JPA(Java Persistence API)

SQL 중심 개발의 문제점

- 무한반복, 지루한 코드가 된다.
- **SQL**에 **의존적인 개발**을 피하기 어렵다.
- 객체지향프로그래밍 vs 관계형 DB의 패러다임 불일치
- 객체의 저장은 어쩔 수 없이 관계형DB에 이루어진다.

- 상속관계, 연관관계,

JPA 개념

- 자바 진영의 ORM 기술 표준

ORM(Object-relational mapping) - 객체 관계 매핑

- 객체는 객체대로 설계
- 관계형 데이터베이스는 관계형 데이터 베이스대로 설계
- ORM 프레임워크가 중간에서 매핑

JAVA 애플리케이션 -> JPA -> JDBC API <=> DB (패러다임의 불일치 해결)

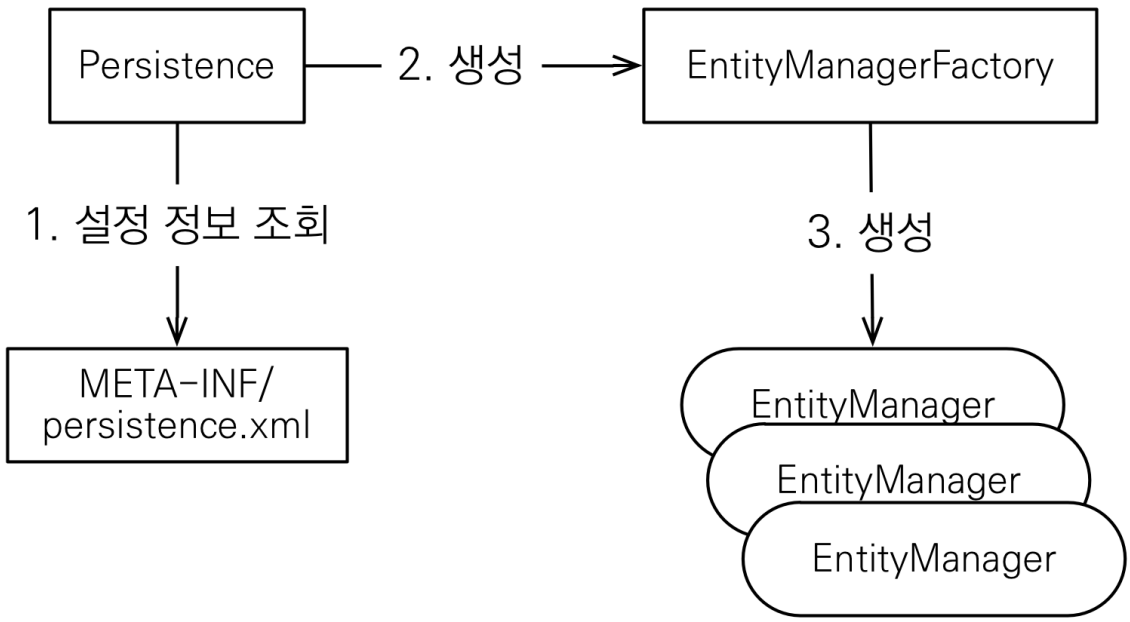
JPA는 표준 명세

- 인터페이스의 모음
- 하이버네이트: 구현체

JPA의 필요성

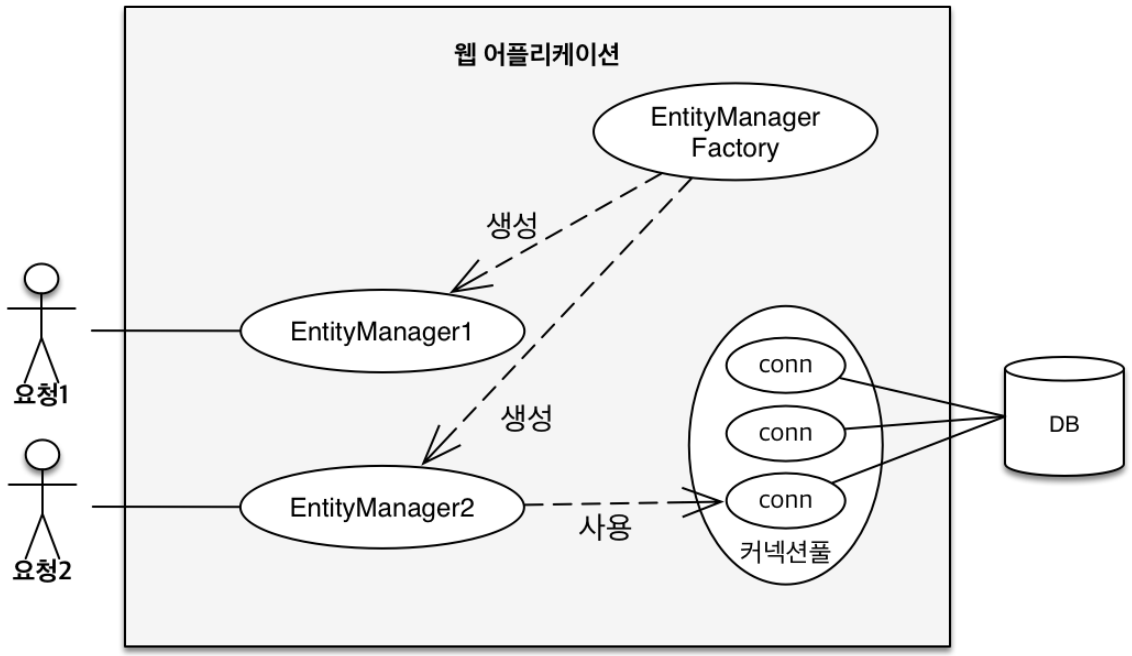
- 생산성이 좋다.
- 유지 보수에 용이 (기존: 필드 변경시 모든 SQL 수정)
- 패러다임의 불일치 해결

JPA 구동 방식



영속성 관리

EntityMangerFactory & EntityManger 구조



EntityMangerFactory

- 데이터베이스 하나당 어플리케이션은 일반적으로 하나의 EntityMangerFactory를 생성한다.
- 한 개의 EntityMangerFactory로 어플리케이션 전체에 공유하도록 설계한다.
- 여러 스레드가 동시에 접근해도 안전, 서로 다른 스레드 간 공유가 가능하다.

EntityManger

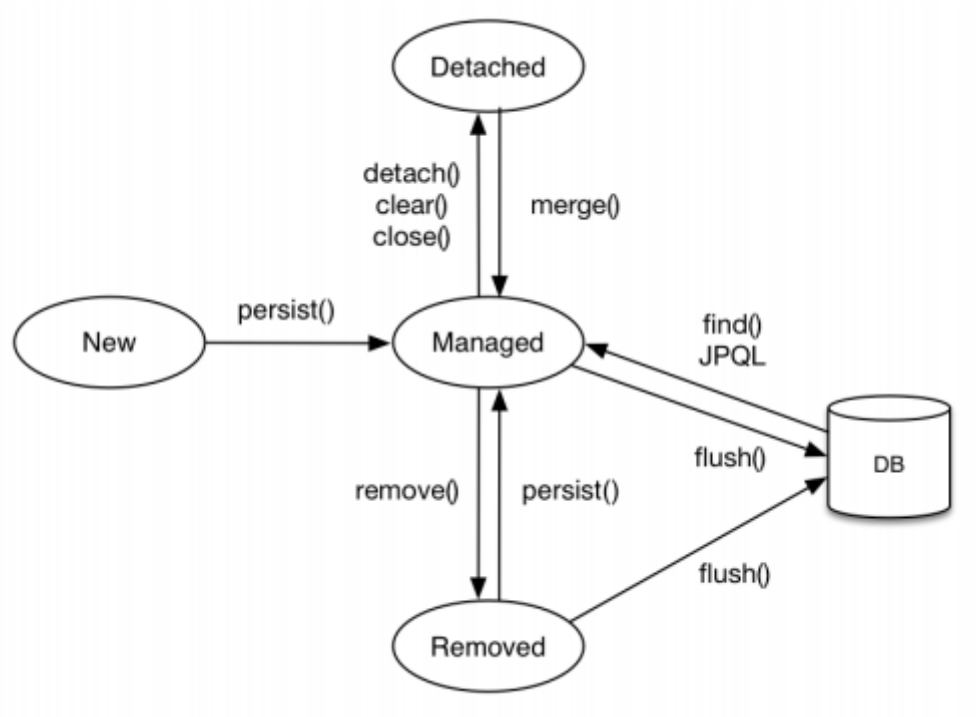
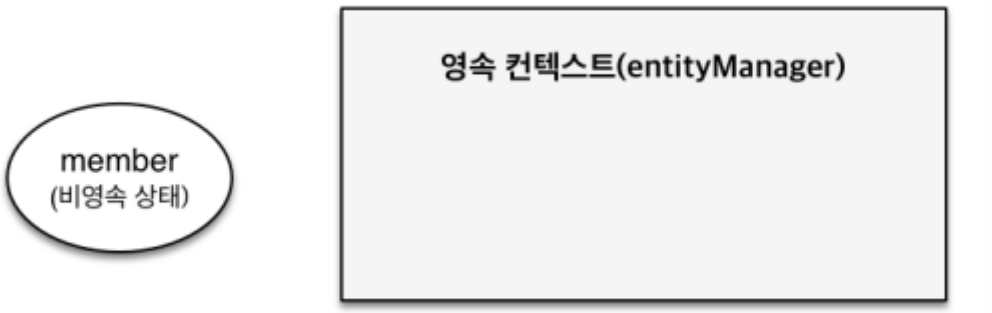
- 여러 스레드가 동시에 접근하면 동시성 문제가 발생한다.
- **스레드간의 공유는 절대 하면 안된다!!**
- 데이터베이스 연결이 필요한 시점까지 커넥션을 얻지 않는다.

