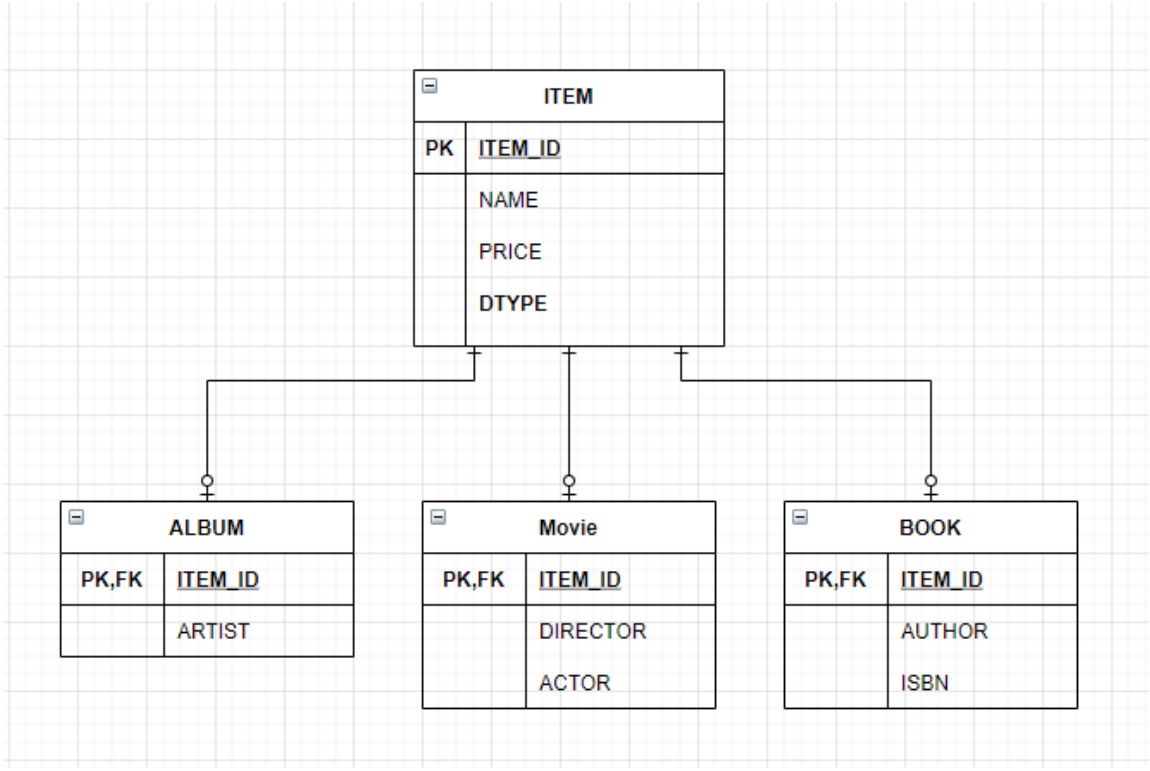
7장 고급 매핑

7장 고급 매핑

7.1 상속 관계 매핑

- 조인 전략



- 위 그림과 같이 엔티티 각각을 모두 테이블로 만들고 자식 테이블이 부모 테이블의 기본 키를 받아 기본 키 + 외래 키로 사용하는 전략 (조인을 자주 사용)
- 주의점: 테이블은 타입의 개념이 없어서 타입을 구분하는 컬럼을 추가 해야 한다! (위 그림에서는 DTYPE)

