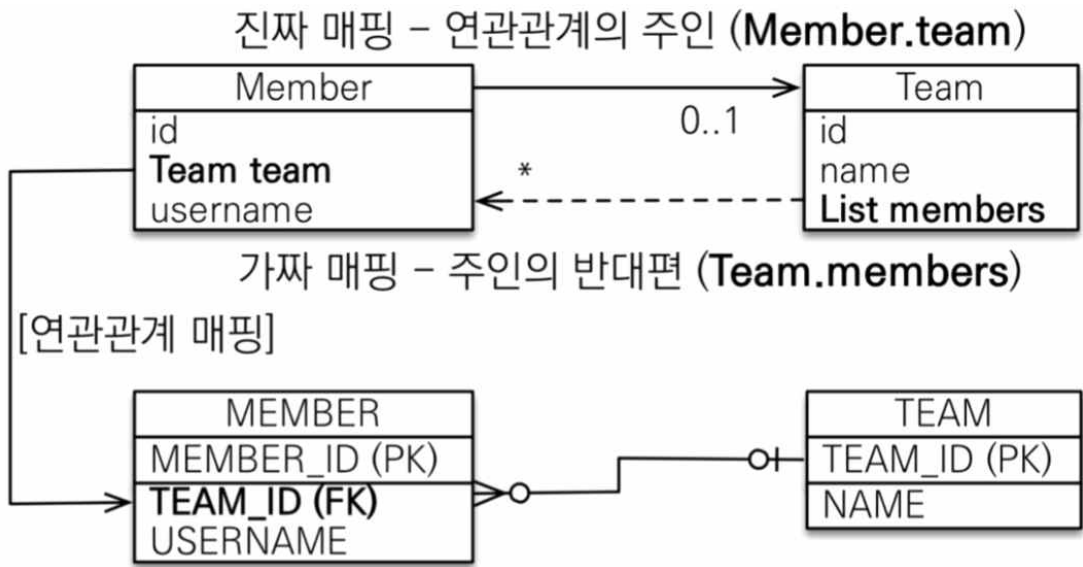


연관관계의 주인을 정한다는 것은 사실 외래 키 관리자를 선택하는 것입니다. 그림을 보면, 여기서는 회원 테이블에 있는 TEAM\_ID 외래 키를 관리할 관리자를 선택해야 합니다. 만약 회원 엔티티에 있는 Member.team을 주인으로 선택하면 자기 테이블에 있는 외래 키를 관리하면 됩니다. 하지만 팀 엔티티에 있는 Team.members를 주인으로 선택하면 물리적으로 전혀 다른 테이블의 외래 키를 관리해야 합니다. 왜냐하면 이 경우 Team.members가 있는 Team 엔티티는 TEAM 테이블에 매핑되어 있는데 관리해야 할 외래 키는 MEMBER에 있기 때문입니다.

연관관계의 주인은 외래 키가 있는 곳

연관관계의 주인은 테이블에 외래 키가 있는 곳으로 정해야 합니다. 여기서는 회원 테이블이 외래 키를 가지고 있으므로 Member.team이 주인이 됩니다. 주인이 아닌 Team.members

에는 `mappedBy="team"` 속성을 사용해서 주인이 아님을 설정해야 합니다. 여기서 `mappedBy`의 값으로 사용된 `team`은 연관관계의 주인인 `Member` 엔티티의 `team` 필드를 말합니다.



정리하면, 연관관계의 주인만 데이터베이스 연관관계와 매핑되고 외래 키를 관리할 수 있습니다. 주인이 아닌 반대편은 읽기만 가능하고 외래 키를 변경하지 못합니다.

데이터베이스 테이블의 다대일, 일대다 관계에서는 항상 다 쪽이 외래 키를 가집니다. 다 쪽인 `@ManyToOne`은 항상 연관관계의 주인이 되므로 `mappedBy`를 설정할 수 없습니다. 따라서 `@ManyToOne`에는 `mappedBy` 속성이 없습니다.

## 양방향 연관관계 저장

```
---- Service
@Transactional
public void insertBothDirectionRelation(){
    //팀1 저장
    Team team = new Team();
    team.setTeamId("team1");
    team.setTeamName("족구 팀");
```



```

em.persist(team);

//회원1 저장
Member member1 = new Member();
member1.setMemberId("member1");
member1.setName("강호동");
member1.setTeam(team);
em.persist(member1);

//회원2 저장
Member member2 = new Member();
member2.setMemberId("member2");
member2.setName("이만기");
member2.setTeam(team);
em.persist(member2);
}
---- Test
@Test
@DisplayName("[Relation] 양방향 연관관계 입력")
public void bothRelationInputTest(){
    relationTestService.insertBothDirectionRelation();
}

```

위 코드를 보면, 팀1을 저장하고 회원1, 회원2에 연관관계의 주인인 Member.team 필드를 통해서 회원과 팀의 연관관계를 설정하고 저장했습니다.