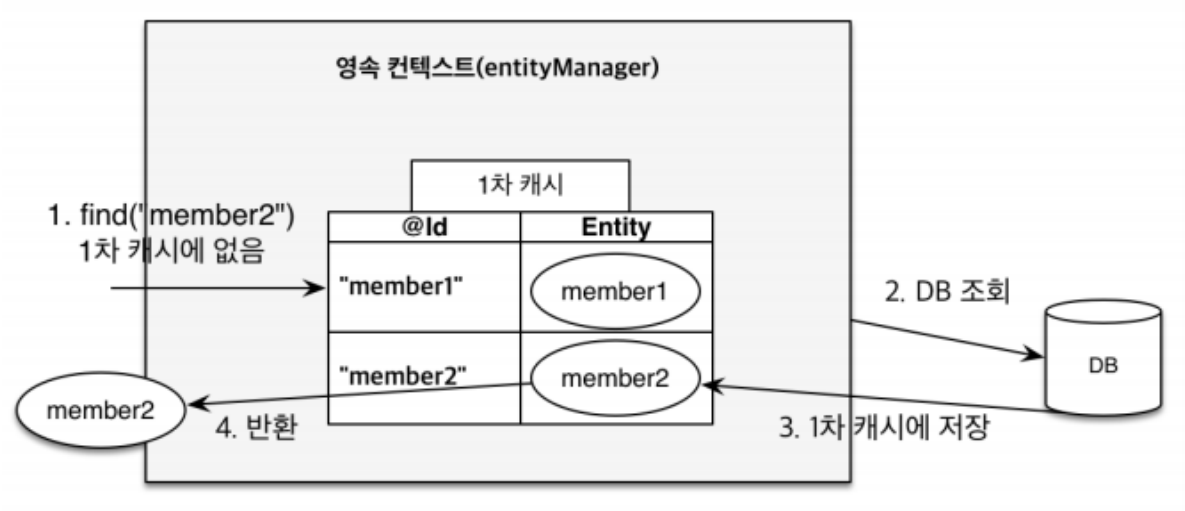
영속성 컨텍스트

- 엔티티를 영구 저장하는 환경
- 엔티티를 식별자 값(@id로 테이블의 기본 키와 매핑한 값)으로 구분한다.

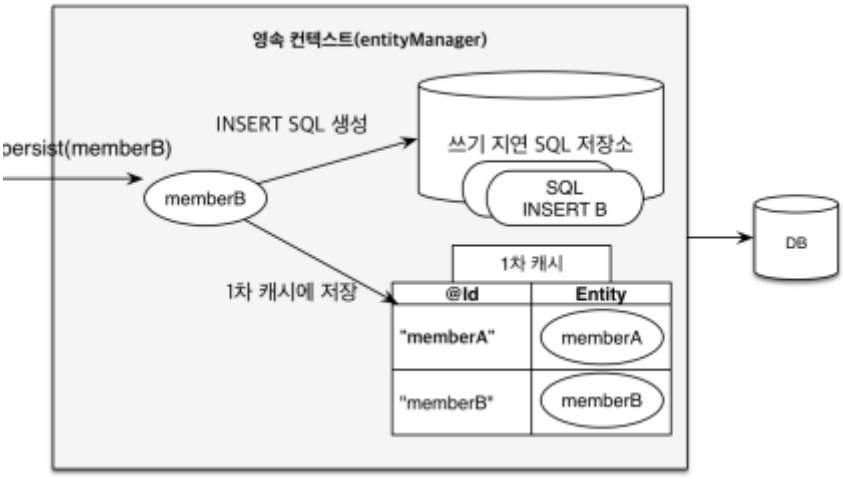
```
@Entity
public class Member{
    @Id
    private Long id;
    private String name;
}
```

- ```
  // 엔티티 영속
  EntitiyManger.persist(entity);
  ```

- 엔티티 매니저를 사용해서 회원 엔티티를 영속성 컨텍스트에 저장
- **엔티티 매니저를 통해서 영속성 컨텍스트에 접근하고 관리** 할 수 있다.
- 엔티티 매니저를 생성할 때 하나 만들어진다.
  ```

엔티티의 생명주기

- 비영속(new/transient) : 영속성 컨텍스트와 전혀 관계가 없는 상태
- 영속(managed): 영속성 컨텍스트에 **관리**되는 상태
- 준영속(detached): 영속성 컨텍스트에 저장되었다가 **분리**된 상태
- 삭제(removed): **삭제**된 상태



- **비영속**




```
// 객체를 생성한 상태(비영속)
Member member = new Member();
member.setId(100L);
member.setName("회원1");
```

- **영속**


```
// 객체 생성
Member member = new Member();
member.setId(100L);
member.setName("회원1");

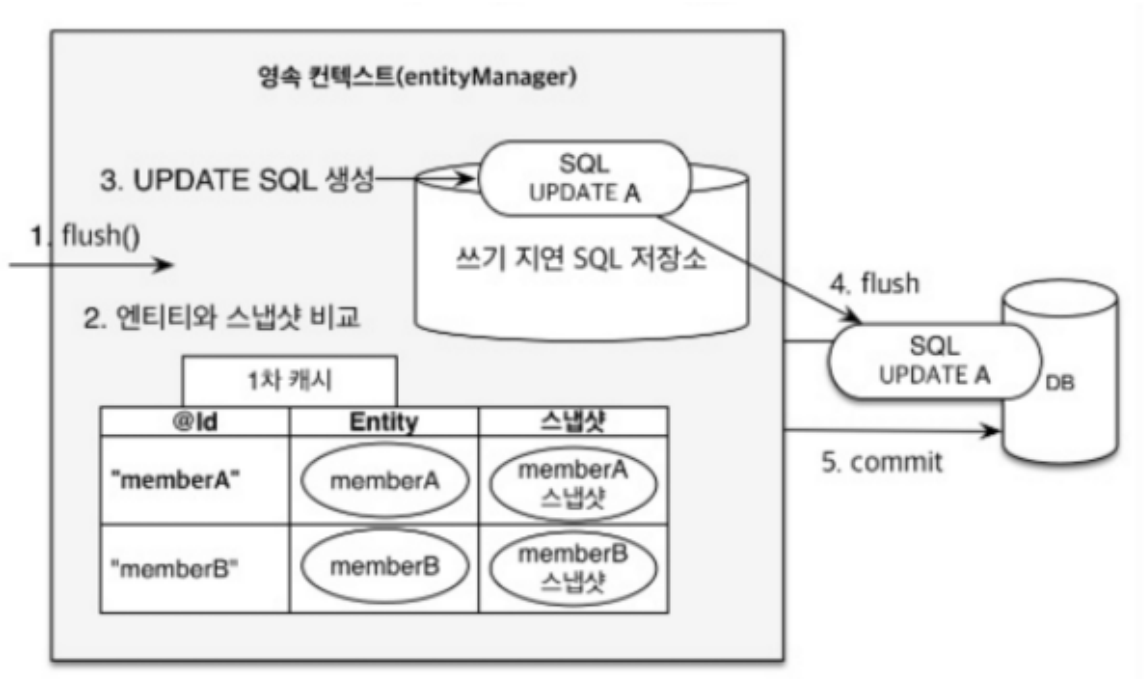
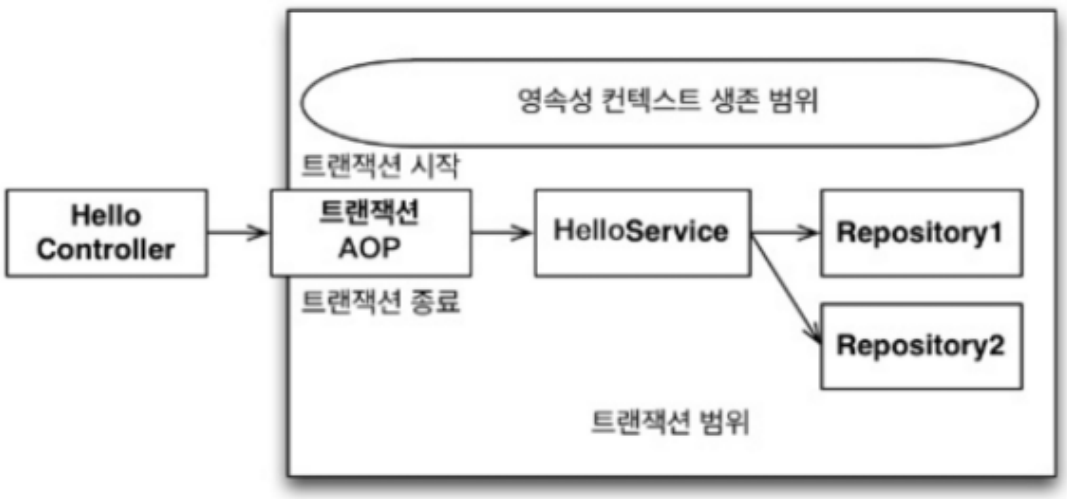
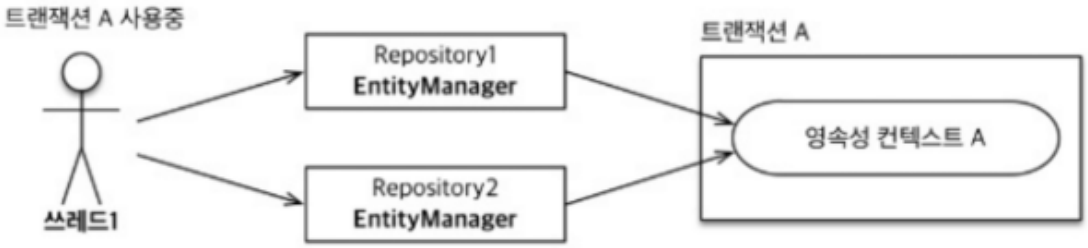
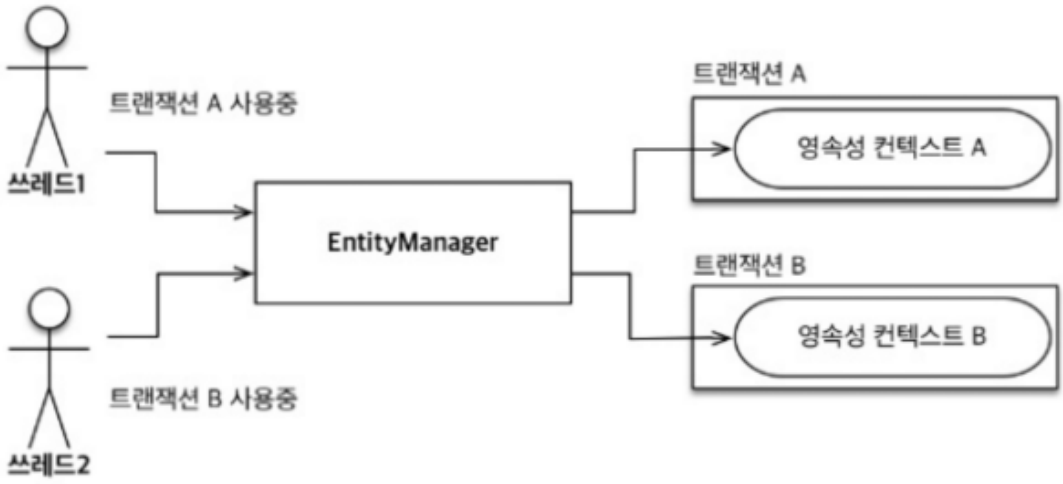
EntitiyManager em = emf.createEntitiyManager();
EntitiyTransaction tx = em.getTransaction();
// 트렌스젝션 시작
tx.begin();
// 객체를 저장한 상태(영속)
System.out.println("=== BEFORE ===");
// 1차 캐시에 저장됨
em.persist(member);
System.out.println("=== AFTER ===");
// 커밋하는 순간 DB에 INSERT SQL를 보낸다.
tx.commit();
```