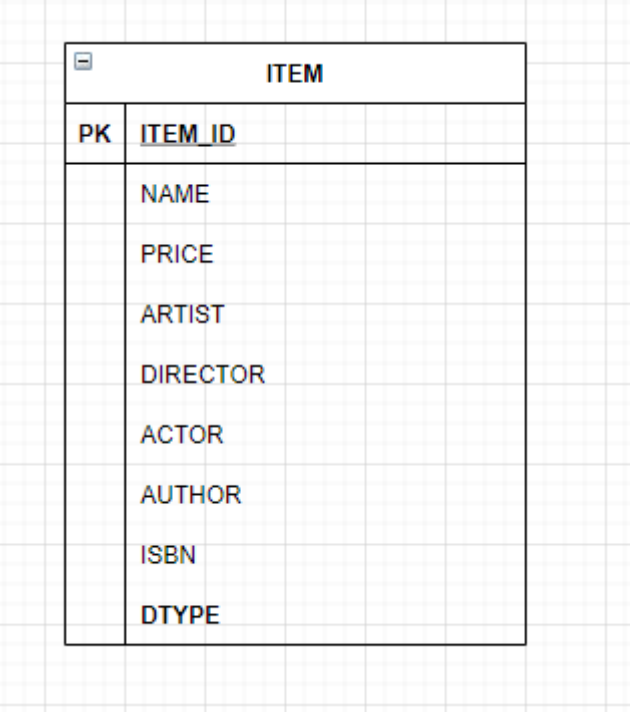
- 장점
  - 테이블이 정규화 된다.
  - 외래 키 참조 무결성 제약조건을 활용 할 수 있다.
  - 저장공간을 효율적으로 사용 한다.
- 단점
  - 조회할 때 조인이 많이 사용되어 성능이 저하될 수 있다.
  - 조회 쿼리가 복잡하다.
  - 데이터 등록 시 INSERT SQL 을 두 번 실행한다.(조인한 테이블에도 넣어줘야 되니까)

```
Hibernate:
    /* insert joinstrategy.Album
        */ insert
        into
            Item
            (ITEM_ID, name, price, DTYPE)
        values
            (null, ?, ?, 'A')
Hibernate:
    /* insert joinstrategy.Album
        */ insert
        into
            Album
            (artist, ITEM_ID)
        values
            (?, ?)
```

- 특징
  - 하이버네이트를 포함한 몇몇 구현체는 구분컬럼(@DiscriminatetorColumn) 없이도 동작
  - @PrimaryKeyJoinColumn(name = "BOOK_ID") 의 경우 autoDDL 기능을 켜둬야 동작

```
Hibernate:
    create table Book (
        author varchar(255),
        isbn varchar(255),
        BOOK_ID bigint not null,
        primary key (BOOK_ID)
    )
Hibernate:
    create table Item (
        DTYPE varchar(31) not null,
        ITEM_ID bigint generated by default as identity,
        name varchar(255),
        price integer not null,
        primary key (ITEM_ID)
    )
```

- 단일 테이블 전략



- 테이블을 하나만 사용하는 전략

- 구분 컬럼(DTYPE) 으로 어떤 자식 데이터가 저장되었는지 구분하며, 조회 시 조인을 사용하지 않아 일반적으로 가장 빠르다.
- 주의점 : 자식 엔티티가 매핑한 컬럼은 모두 null 을 허용 해야 한다!
  - 예를들어 BOOK 엔티티를 저장 시 ITEM 테이블의 AUTHOR, ISBN 컬럼만 사용 되고 다른 엔티티와 매핑된 컬럼은 사용하지 않으므로 null 이 입력되기 때문

SELECT * FROM ITEM;

DTYPE	ITEM_ID	NAME	PRICE	ARTIST	AUTHOR	ISBN	ACTOR	DIRECTOR
A	1	쑨앨범	10000	Soon 아티스트	*null*	*null*	*null*	*null*

(1 row, 2 ms)

- 위 설명한 조인전략 코드와 @Inheritance(strategy = InheritanceType.SINGLE_TABLE) 을 제외 하고 구성이 같으므로 생략
- 장점
  - 조인이 필요 없으므로 일반적으로 조회 성능이 빠름
  - 조회 쿼리가 단순
- 단점
  - 자식 엔티티가 매핑한 컬럼은 모두 null 허용 해야 함
  - 단일 테이블에 모든 것을 저장하므로 테이블이 커질 수 있다. 그러므로 상황에 따라서는 오히려 조인전략보다 성능이 느려질 수 있다.
- 특징
  - 구분 컬럼(@DiscriminatorColumn)을 꼭 사용 해야 함.(테이블 하나에 다 넣으니까..!)
    - 근데 웃긴건 안써도 default 로 JPA가 만들어 준다.