데이터베이스에서 회원 테이블을 조회해보겠습니다. ( SELECT \* FROM MEMBER; )

member_id	name	team_id
member1	강호동	team1
member2	이만기	team1

TEAM\_ID 외래 키에 팀의 기본 키 값이 저장되어 있습니다.

양방향 연관관계는 연관관계의 주인이 외래 키를 관리합니다. 따라서 주인이 아닌 방향은

값을 설정하지 않아도 데이터베이스에 외래 키 값이 정상 입력됩니다.

```
team1.getMembers().add(member1); // 무시(연관관계의 주인이 아님)
team1.getMembers().add(member2); // 무시(연관관계의 주인이 아님)
```

이런 코드가 추가로 있어야 할 것 같지만 Team.members는 연관관계의 주인이 아닙니다. 주인이 아닌 곳에 입력된 값은 외래 키에 영향을 주지 않습니다.

## 양방향 연관관계 후 팀1의 멤버이름 출력

```
@Test
@DisplayName("[Relation] 입력 후 팀1의 멤버 출력 테스트")
public void inputAfterTeamMemberSearch(){
    Team team = em.find(Team.class,"team1");
    List<Member> members = team.getMemberList();
    System.out.println("members.size = " + members.size());
    for(Member m : members){
        System.out.println("-----" + m.getName());
    }
}
```

## 양방향 연관관계의 주의점

```
---- Service
@Transactional
public void insertErrorBothDirectionRelation(){
    //회원1 저장
    Member member1 = new Member();
    member1.setMemberId("member1");
    member1.setName("강호동");
    em.persist(member1);

    //회원2 저장
    Member member2 = new Member();
    member2.setMemberId("member2");
```

```

        member2.setName("이만기");
        em.persist(member2);





        //팀1의 멤버리스트에 멤버 추가
        Team team = new Team();
        team.setTeamId("team1");
        team.setTeamName("족구");
        team.getMemberList().add(member1);
        team.getMemberList().add(member2);
        em.persist(team);
    }
}

---- Test
@Test
@DisplayName("[Relation] 연관관계 주인 아닌곳에 입력 테스트")
public void bothRelationInputErrorTest(){
    relationTestService.insertErrorBothDirectionRelation();
}

```

회원1, 회원2를 저장하고 팀의 컬렉션에 담은 후에 팀을 저장했습니다. 과연 위 코드가 정상적으로 저장되었을까요?

위 코드를 실행하고 데이터 베이스에서 회원 테이블을 조회하면 다음과 같은 결과가 나옵니다.

Result Grid			
Filter Rows: <input type="text"/>			
Edit:     Export/Imp			
	member_id	name	team_id
▶	member1	강호동	NULL
	member2	이만기	NULL

외래 키 TEAM\_ID에 team1이 아닌 null 값이 입력되어 있는데, 연관관계의 주인이 아닌 Team.members에만 값을 저장했기 때문입니다. 다시 한번 강조하지만 연관관계의 주인만이 외래 키의 값을 변경할 수 있습니다.

연관관계의 주인인 Member.team에 아무 값도 입력하지 않았기 때문에, TEAM\_ID 외래 키의 값도 null이 저장됩니다.

## 순수한 객체까지 고려한 양방향 연관관계

그렇다면 정말 연관관계의 주인에만 값을 저장하고 주인이 아닌 곳에는 값을 저장하지 않아도 될까요?

사실은 객체 관점에서 양쪽 방향에 모두 값을 입력해주는 것이 안전합니다. 양쪽 방향 모두 값을 입력하지 않으면 JPA를 사용하지 않는 순수한 객체 상태에서 심각한 문제가 발생할 수 있습니다.

예를 들어 JPA를 사용하지 않고 엔티티에 대한 테스트 코드를 작성한다고 가정해봅시다. ORM은 객체와 관계형 데이터베이스 둘 다 중요합니다. 데이터베이스뿐만 아니라 객체도 함께 고려해야 합니다.

```
@Transactional
public void insertCompleteRelationTest01(){
    //팀1의 멤버리스트에 멤버 추가
    Team team = new Team();
    team.setTeamId("team1");
    team.setTeamName("족구팀");
    em.persist(team);

    //회원1 저장
    Member member1 = new Member();
    member1.setMemberId("member1");
    member1.setName("강호동");
    member1.setTeam(team);
    em.persist(member1);

    //회원2 저장
    Member member2 = new Member();
    member2.setMemberId("member2");
    member2.setName("이만기");
    member2.setTeam(team);
    em.persist(member2);

    List<Member> members = team.getMemberList();
```

```

        System.out.println("member_size = " + members.size());
    }

    @Test
    @DisplayName("[Relation] 팀 멤버리스트에 추가 안할 경우 테스트")
    public void completeInputTest01(){
        relationTestService.insertCompleteRelationTest01();
    }