```
=== BEFORE ===
=== AFTER ===
Hibernate:
    /* insert hellojpa.Member
        */ insert
        into
            Member
            (name, id)
        values
            (?, ?)
```

결과

- em.persist(member) 당시에는 아직DB에 저장되지 않은 상태이다. (1차 캐시에 저장됨)
- tx.commit() 이 이루어지는 순간에 DB에 INSERT SQL를 보낸다.

영속성 컨텍스트의 이점

- 1차 캐시
- 동일성 보장
- 트랜잭션을 지원하는 쓰기 지연
- 변경 감지(Dirty Checking)
- 지연 로딩

데이터베이스 조회과정

```
Member findMember1 = em.find(Member.class, 150L);
Member findMember2 = em.find(Member.class, 150L);
```



- 처음 조회할때 DB에서 가져오는것이 아닌 **1차캐시**에서 가져온다.
- 1차캐시에 없고, DB에 있는 경우 **DB에서 select문으로 1차 캐시**로 가져온다.
- findMember1에서 select문으로DB에서 가져오고, findMember2에서는 1차 캐시에서 가져온다.

실행결과

```
Hibernate:
    select
        member0_.id as id1_0_0_,
        member0_.name as name2_0_0_
    from
        Member member0_
    where
        member0_.id=?
findMember.getId() = 150
findMember.getName() = memberA
```

트랜잭션을 지원하는 쓰기 지연

```
EntitiyManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
// 엔티티 매니저는 데이터 변경시 트랜잭션을 시작해야 한다.
tx.begin();

// 쓰기지연 SQL저장소에 INSERT문이 저장된다.
em.persist(memberA);
em.persist(memberB);

// 트랜잭션 커밋이 이루어질때 INSERT SQL를 DB에 보낸다.
tx.commit();
```



엔티티 수정- 변경감지(Dirty Checking)

```
EntitiyManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

Member memberA = em.find(Member.class, "memberA");

// update문을 따로 작성하지 않아도 된다.
memberA.setUsername("memberAAA");
memberA.setAge(10);

tx.commit();
```



수정순서

1. 트랜잭션 커밋 -> 엔티티 매니저 내부에서 플러시 호출
2. 엔티티와 스냅샷을 비교해서 변경된 엔티티를 찾는다.
3. 변경된 엔티티가 있으면 수정 쿼리를 생성해서 쓰기 지연 SQL저장소로 보낸다.
4. 쓰기 지연 저장소의 SQL을 DB로 보낸다.
5. 데이터베이스 트랜잭션을 커밋한다.

플러시

- 영속성 컨텍스트의 변경내용을 데이터베이스에 반영한다.
- 플러시 실행 메커니즘
    1. 변경 감지 동작(모든 엔티티를 스냅샷과 비교)
    2. 수정된 엔티티는 수정쿼리를 만들어 **쓰기 지연 SQL저장소** 등록
    3. 쓰기 지연 SQL 저장소의 쿼리를 DB에 전송
- **플러시를 한다고 1차 캐시가 사라지는 것은 아니다.(영속성 컨텍스트에 보관된 엔티티는 유지된다.)**
- **변경된 내용을 DB에 적용하는 것 뿐이다.**

플러시하는 방법

1. em.flush() 직접 호출
2. 트랙잭션 커밋시 플러시가 자동 호출
3. JPQL 쿼리 실행시 플러시가 자동 호출

- 직접 호출은 자주 사용하지는 않는다.

준영속 상태

- 영속상태 -> 준영속 상태
- 영속 상태의 엔티티가 영속성 컨텍스트에서 분리(detached)
- 영속성 컨텍스트가 제공하는 기능을 사용 못함

트랜잭션 범위의 영속성 컨텍스트

- @Transaction 어노테이션



- 트랜잭션이 같으면 같은 영속성 컨텍스트 사용한다.



- 트랜잭션이 다르면 다른 영속성 컨텍스트를 사용한다.



엔티티 매핑

엔티티 매핑 소개

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

@Entity

- @Entity가 붙은 **클래스**는 JPA가 관리, **엔티티라고 한다.**
- JPA를 사용해서 테이블과 매핑할 클래스는 **@Entity**가 필수이다.
- 기본 생성자를 필수로 작성해주자(파라미터가 없는 public 또는 protected 생성자)
- final클래스, enum, interface, inner 클래스에는 사용할 수 없다.
- 저장할 필드에 final을 사용하면 안된다.

```
@Entity
public class Member{
    private Long id;
    private String name;
}
```

```
// 기본 생성자를 생성해준다.
public Member(){}

// 사용할 생성자
public Member(Long id, String name){
    this.id = id;
    this.name = name;
}
```

@Table

- 엔티티와 매핑할 테이블을 지정한다.

```
@Entity
@Table(name="MEMBER")
public class Member{
    ...
}
```

데이터베이스 스키마 자동 생성

- DDL(Date Definition Language)을 애플리케이션 실행 시점에 자동 생성
-

옵션	설명
create	기존테이블 삭제 후 다시 생성(DROP + CREATE)
create-drop	create와 같으나 종료시점에 테이블 DROP
update	변경분만 반영(운영DB에는 절대 사용하면 안된다.)
validate	엔티티와 테이블이 정상 매핑되었는지 확인
none	사용하지 않음

-
```
<property name="hibernate.hbm2ddl.auto" value="create" />
```

- **validate 또는 none만 사용할 것을 권장한다!!**

다양한 매핑 사용

```
@Entity
public class Member {

    @Id // Pk로 매핑
    private Long id;

    // 컬럼 매핑
    @Column(name = "name")
    private String username;

    private Integer age;

    // 자바 enum 타입을 매핑할 때 사용
    @Enumerated(EnumType.STRING)
```

```
    private RoleType roleType;

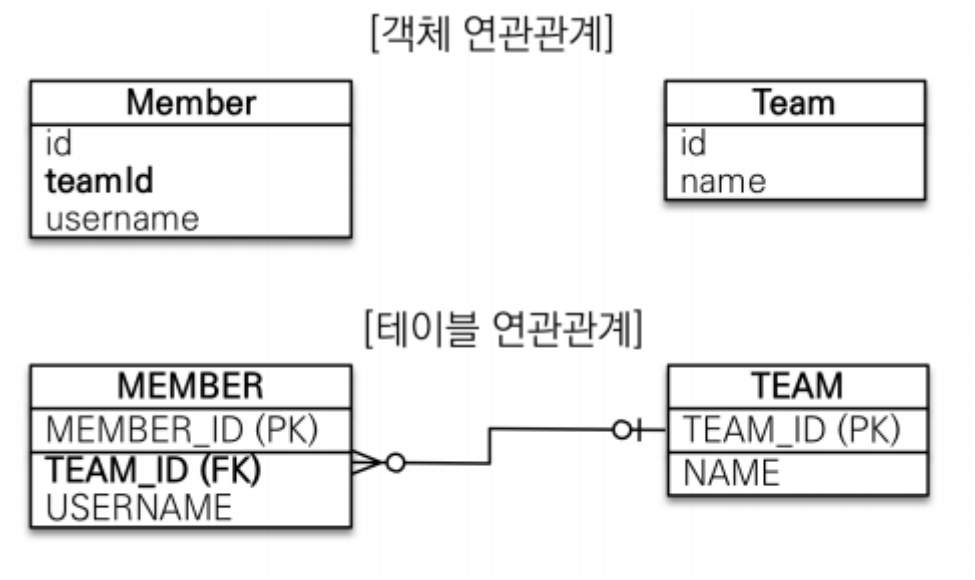
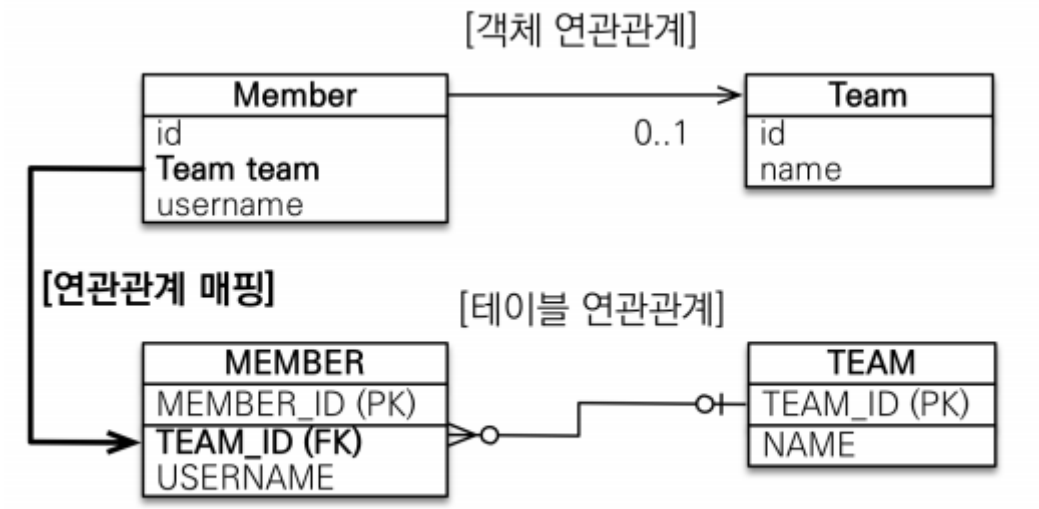
    // 날짜 타입을 매핑할 때 사용
    @Temporal(TemporalType.TIMESTAMP)
    private Date createdDate;