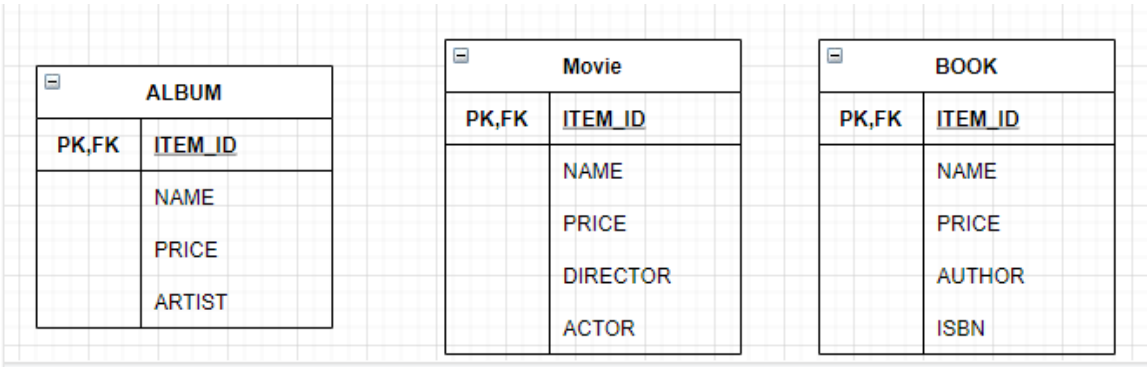
```java
//부모 클래스에 구분 컬럼을 지정 한다
//@DiscriminatorColumn(name = "DTYPE")
@Getter
@Setter
@NoArgsConstructor
@SuperBuilder
public abstract class Item {
```

```
Tests passed: 1 of 1 test – 764 ms

WARN: HHH000137: Root entity should not hold an PrimaryKeyJoinColum(s), will be ignored
7월 11, 2021 6:03:07 오후 org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory <init>
INFO: HHH000397: Using ASTQueryTranslatorFactory
7월 11, 2021 6:03:07 오후 org.hibernate.tool.hbm2ddl.SchemaExport execute
INFO: HHH000227: Running hbm2ddl schema export
Hibernate:
    drop table Item if exists
Hibernate:
    create table Item (
        DTYPE varchar(31) not null,
        ITEM_ID bigint generated by default as identity,
        name varchar(255),
        price integer not null,
        artist varchar(255),
        author varchar(255),
        isbn varchar(255),
        actor varchar(255),
        director varchar(255),
        primary key (ITEM_ID)
    )
```

