- **연관관계 매핑 - 심화2, 복합키와 식별 관계**
  - **식별 관계 vs 비식별 관계**
    - - 식별 관계
    - - 비식별 관계
  - **복합 키 - 비식별 관계 매핑**
    - - @IdClass
    - - @EmbeddedId
    - - 복합 키는 equals와 hashCode를 필수로 구현해야 한다.
    - - 정리
  - **복합 키 - 식별 관계 매핑**
    - - @IdClass와 식별 관계
    - - @EmbeddedId와 식별 관계
  - **비식별 관계로 구현**
  - **일대일 식별 관계**
  - **식별, 비식별 관계의 장단점**
  - **조인 테이블**
    - - 일대일 조인테이블
    - - 일대다 조인테이블
    - - 다대일 조인테이블
    - - 다대다 조인테이블
  - **엔티티 하나에 여러 테이블 매핑하기**

연관관계 매핑 - 심화2, 복합키와 식별 관계

식별 관계 vs 비식별 관계

데이터베이스 테이블 사이에 관계는 외래 키가 기본 키에 포함되는지 여부에 따라 식별 관계와 비식별 관계로 구분한다. 두 관계의 특징을 이해하고 각각을 어떻게 매핑하는지 알아보자.

- 식별 관계(Identifying Relationship)
- 비식별 관계(Non-Identifying Relationship)

- 식별 관계

식별 관계는 부모 테이블의 기본 키를 내려 받아서 자식 테이블의 기본 키 + 외래 키로 사용하는 관계다.

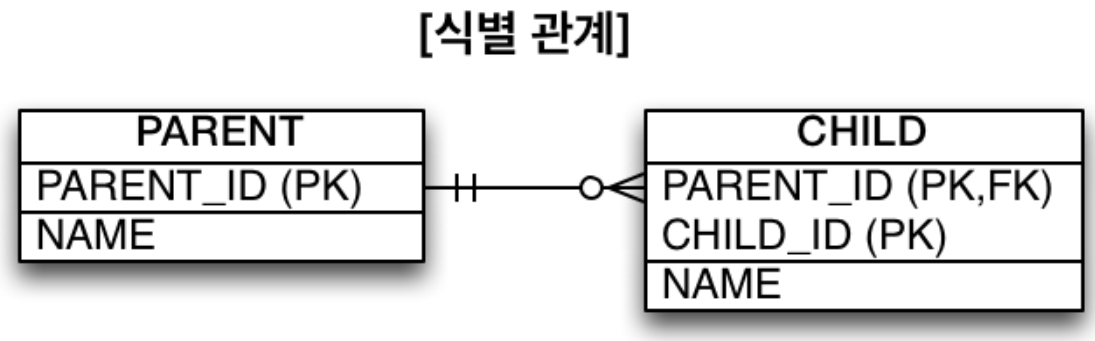

그림 - 식별 관계

[그림 - 식별 관계]를 보면 `PARENT` 테이블의 기본 키 `PARENT_ID`를 받아서 `CHILD` 테이블의 기본 키(PK) + 외래 키(FK)로 사용한다.

- 비식별 관계

비식별 관계는 부모 테이블의 기본 키를 받아서 자식 테이블의 외래 키로만 사용하는 관계다.