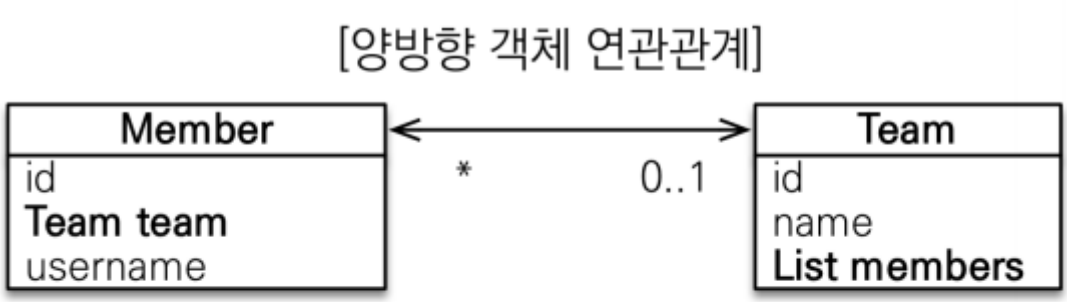
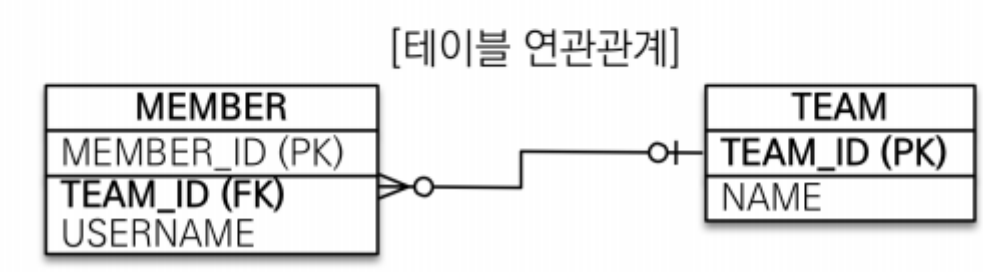
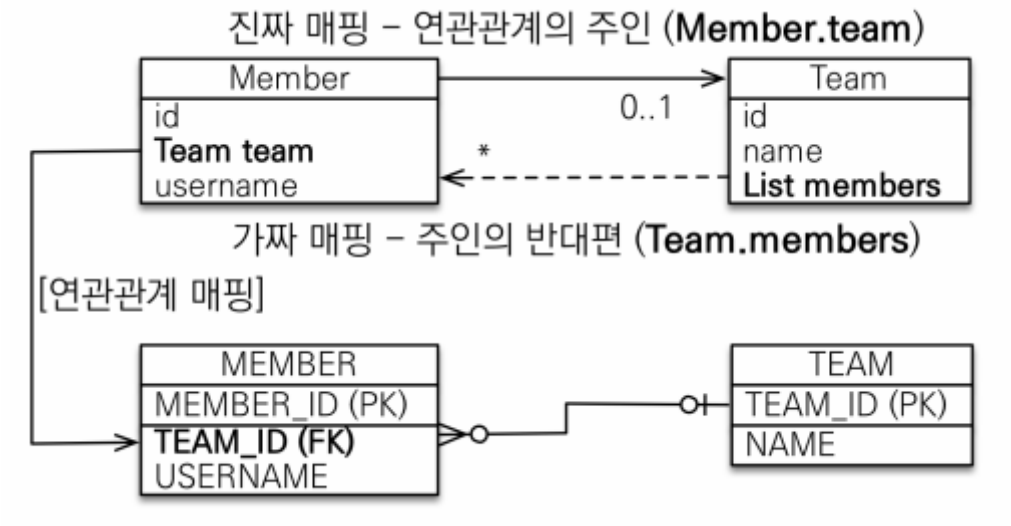
    @Temporal(TemporalType.TIMESTAMP)
    private Date lastModifiedDate;

    @Lob
    private String description;

    // 주로 메모리상에만 임시로 어떤 값을 저장할 때 사용 ---> DB에 반영 되지 않는다.
    @Transient
    private int tmp;
}
```

@Enumerate

- 기본값이 ORDINAL이다 : enum 순서를 데이터베이스에 저장한다.(0, 1, ,2 ...)
- **ORDINAL값으로 저장하게 되면 변경이 있을 때 그 전 데이터는 변하지 않기 때문에 큰 문제를 야기할 수 있다!!!**
- 거의 무조건 **EnumType.STRING (: enum 이름을 데이터베이스에 저장)**을 사용

기본 키 매핑

- **@Id**
- **@GeneratedValue(기본값: 자동생성)**

```
public class Member{

    @Id @GeneratedValue
    private Long id;
}
```

데이터 중심 설계의 문제점

- 객체 설계를 테이블 설계에 맞추게 되면 **테이블의 외래키를 객체에 그대로 가져오게 된다.**
- 객체 그래프 탐색이 불가능 하다.
- 참조가 없으므로 UML(통합모델링언어) 도 잘못되었다.

```
// order를 주문한 멤버를 찾고자 할때  ---> 객체지향적이지 않다.
Order order = em.find(Order.class, 1L);
Long memberId = order.getMemberId();

Member member = em.find(Member.class, memebrId);
```

연관관계 매핑 - 연관관계

- 방향 : 단방향, 양방향
- 다중성: 다대일(N:1), 일대다(1:N), 일대일(1:1), 다대다(N:M)
- 연관관계의 주인(Owner): 객체를 양방향 연관관계로 만들면 **연관관계의 주인을 정해야 한다.**

단방향 연관관계

연관관계가 없는 객체인 경우 -> 외래 키 식별자를 직접 다룬다.



객체 연관관계를 사용한 경우




```
// Member
@ManyToOne // 멤버 입장에서 봐야한다. 멤버가 N이고 One인 Team으로 매핑한다.
@JoinColumn(name = "TEAM_ID")
private Team team;
```

```
// 팀 저장
Team team = new Team();
team.setName("TeamA");
em.persist(team);

// 회원 저장
Member member = new Member();
member.setUsername("member1");
// JPA가 알아서 team에서 Pk값을 꺼내서 INSERT할때 FK으로 사용한다.
member.setTeam(team);
em.persist(member);
```

- @ManyToOne
  - 다대일(N:1) 관계라는 매핑 정보
  - 어노테이션 필수
- @JoinColumn(name="TEAM_ID")
  - 외래키(FK)를 매핑할 때 사용
  - N:1 일때 N쪽에 Foreign Key 가 들어간다.(Foreign에 n이 들어가니까 맞춰준다고 외워두자 👍)
  - name 속성에 매핑할 외래 키 이름을 지정한다.
  - 생략 가능하다.

양방향 연관관계

- **단방향 매핑만으로도 이미 연관관계 매핑은 완료된다.**
- 반대 방향으로 조회 기능이 추가되는것이다.
- **단방향 매핑을 하고 양방향은 필요할 때 추가해도 된다. (테이블에 영향을 주지 않는다!)**





- **테이블 연관관계**에서는 **FK**로 양쪽을 다 갈 수 있다. ---> 사실 방향이 없는거라고 볼 수 있다.
- **객체 연관관계**에서는 둘 다 세팅을 해줘야 양쪽을 볼 수있다. --->사실 단방향이 2개라고 볼 수 있다.
  - 회원 -> 팀 연관관계 1개 (단방향)
  - 팀 -> 회원 연관관계 1개 (단방향)

```
class A{
    B b;
}
================================================================================
=========
class B{
    A a;
}
```

👍연관관계의 주인(Owner)

- 객체의 두 관계중 하나를 연관관계의 주인으로 지정해야한다!!
- 연관관계의 주인만이 외래 키를 관리한다.(등록, 수정)
- 주인이 아닌쪽은 읽기만 가능하다.
- 주인이 아닌쪽은 mappedBy 속성으로 주인을 지정해줘야 한다.
- **✨ 항상 N쪽, 외래 키를 가진쪽이 주인이다. ✨**



양방향 매핑시 주의 점

- 연관관계의 주인에 값을 입력하지 않았을 때 ---> 값이 반영되지 않는다.(읽는 역할만 하기 때문)

```
Team team = new Team();
team.setName("TeamA");
em.persist(team);

Member member = new Member();
member.setName("member1");

// 역방향(주인이 아닌 방향)만 연관 관계 설정
team.getMember().add(member);

em.persist(member);
```

SELECT * FROM MEMBER ;

MEMBER_ID	USERNAME	TEAM_ID
1	member1	*null*

(1 row, 1 ms)

SELECT * FROM TEAM ;

TEAM_ID	NAME
2	TeamA

(1 row, 1 ms)

- 연관관계 주인에 값을 입력했을 때 ---> 순수한 객체 관계를 고려하면 항상 양쪽 다 값을 입력해야 한다.

```
Team team = new Team();
team.setName("TeamA");
em.persist(team);

Member member = new Member();
member.setName("member1");