- @DiscriminatorValue 미지정 시 기본 값인 엔티티 이름을 사용

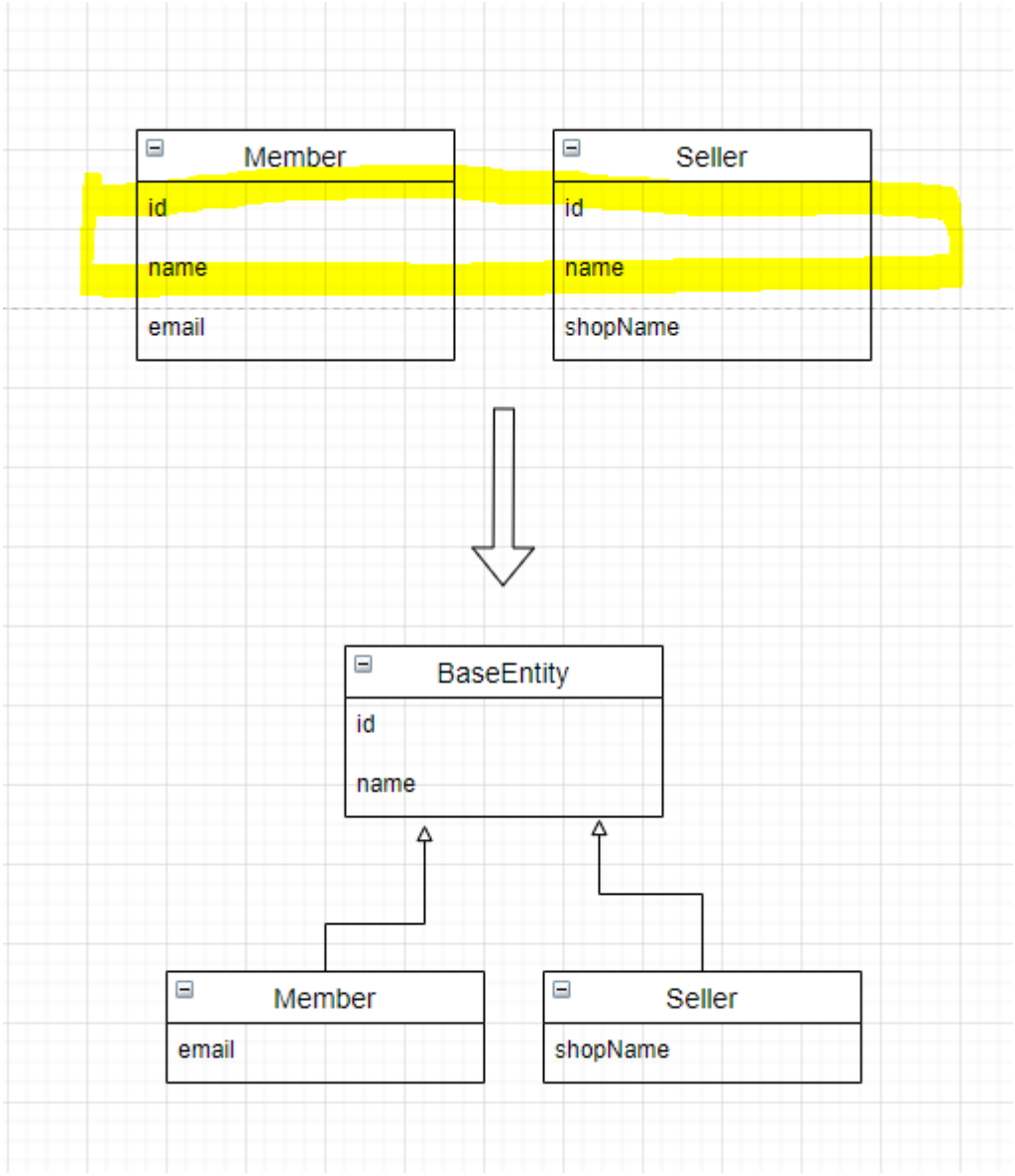
- 구현 클래스마다 테이블 전략



- 위 전략들과 마찬가지로 @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) 설정 제외하고서 코드 동일
- 이 전략은 자식 엔티티마다 테이블을 만든다. 추천하지 않는 전략임
- 장점
  - 서브 타입을 구분해서 처리 할 때 효과 적
  - NOT NULL 제약 조건을 이용 할 수 있다.
- 단점
  - 여러 자식 테이블을 함께 조회할 때 성능이 느리다. (SQL UNION을 사용 해야 한다. 겹치는게 없을테니)
  - 자식 테이블을 통합해서 쿼리하기 어렵다.
- 특징
  - 구분 컬럼을 사용하지 않음
- 이 전략은 DB 설계자, ORM 전문가 둘 다 추천하지 않는 전략이라고 한다. 조인이나 단일 테이블 사용하는걸로..!

7.2 @MappedSuperclass

- 부모 클래스는 테이블과 매핑하지 않고 부모 클래스를 상속받는 자식 클래스에게 매핑 정보만 제공 하고 싶을 경우 사용
- @Entity 는 실제 테이블과 매핑되지만 @MappedSuperclass는 실제 테이블과 매핑 되진 않는다!
- 매핑정보를 상속할 목적으로만 사용됨 (추상클래스와 비슷)
-



-

```
@AttributeOverride :부모에게 물려받은 매핑 정보 재정의
@AttributeOverrides : AttributeOverride둘 이상 재정의
```

- 특징
  - 테이블과 매핑되지 않고 자식 클래스에 엔티티 매핑 정보를 상속 하기 위해 사용
  - @MappedSuperclass로 지정한 클래스는 엔티티가 아니므로! em.find() 나 JPQL 에서 사용 할 수 없다.
    - 데이터베이스 테이블 구조가 바뀐게 아님! 객체에서만 상속모델을 따랐을 뿐!