```

OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader

member\_size = 0

Hibernate:

코드를 보면 Member.team에만 연관관계를 설정하고 반대 방향은 연관관계를 설정하지 않았습니다. 마지막 줄에서 팀에 소속된 회원이 몇 명인지 출력해보면 결과는 0이 나옵니다. 이것은 우리가 기대하는 양방향 연관관계의 결과가 아닙니다. 따라서 양방향은 양쪽다 관계를 설정해야 합니다. 이처럼 회원 → 팀 을 설정하면 다음 코드처럼 반대방향인 팀 → 회원도 설정해야 합니다. 양쪽 모두 관계를 설정한 코드를 보겠습니다.

```

@Transactional
public void insertCompleteRelationTest02(){
    //팀1의 멤버리스트에 멤버 추가
    Team team = new Team();
    team.setTeamId("team1");
    team.setTeamName("족구팀");
    em.persist(team);

    //회원1 저장
    Member member1 = new Member();
    member1.setMemberId("member1");
    member1.setName("강호동");
}

```

```

        member1.setTeam(team);
        team.getMemberList().add(member1);
        em.persist(member1);

        //회원2 저장
        Member member2 = new Member();
        member2.setMemberId("member2");
        member2.setName("이만기");
        member2.setTeam(team);
        team.getMemberList().add(member2);
        em.persist(member2);

        List<Member> members = team.getMemberList();
        System.out.println("member_size = " + members.size());
    }

    @Test
    @DisplayName("[Relation] 팀 멤버리스트에 추가한 경우 테스트")
    public void completeInputTest02(){
        relationTestService.insertCompleteRelationTest02();
    }
}

```

OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader

member\_size = 2

Hibernate:

양쪽 모두 관계를 설정하니 기대했던 결과값 2가 나옵니다.  
순수한 객체 상태에서도 동작하며, 테이블의 외래 키도 정상 입력됩니다.

## 연관관계 편의 메서드

양방향 연관관계는 결국 양쪽 다 신경 써야 합니다.

member.setTeam(team)과 team.getMembers().add(member)를 각각 호출하다 보면 실수로 둘 중 하나만 호출해서 양방향에 깨질 수 있습니다.

그래서 Member 클래스의 setTeam() 메서드를 수정해서 코드를 리팩토링 해보겠습니다.

```
public class Member{

    private Team team;

    public void setTeam(Team team) {
        this.team = team;
        team.getMembers().add(this);
    }
    // ...
}
```

setTeam() 메서드 하나로 양방향 관계를 모두 설정하도록 변경했습니다.  
이렇게 한 번에 양방향 관계를 설정하는 메서드를 연관관계 편의 메서드라 합니다.

## 연관관계 편의 메서드 작성 시 주의사항

연관관계를 변경할 때는 기존 팀이 있으면 기존 팀과 회원의 연관관계를 삭제하는 코드를 추가해야 합니다.

아래는 Member 클래스에 연관관계 편의메소드 addTeam() 추가

```
public class Member {
    @Id
    @Column(nullable = false, unique = true)
    private String memberId;
    private String name;

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

```
public void addTeam(Team team){
    if(this.team != null){
        this.team.getMemberList().remove(this);
    }
    this.team = team;
    team.getMemberList().add(this);
}
}
```

---

※ 편의 메소드 적용

---

@Transactional

```
public void insertCompleteRelationTest03(){
    //팀1의 멤버리스트에 멤버 추가
    Team team = new Team();
    team.setTeamId("team1");
    team.setTeamName("족구팀");
    em.persist(team);

    //회원1 저장
    Member member1 = new Member();
    member1.setMemberId("member1");
    member1.setName("강호동");
    member1.addTeam(team);
    em.persist(member1);

    //회원2 저장
    Member member2 = new Member();
    member2.setMemberId("member2");
    member2.setName("이만기");
    member2.addTeam(team);
    em.persist(member2);
}
```



```
List<Member> members = team.getMemberList();  
System.out.println("member_size = " + members.size());  
}
```

```
@Test
```

```
@DisplayName("[Relation] 편의메소드 추가 후 테스트")
```

```
public void completeInputTest03(){
```

```
    relationTestService.insertCompleteRelationTest02();
```

```
}
```