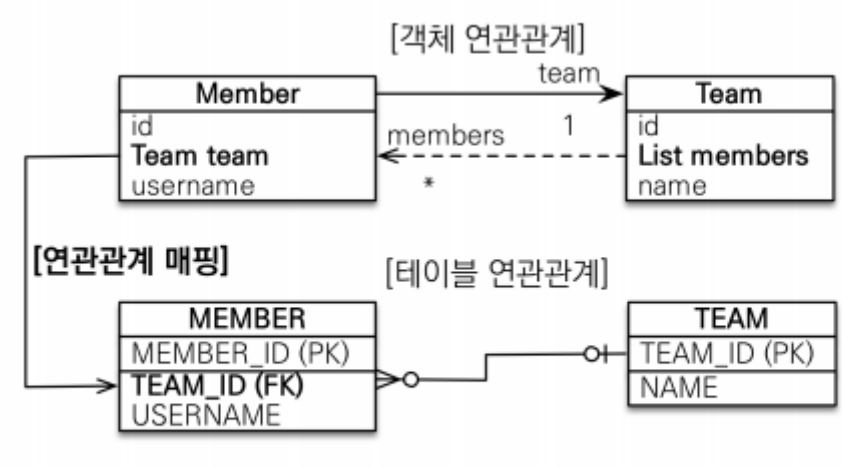
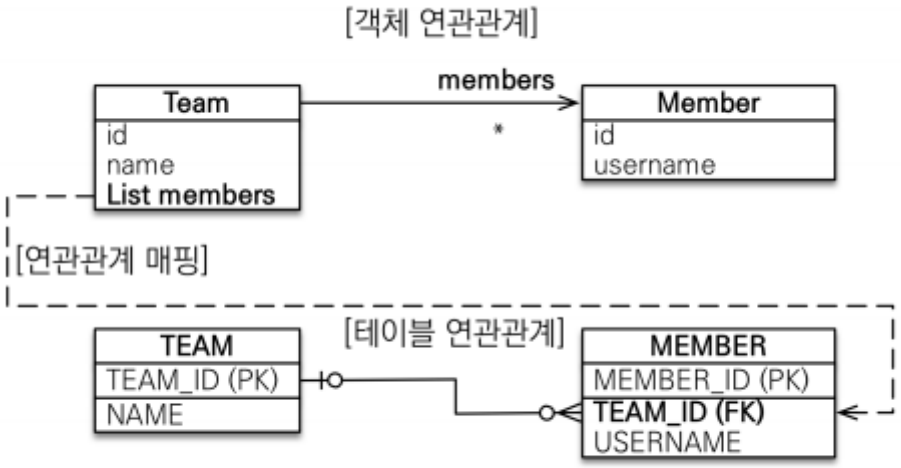
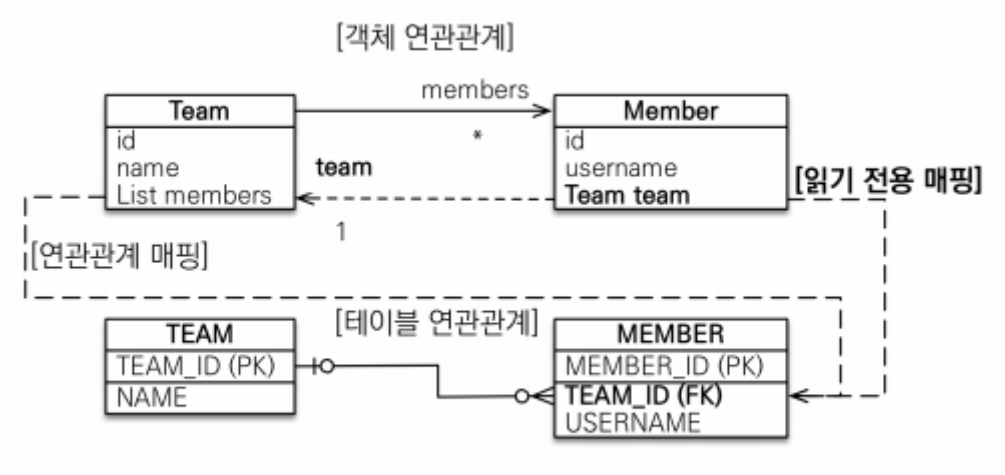
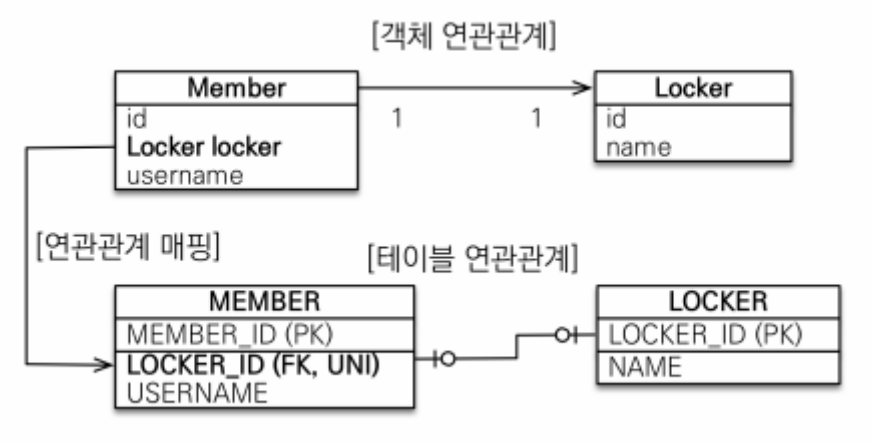
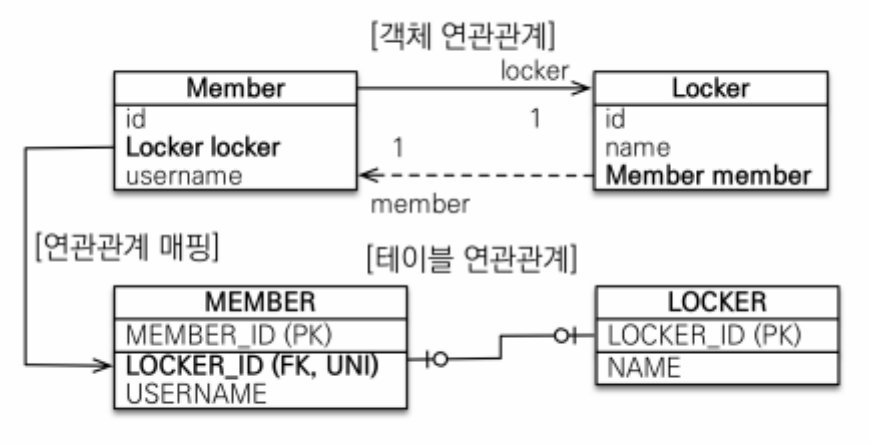
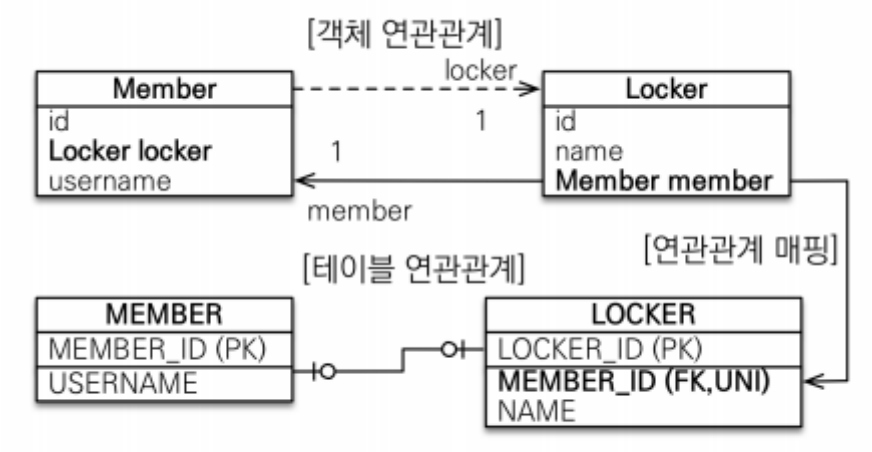
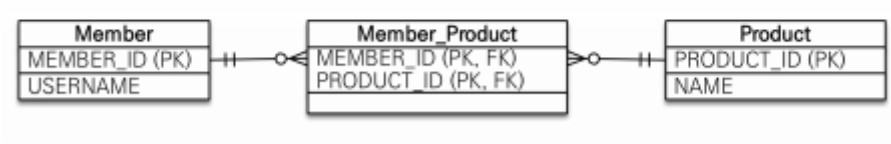
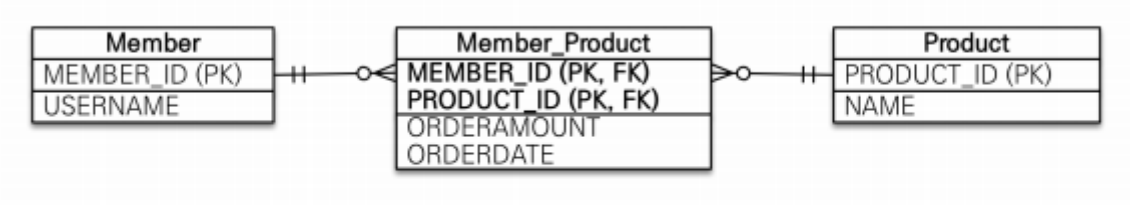
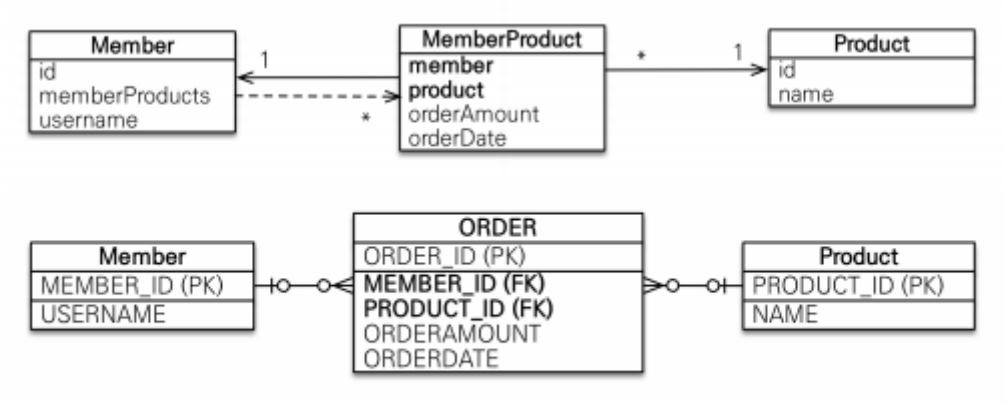
team.getMember().add(member);
// 연관관계의 주인에 값 설정
member.setTeam(team);

em.persist(member);
```

SELECT * FROM MEMBER ;

MEMBER_ID	USERNAME	TEAM_ID
2	member1	1

(1 row, 0 ms)

SELECT * FROM TEAM ;

TEAM_ID	NAME
1	TeamA

(1 row, 1 ms)

연관관계 편의 메소드 생성

- 순수 객체 상태를 고려해서 항상 양쪽에 값을 설정한다.
- 연관관계 편의 메소드를 생성하면 오류를 줄일 수 있다.

```
Team team = new Team();
team.setName("TeamA");
em.persist(team);

Member member = new Member();
member.setName("member1");

// 연관관계 편의 메소드를 생성
member.changeTeam(team);

em.persist(member);
```

```
public class Member{
    ....