SELECT * FROM SELLER;

ID	NAME	SHOPNAME
1	순명	soonShop

(1 row, 1 ms)

SELECT * FROM MEMBER;

MEMBER_ID	MEMBER_NAME	EMAIL
1	순명	soonworld91

(1 row, 1 ms)

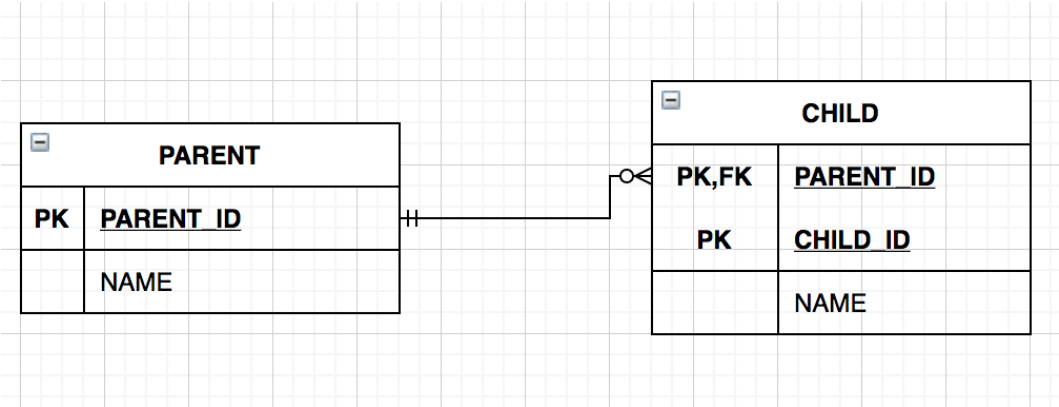
- 이 클래스를 직접 생성 할 일은 거의 없으므로 추상 클래스로 만드는걸 권장

- Member 테이블 만들 시 id,와 name을 MEMBER_ID ,MEMBER_NAME 으로 생성

```
Hibernate:
    create table Member (
        MEMBER_ID bigint generated by default as identity,
        MEMBER_NAME varchar(255),
        email varchar(255),
        primary key (MEMBER_ID)
    )
Hibernate:
    create table Seller (
        id bigint generated by default as identity,
        name varchar(255),
        shopName varchar(255),
        primary key (id)
    )
```