    public void changeTeam(Team team){
        this.team = team;
        // team쪽에도 변경된 결과를 반영시켜준다.
        team.getMembers().add(this);
    }
}
```

연관관계 매핑 - 다중성

다대일[N:1]

단방향

- **가장 많이 사용하는 연관 관계**
- 

양방향

- 외래 키가 있는 쪽이 연관관계의 주인이다.
- 양쪽을 서로 참조하도록 개발

일대다[1:N]

N : 1 관계를 되도록이면 사용하도록 하자!!!

단방향

- 

- One(1)이 연관관계의 주인이다.
- 테이블 1 : N 관계는 항상 Many(N) 쪽에 외래 키가 있다.
- 객체와 테이블의 차이 때문에 반대편 테이블의 외래 키를 관리하는 이상한 구조이다.

  -> 연관관계 관리를 위해 추가로 UPDATE SQL을 실행한다.
- @JoinColumn을 꼭 사용해야 한다.

양방향

- 

- 공식적으로 존재하지 않는다.
- @JoinColumn(insertable = false, updateable = false) 를 사용해서 억지로 가능하다.

일대일[1: 1]

주 테이블이나 대상 테이블 중에 외래 키 선택 가능하다.

- 자주 사용되는 테이블을 주 테이블 개념으로 잡고가면 된다.

외래 키에 데이터베이스 유니크(UNI) 제약조건 추가

주 테이블에 외래 키 단방향

- 

주 테이블에 외래 키 양방향

- 

- **외래 키가 있는 곳이 연관관계의 주인이다.**
- 반대쪽은 mappedBy 적용

대상 테이블에 외래 키 양방향

- 

- 방법은 주 테이블 외래 키 양방향과 동일하다.

주 테이블 vs 대상 테이블 외래 키

-> 주 테이블에 외래 키를 사용하도록 하자

	주 테이블에 외래 키	대상 테이블에 외래 키
장점	주 테이블만 조회해도 대상 테이블에 데이터가 있는지 확인 가능	주 테이블과 대상 테이블을 일대일에서 일대다 관계로 변경할 때 테이블 구조 유지
단점	값이 없으면 외래 키에 null 허용	프록시 기능의 한계로 지연로딩으로 설정해도 항상 즉시 로딩된다.

다대다[N:M]

🔘 결론적으로 사용하지 말자

- 관계형 데이터베이스는 정규화된 테이블 2개로 다대다 관계를 표현할 수 없다

  --> 연결 테이블을 추가해서 일대다, 다대일 관계로 풀어내야 한다.



- 객체는 컬렉션을 사용해서 객체 2개로 다대다 관계가 가능하다.

- @ManyToMany 사용
- @JoinTable로 연결 테이블 지정

다대다 매핑의 한계



- 연결 테이블이 단순히 연결만 하고 끝나지 않는다.

  --> 조인 테이블에 주문시간, 주문수량 같은 추가 데이터가 들어갈 수 있다.
- 매핑 정보를 넣는 것은 가능하지만, 추가 정보를 넣는 것 자체가 불가능하다.
- 중간 테이블이 숨겨져 있기 때문에 예상하지 못하는 쿼리들이 나간다.
- **이런 문제들 때문에 실무에서는 사용하지 않는것을 권장한다!!**

다대다 한계 극복



- 연결 테이블용 엔티티를 추가한다(**연결 테이블을 엔티티로 승격**)

  -> JPA가 만들어주는 숨겨진 매핑테이블의 존재를 바깥으로 꺼내는 것이다.
- @ManyToMany --> @OneToMany & @ManyToOne 로 관계를 맺어준다.
- MemberProduct의 MEMBER_ID, PRODUCT_ID를 묶어서 PK로 사용하지 말고

  --> **ORDER_ID 를 PK로 독립적으로 generated되는 id를 사용하는 것을 권장한다.**

  --> ID가 2개의 테이블에 종속되지 않고 유연하게 개발이 가능하다.(PK를 운영중에 업데이트 하는 상황 대비)
- ```
  // Member
  @Entity
  public class Member {
      ...
      @OneToMany(mappedBy = "member")
      // 관례로 ArrayList로 초기화 시켜둔다. add 할때 null 이 안뜬다
      private List<MemberProduct> memberProducts = new ArrayList<>();
      ...
  }
  ```
  ```

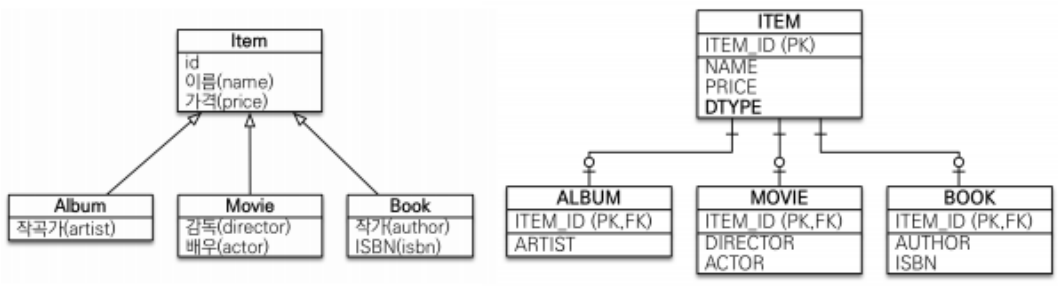
- ```java
  //Product
  @Entity
  public class Product {
      ...
      @OneToMany(mappedBy = "product")
      private List<MemberProduct> members = new ArrayList<>();
      ...
  }
  ```

- ```java
  //MemberProduct : 만약 추가 데이터가 들어가면 의미있는 엔티티 이름(Order)로 변경하면 된다.
  @Entity
  public class MemberProduct{
      @Id @GenratedValue
      private Long id;

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

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

고급 매핑

상속관계 매핑

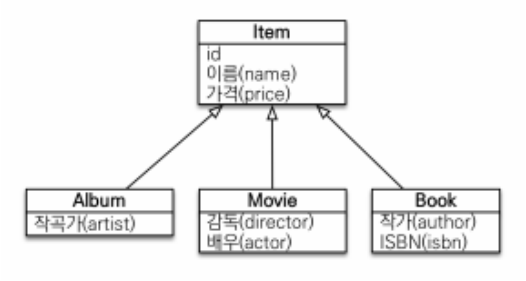
- 관계형 데이터베이스는 상속관계가 없다.
- 슈퍼타입 서브타입 관계라는 모델링 기법이 객체 상속과 유사하다.
- 슈퍼타입 서브타입 논리 모델을 실제 물리 모델로 구현하는 방법을 말한다.

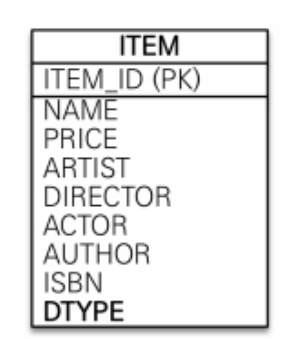
1. **조인 전략 - 각각 테이블로 변환**
   - @Inheritance(startegy = InheritanceType.JOINED)
   - 

   - 장점
     - 테이블 정규화
     - 외래 키 참조 무결성 제약조건 활용가능하다.
     - 저장 공간 효율화
   - 단점
     - 조회시 조인을 많이 사용 -> 성능 저하

- 조회 쿼리가 복잡하다.
- 데이터 저장시 INSERT SQL이 2번 호출 된다.

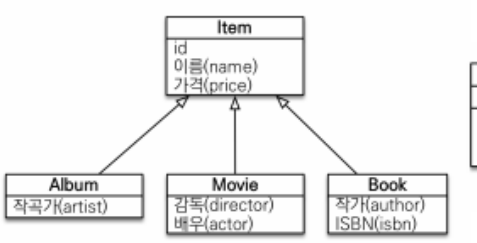
2. **단일 테이블 전략 - 통합 테이블로 변환**
   - @Inheritance(startegy = InheritanceType.SINGLE_TABLE)
   -

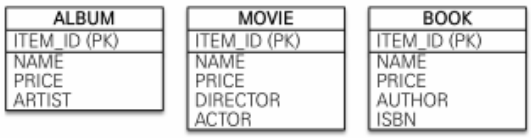




   - 장점
     - 조인이 필요없으므로 일반적으로 조회 성능이 빠르다.
     - 조회 쿼리가 단순함
   - 단점
     - 자식 엔티티가 매핑한 컬럼은 모두 null이 허용된다.
     - 단일 테이블에 모든것을 저장하므로 테이블이 커질 수 있다.

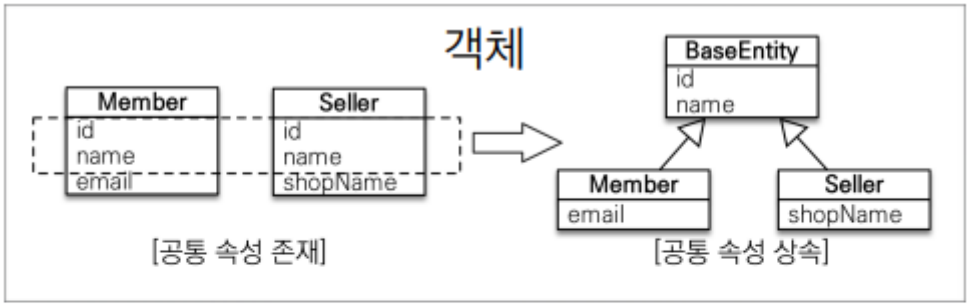
3. **구현 클래스마다 테이블 전략 - 서브타입 테이블로 변환**
   - @Inheritance(startegy = InheritanceType.TABLE_PER_CLASS)
   - 거의 사용하지 않는것을 추천
   -

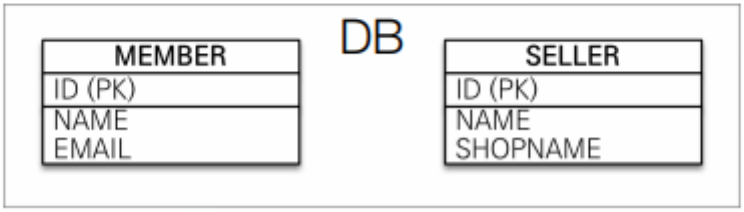




   - 장점
     - 서브 타입을 명확하게 구분해서 처리할 때 효과적
     - not null 제약조건을 사용 가능하다.
   - 단점
     - 여러 자식 테이블을 함께 조회할 때 성능이 느림(UNION SQL 필요)
     - 자식 테이블을 통합해서 쿼리하기가 어렵다.

@MappedSuperclass

- **공통 매핑 정보가 필요할 때 사용한다. --> 속성을 같이 쓰고 싶다!**
- 



- 상속관계 매핑이 아니다.
- **엔티티가 아니며, 테이블과 매핑되지 않는다**
- 부모 클래스를 상속 받는 **자식클래스에 매핑 정보만 제공한다.**
- em.find(BaseEntity) 가 불가능 하다.
- 직접 생성해서 사용할 일이 없으므로 **추상 클래스를 권장한다.**
- 주로 등록일, 수정일, 등록자, 수정자 같은 전체 엔티티에서 공통으로 적용하는 정보를 모을 때 사용한다.

```
@MappedSuperclass
public abstract class BaseEntity{

    private String createdBy;
    private LocalDateTime createdDate;
    private String lastModifiedBy;
    private LocalDateTime lastModifiedDate;
}
```

```
@Entity
public class Member extends BaseEntity{
    ...
}
```

```
@Entity
public class Team extends BaseEntity{
    ...
}
```

- BaseEntity에 선언된 칼럼들이 생성된다.