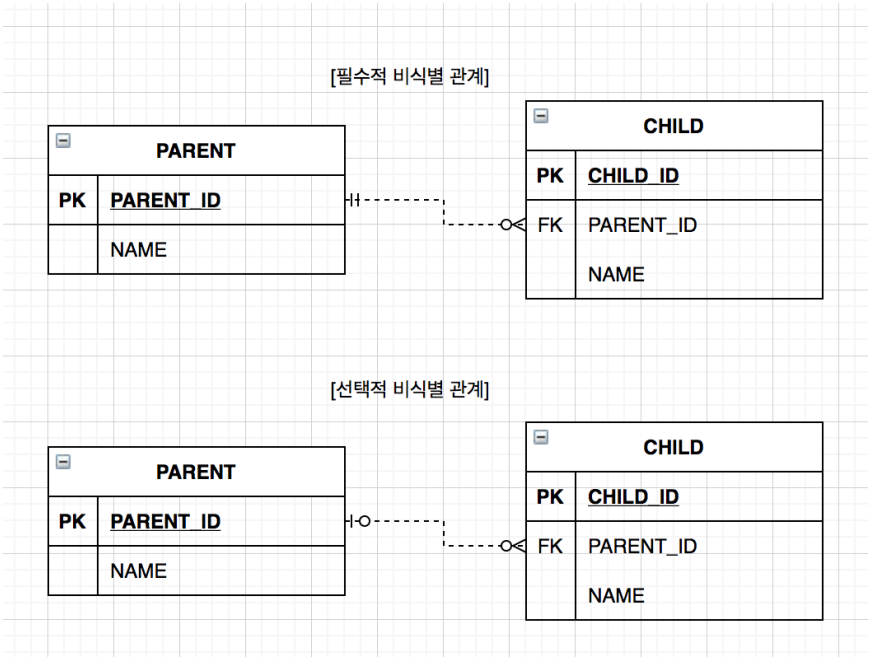
7.3 복합 키와 식별 관계 매핑

- 식별 관계



- 부모 테이블의 PARENT_ID 를 받아 자식 테이블의 PK+FK 로 사용

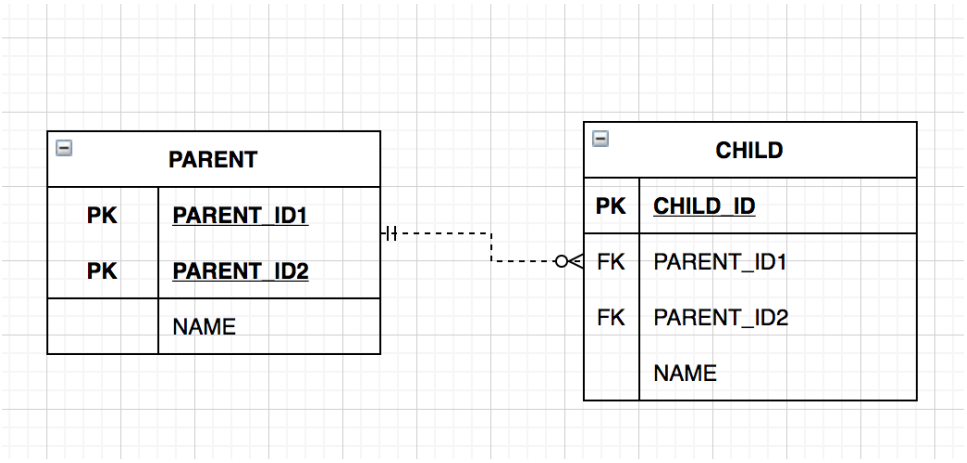
- 비식별 관계



  - 필수적 비식별 관계
    - 외래 키에 NULL을 허용하지 않음, 연관관계를 필수로 맺어야 한다.
  - 선택적 비식별 관계
    - 외래 키에 NULL을 허용, 연관관계를 맺을지 말지 선택 할 수 있다.

  - 최근에는 비식별관계를 주로 사용하고 꼭 필요한 곳에만 식별 관계를 사용하는 추세

- 복합 키 : 비식별 관계 매핑
  - 둘 이상의 식별자를 이용하려면 별도의 식별자 클래스를 만들어야 한다. (6장에 있었던 내용)
    - 복합 키 말고 새로운 기본키 전략(Long Type) 쓰는게 더 편리한걸로 정리했었는데?! 나 이거 안쓸껀데??! → 투입된 프로젝트에서 이미 복합키로 구성 되있으면?
  - @IdClass (예시로 알아보겠습니다.)



[복합 키 테이블]

```
[부모 클래스]
@Entity
@IdClass(ParentId.class)
public class Parent {

  @Id
  @Column(name = "PARENT_ID1")
  private String id1; //ParentId.id1 과 연결

  @Id
  @Column(name = "PARENT_ID2")
  private String id2; //ParentId.id2 와 연결

  private String name;
  ...
}
```

```
[식별자 클래스]

public class ParentId implements Serializable {
```

```
  private String id1; //Parent.id1 매핑
  private String id2; //Parent.id2 매핑

  public ParentId(){}
  public ParentId(String id1, String id2) {
    this.id1 = id1;
    this.id2 = id2;
  }

  @Override
  public boolean equals(Object o) {...}

  @Override
  public int hashCode() {...}

}
```

- 식별자 클래스의 조건
  - 식별자 클래스의 속성명 == 엔티티에서 사용하는 식별자의 속성명
  - Serializable 인터페이스 구현
  - equals, hashCode 구현
  - 기본 생성자
  - 식별자 클래스 범위는 public

- 복합키를 사용하는 엔티티 저장 예시

```
Parent parent = new Parent();
parent.setId1("myId1");
parent.setId2("myId2");
parent.setName("SOONPARENT");
em.persist(parent);
```

- 식별자 클래스인 ParnetId의 경우 영속성 컨텍스트에 엔티티를 등록하기 직전에 내부에서 생성 한 뒤 컨텍스트의 키로 사용

- 복합키 조회 예시

```
ParentId parentId = new ParentId("myId1", "myId2");
Parent parent = em.find(parent.class, parentId);
```

- @EmbeddedId
  - @IdClass 가 데이터베이스에 맞춘 방법이라면 @EmbeddedId 는 좀 더 객체지향적인 방법 이다.

```
@Entity
public class Parent {
  @EmbeddedId
  private ParentId id; //식별자 클래스 직접 지정

  private String name;
  ...
}

@Embeddable
public class ParentId implements Serializable {

  @Column(name = "PARENT_ID1")
  private String id1;

  @Column(name = "PARENT_ID2")
  private String id2;

  //equals and hashcode 구현
  ...
}
```

  - @EmbeddedId 적용한 식별자 클래스 조건
    - @Embeddable 어노테이션 붙여야 함
    - Serializable 인터페이스 구현
    - equals, hashCode 구현
    - 기본 생성자
    - public
- 복합키: 식별 관계 매핑
  - 식별 관계에서의 자식 테이블은 부모 테이블의 기본 키를 포함하여 복합 키를 구성해야 하므로 @IdClass 나 @EmbeddedId 를 사용해서 식별자를 매핑해야 한다.

- 식별, 비식별 관계의 장단점

- DB 설계 관점에서는 식별 관계보다는 비 식별관계를 선호 한다. 그 이유는?
  - 식별관계는 부모 테이블의 기본 키를 자식 테이블로 계속 전파 하면서 자식 테이블의 기본 키 컬럼이 점점 늘어나게 된다.
  - 결국 조인할 때 SQL 이 복잡해지고 기본 키 인덱스가 불필요하게 커질 수 있다.
  - 식별 관계는 2개이상의 컬럼을 합하여 복합 기본 키를 만들어야 하는 경우가 많다!
  - 식별 관계의 자연 키 컬럼들이 자식에 손자까지 전파되면 변경하기 힘듬
- 객체 관계 매핑 관점에서 비식별 관계를 선호 하는 이유는?
  - 6장에서 배운 대리 키 생성하여 처리하는 편리한 방법이 있으니까
- 정리 해보면 ORM 신규 프로젝트 진행 시 될 수 있으면 비식별 관계를 사용하고 기본 키는 Long 타입의 대리 키를 사용 하자

7.4 조인 테이블

- 조인 컬럼 사용 (외래 키)
  - 단순히 외래 키 컬럼만 추가해서 연관관계를 맺음
- 조인 테이블 사용(테이블)
  - 연관관계 관리하는 조인 테이블 추가하며, 이 테이블은 조인하는 두 테이블의 외래 키를 가지고 연관관계를 관리 함
  - 단점은 테이블을 하나 추가 해야 되는 점 (관리 포인트 늘어남)
  - 일대일 조인 테이블
    - 외래 키 컬럼 각각에 총 2개의 유니크 제약 조건 걸어야 함
  - 일대다 조인 테이블
    - 다(N)와 관련된 컬럼에 유니크 제약 조건을 걸어야 함
  - 다대일 조인 테이블
    - 일대다와 방향만 반대임 (똑같이 다(N)와 관련된 컬럼에 유니크 제약조건 걸어야 함)
  - 다대다 조인 테이블
    - 조인 테이블의 두 컬럼을 하나의 복합 유니크 제약조건을 걸어야 함

7.5 엔티티 하나에 여러 테이블 매핑

- @SecondaryTable 을 사용하여 한 엔티티에 여러 테이블 매핑 가능
- 권장하지는 않음 (이 방법은 항상 두 테이블을 조회하므로 최적화하기가 어렵다, 차라리 테이블 엔티티를 각각 만들어 일대일 매핑하는걸 권장)

7.6 정리

- 7.1장과 7.2장의 내용이 핵심이라고 볼 수 있다.