```
Hibernate:
    create table Member (
       id bigint generated by default as identity,
        createdBy varchar(255),
        createdDate timestamp,
```

```
        lastModifiedBy varchar(255),
        lastModifiedDate timestamp,
        age integer,
        description clob,
        roleType varchar(255),
        name varchar(255),
        locker_id bigint,
        team_id bigint,
        primary key (id)
    )

Hibernate:
    create table Team (
       id bigint generated by default as identity,
        createdBy varchar(255),
        createdDate timestamp,
        lastModifiedBy varchar(255),
        lastModifiedDate timestamp,
        name varchar(255),
        primary key (id)
    )
```

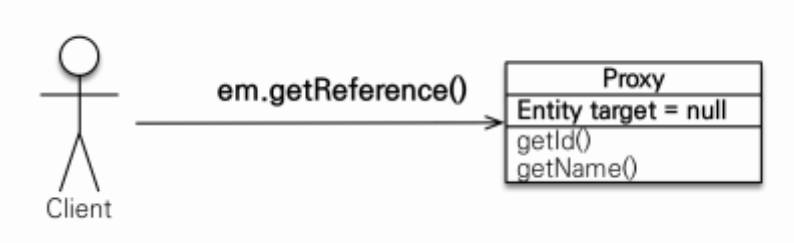
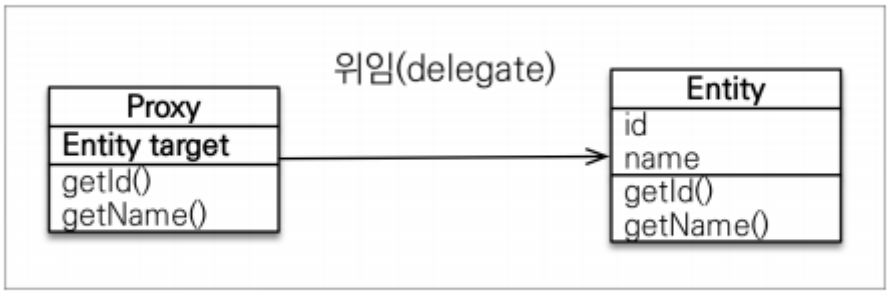
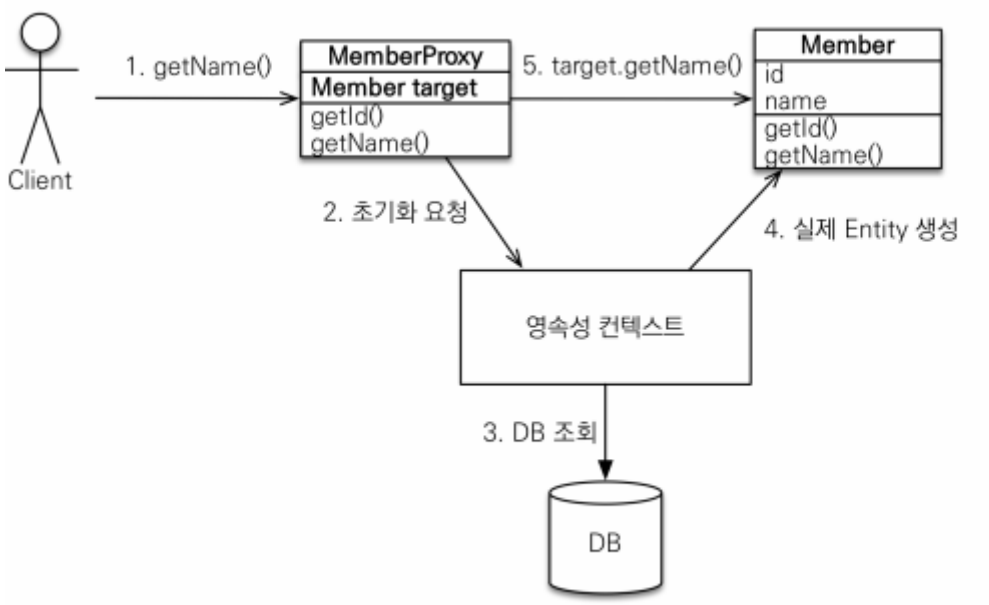
프록시와 연관관계 관리

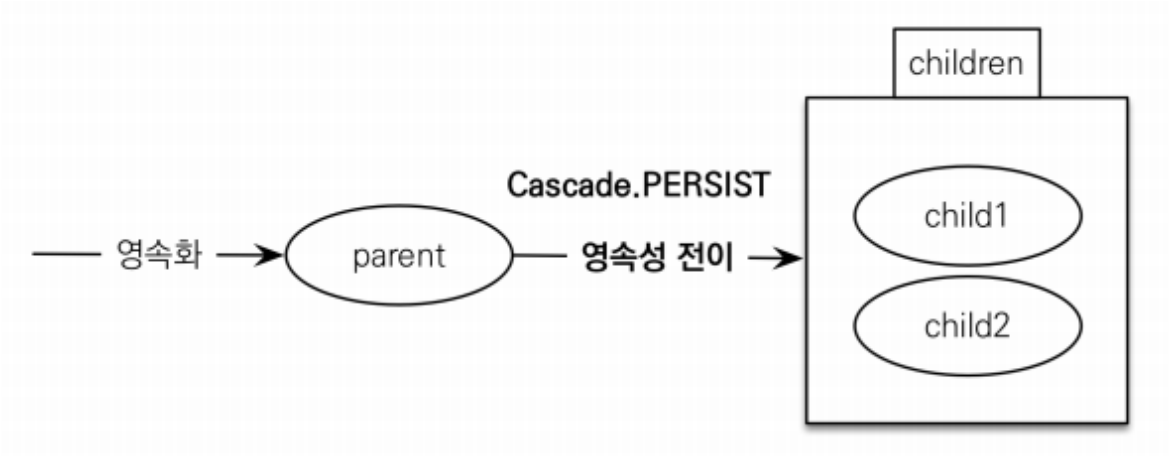
프록시를 사용하면 연관된 객체를 처음부터 조회하는 것이 아니라 실제 사용하는 시점에 DB에서 조회할 수 있다. 그러나 자주 사용하는 객체들은 조인을 사용해서 함께 조회하는 것이 효과적이다.

프록시를 사용해야하는 이유

- 회원정보만 사용하는 곳에서 연관된 팀 엔티티까지 DB에서 조회해 두는 것은 효율적이지 않다.
- JPA는 엔티티가 실제 사용될 때까지 DB조회를 지연하는 방법을 제공한다 -> **지연로딩**

프록시 객체 소개

- 지연 로딩을 사용하기 위해서는 "가짜 객체"가 필요하다.
- ```
  // 엔티티 직접 조회 - 영속성 컨텍스트에 없으면 DB조회
  Member member = em.find(Member.class, 100L);

  // 엔티티를 실제 사용하는 시점까지 미루는 프록시 객체
  Member member = em.getReference(Member.class, 100L);
  ```
- 
  ```

프록시 특징

- 실제 클래스를 상속받아서 만들어진다.
- 실제 클래스와 겉 모양이 같다. -> 사용하는 입장에선 구분하지 않고 사용하면 된다.
- 프록시 객체는 실제 객체의 참조(target)를 보관한다.
- 프록시 객체를 호출하면 프록시 객체는 실제 객체의 메소드를 호출한다.
- 

- 프록시 객체는 처음 사용할 때 한번만 초기화 된다.
- 프록시 객체가 초기화 되면 프록시 객체를 통해 실제 엔티티에 접근 가능하다.
- 프록시 객체는 원본 엔티티를 상속받음 -> 타입 체크시 주의해야한다(== 대신 instance of 사용해야 한다.)
- 영송석 컨텍스트에 찾는 엔티티가 이미 있으면 em.getReference()를 호출해도 **실제 엔티티를 반환** 한다.



즉시 로딩과 지연 로딩

지연 로딩(무조건 사용)

- LAZY를 사용해서 프록시로 조회

```
@Entity
public class Member{
    @Id
    @GeneratedValue
    private Long id;
```

```
    @Column(name = "USERNAME")
    private String name;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "TEAM_ID")
    private Team team;
    ...
}
```

- Member member = em.find(Member.class, 1L);

- Team team = member.getTeam();

  team.getName();

  실제 team을 사용하는 시점에 초기화(DB조회)

- 즉시 로딩을 적용하면 예상치 못한 SQL이 발생한다.
- 즉시 로딩은 JPQL에서 N+1문제를 일으킨다.

영속성 전이: CASCADE

특정 엔티티를 영속 상태로 만들 때 연관된 엔티티도 함께 영속 상태로 만들고 싶을 때 사용

예시) 부모 엔티티를 저장할 때 자식 엔티티도 함께 저장

영속성 전이: 저장

```
@Entity
public class Parent {
    ...
    @OneToMany(mappedBy = "parent", cascade = CascadeType.PERSIST)
    private List<Child> children = new ArrayList<Child>();
    ...
}
```



- 영속성 전이는 연관관계를 매핑하는 것과는 아무런 관련이 없다.
- 엔티티를 영속화 할 때 연관된 엔티티도 함께 영속화하는 편리함을 제공한다.
- **사용조건 1. 라이프 사이클이 동일 할때 2. 단일 소유자인 경우 사용하도록 하자**

고아 객체

- 고아 객체 제거: 부모 엔티티와 연관관계가 끊어진 자식 엔티티를 자동으로 삭제 해준다.
- **orphanRemoval = true 사용**
- ```
  @Entity
  public class Parent {

      @Id @GeneratedValue
      private Long id;

      @OneToMany(mappedBy = "parent", orphanRemoval = true)
      private List<Child> children = new ArrayList<Child>();
      ...
  }
  ```
  ```

- 참조하는 곳이 하나일 때 사용해야 한다!
- 특정 엔티티가 개인 소유할 때 사용해야 한다!
- @OneToOne, @OneToMany만 사용 가능하다.

값 타입

JPA 데이터 타입 분류

- 엔티티 타입
  - @Entity로 정의하는 객체
  - 데이터가 변해도 식별자로 지속해서 추적 가능
  - ex) 회원 엔티티의 키나 나이 값을 변경해도 식별자로 인식 가능
- 값 타입
  - int, Integer, String 처럼 단순히 값으로 사용하는 자바 기본 타입이나 객체
  - 식별자가 없고 값만 있으므로 변경시 추적이 불가능
  - ex) 숫자 100을 200으로 변경하면 완전히 다른 값으로 대체

기본값 타입

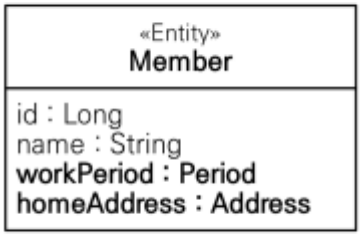
- ex) String name, int age
- 생명주기를 엔티티의 의존

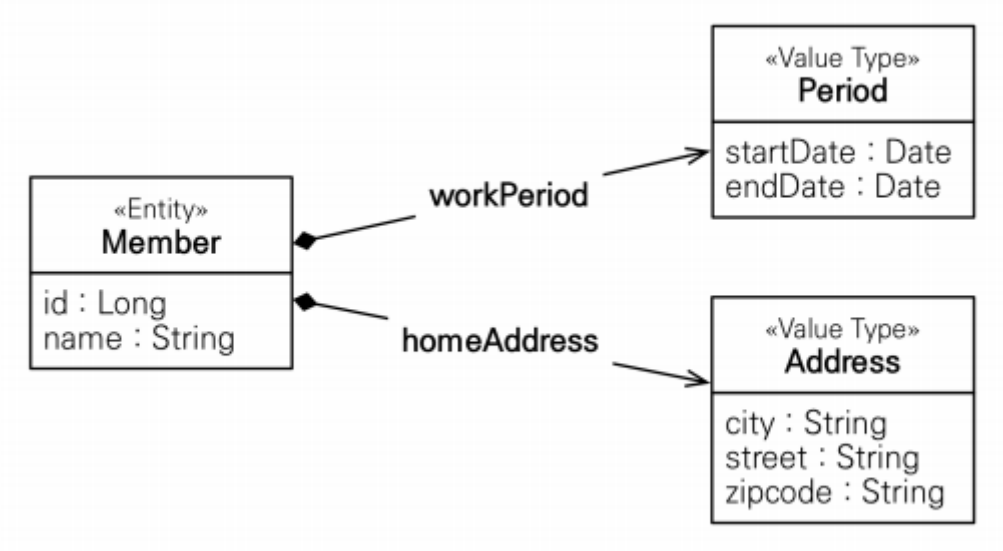
  ex) 회원을 삭제하면 이름, 나이 필드도 함께 삭제
- **값 타입은 공유하면 안된다.**

  ex) 회원 이름 변경시 다른 회원의 이름도 함께 변경되면 안된다.
- 항상 값을 복사한다.

임베디드 타입(복합 값 타입)

- 새로운 값 타입을 직접 정의할 수 있다.
- JPA는 임베디드 타입 이라 한다.
- 주로 기본 값 타입을 모아서 만들어서 복합 값 타입이라고도 한다.





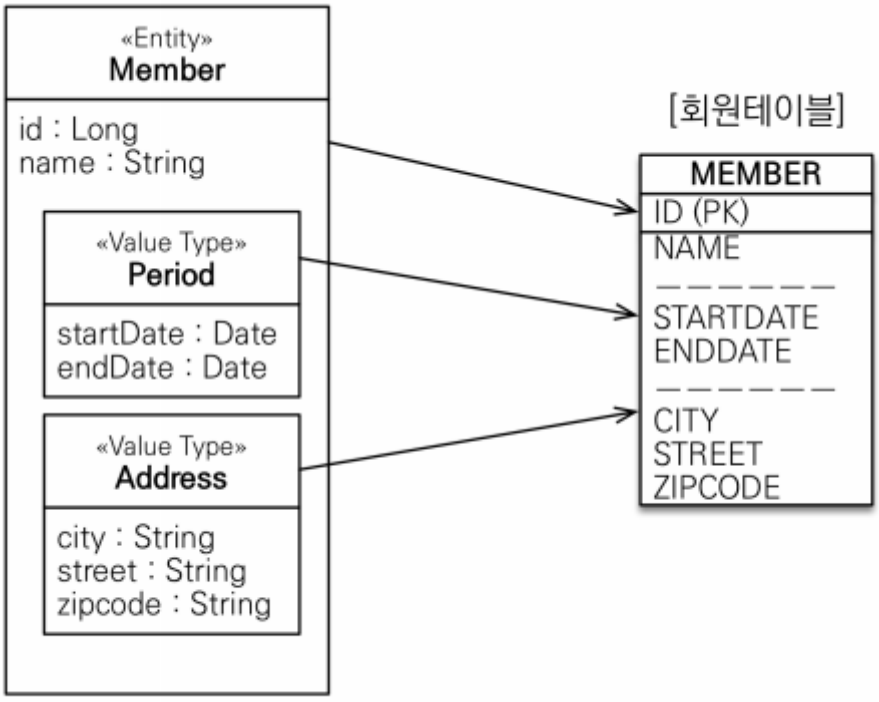
임베디드 타입 사용법

- @Embeddable: 값 타입을 정의하는 곳에 표시
- @Embedded: 값 타입을 사용하는 곳에 표시
- 기본 생성자는 필수이다

임베디드 타입의 장점

- 재사용
- 높은 응집도
- Period.isWork() 처럼 해당 값 타임만 사용하는 의미 있는 메소드를 만들 수 있다.
- **임베디드 타입을 포함한 모든 값 타입은, 값 타입을 소유한 엔티티에 생명주기를 의존한다.**

임베디드 타입과 테이블 매핑



- 임베디드 타입은 엔티티의 값일 뿐이다.
- 임베디드 타입을 사용하기 전과 후에 매핑하는 **테이블은 변함이 없다**
- 객체와 테이블을 아주 세밀하게 매핑하는 것이 가능하다.

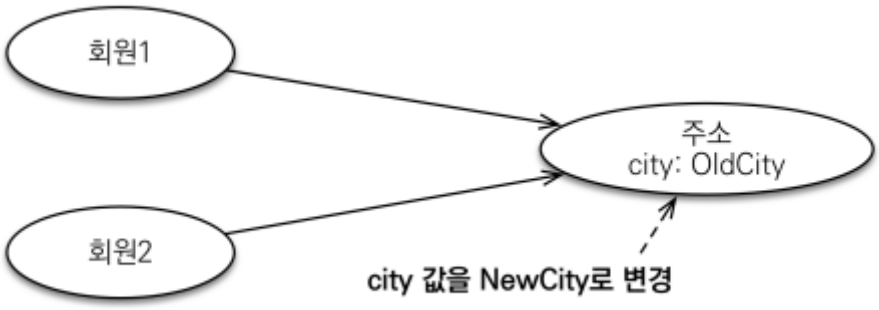
@AttributeOverride: 속성 재정의

- 한 엔티티에서 같은 값 타입을 사용하면 -> 컬럼 명이 중복된다.
- @AttributeOverrides, @AttributeOverride를 사용해서 컬럼 명 속성을 재정의

값 타입과 불변 객체

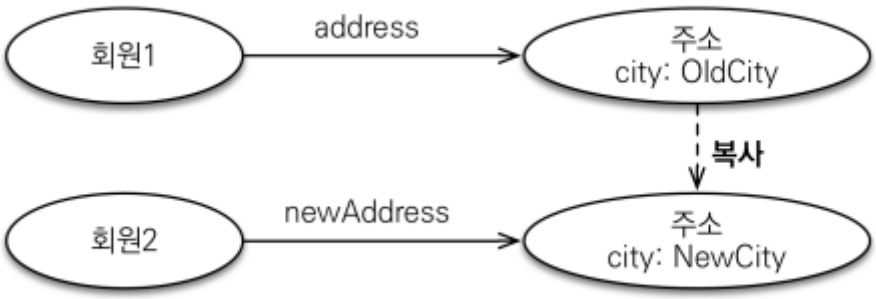
값 타입 공유 참조

- 임베디드 타입 같은 값 타입을 여러 엔티티에서 공유하면 위험하다 -> 부작용(Side Effect) 발생



값 타입 복사

- 값 타입의 실제 인스턴스 값을 공유하는 것은 위험하다.
- 대신 값(인스턴스)를 **복사해서 사용**



불변 객체

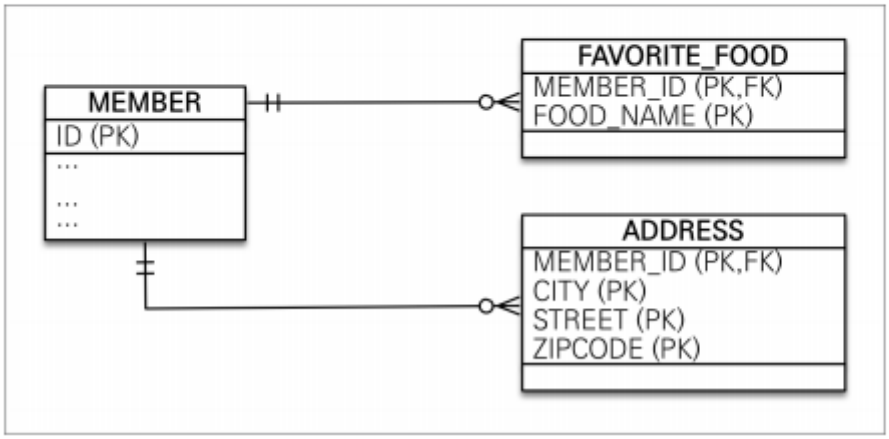
- 객체 타입을 수정할 수 없게 만들면 부작용을 원천 차단 할 수 있다
- 값 타입은 불변 객체(immutable object)로 설계해야 한다.
- **생성 시점 이후 절대 값을 변경 할 수 없는 객체**
- 생성자로만 값을 설정하고 수정자(Setter)를 만들지 않는다!

값 타입 컬렉션

정말 값 타입이라 판단 될때만 사용하도록 한다.

엔티티와 값 타입을 혼동해서 엔티티를 값 타입으로 만들면 안된다.

식별자가 필요하고, 지속해서 값을 추적, 변경을 해야 한다면 그것은 엔티티 이다.



- 값 타입을 하나 이상 저장할 때 사용한다.
- @ElementCollection, @CollectionTable사용
- 데이터베이스는 컬렉션을 같은 테이블에 저장할 수 없다.
- 컬렉션을 저장하기 위한 별도의 테이블이 필요하다.

값 타입 컬렉션의 제약사항

- 엔티티와 다르게 식별자 개념이 없다.
- 값은 변경하면 추적이 어렵다.
- 값 타입 컬렉션에 변경 사항이 발생하면, 주인 엔티티와 연관된 모든 데이터를 삭제하고, 현재값을 모두 다시 저장한다.
- 값 타입 컬렉션을 매핑하는 테이블은 모든 컬럼을 묶어서 기본키를 구성해야 한다.

  null 입력 X , 중복 저장 X

```
//값타입 수정

// homeCity --> newCity
// findMember.getHomeAddress().setCity("newCity"); 하면 큰일 난다. sideeffect유발

// 값 전체를 새로 넣어줘야한다.
Address a =findMember.getHomeAddress();
findMember.setHomeAddress(new Address("newCity", a.getStreet(),
a.getZipcode()));
```

```
// 치킨 -> 한식
findMember.getFavoriteFoods().remove("치킨");
findMember.getFavoriteFoods().add("한식");
```

```
// 관련된 모든 데이터를 삭제하고 남아있는 값을 다시 INSERT한다.
// 즉 전체를 지우고 남은 old2랑 newCity1 둘다 INSERT함
findMember.getAddressHistory().remove(new Address("old1", "street", "10000"));
findMember.getAddressHistory().add(new Address("newCity1", "street", "10000"));
```

값 타입 컬렉션 대안

- 값 타입 컬렉션 대신에 일대다 관계를 고려
- 일대다 관계를 위한 엔티티를 만들고, 여기에서 값 타입을 사용
- 영속성 전이(Cascade) + 고아 객체 제거를 사용해서 값 타입 컬렉션 처럼 사용