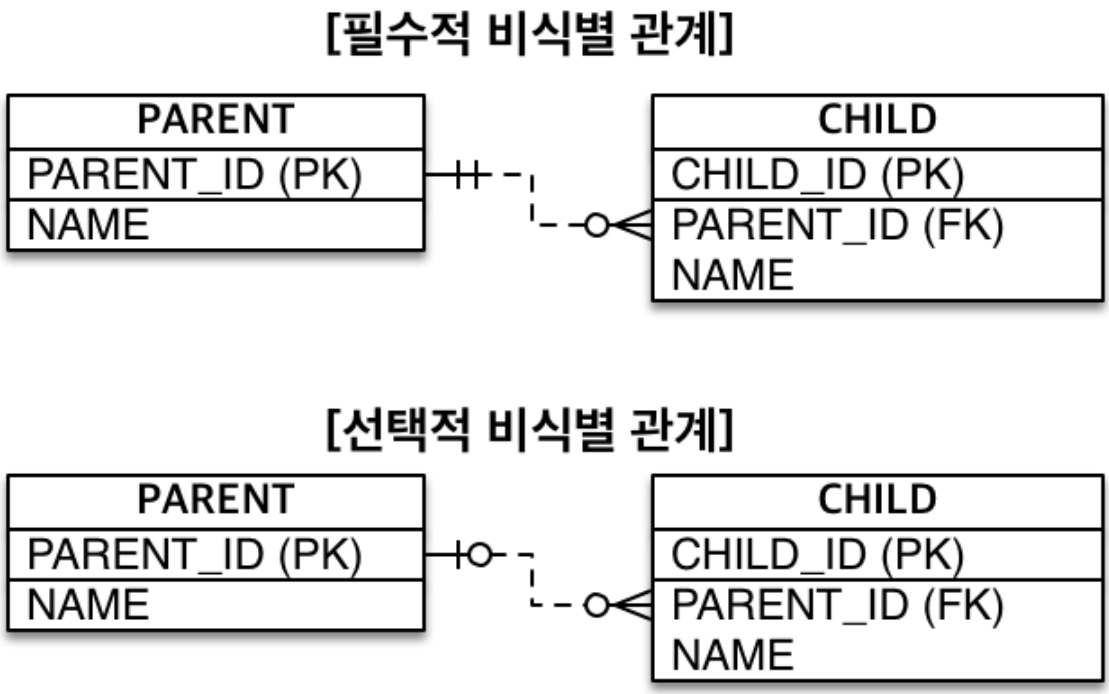

그림 - 비식별 관계

[그림 - 비식별 관계]를 보면 `PARENT` 테이블의 기본 키 `PARENT_ID`를 받아서 `CHILD` 테이블의 외래 키(FK)로만 사용한다.

비식별 관계는 외래 키에 `NULL`을 허용하는지에 따라 필수적 비식별 관계와 선택적 비식별 관계로 나눈다.

- 필수적 비식별 관계(Mandatory) : 외래 키에 `NULL`을 허용하지 않는다. 연관관계를 필수적으로 맺어야 한다.
- 선택적 비식별 관계(Optional) : 외래 키에 `NULL`을 허용한다. 연관관계를 맺을지 말지 선택할 수 있다.

데이터베이스 테이블을 설계할 때 식별 관계나 비식별 관계 중 하나를 선택해야 한다. 최근에는 비식별 관계를 주로 사용하고 꼭 필요한 곳에만 식별 관계를 사용하는 추세다. JPA는 식별 관계와 비식별 관계를 모두 지원한다.

식별 관계와 비식별 관계를 어떻게 매핑하는지 알아보자. 우선 복합 키를 사용하는 비식별 관계부터 보자.

복합 키 - 비식별 관계 매핑

기본 키를 구성하는 컬럼이 하나면 다음처럼 단순하게 매핑한다.

```
@Entity
public class Hello {
    @Id
    private String id;
}
```

둘 이상의 컬럼으로 구성된 복합 기본 키는 다음처럼 매핑하면 될 것 같지만 막상 해보면 매핑 오류가 발생한다. JPA에서 식별자를 둘 이상 사용하려면 별도의 식별자 클래스를 만들어야 한다.

```
@Entity
public class Hello {
    @Id
    private String id1;
    @Id
    private String id2; //실행 시점에 매핑 예외 발생
}
```

JPA는 영속성 컨텍스트에 엔티티를 보관할 때 엔티티의 식별자를 키로 사용한다. 그리고 식별자를 구분하기 위해 `equals`와 `hashCode`를 사용해서 동등성 비교를 한다. 그런데 식별자 필드가 하나일 때는 보통 자바의 기본 타입을 사용하므로 문제가 없지만, 식별자 필드가 2개 이상이면 별도의 식별자 클래스를 만들고 그곳에 `equals`와 `hashCode`를 구현해야 한다.

JPA는 복합 키를 지원하기 위해 `@IdClass`와 `@EmbeddedId` 2가지 방법을 제공하는데 `@IdClass`는 관계형 데이터베이스에 가까운 방법이고 `@EmbeddedId`는 좀 더 객체 지향에 가까운 방법이다.

먼저 `@IdClass`부터 알아보자.

- @IdClass

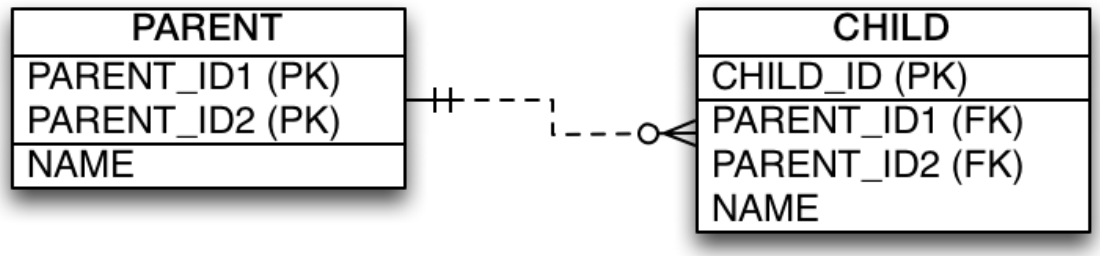

그림 - 복합 키 테이블

[그림 - 복합 키 테이블]은 비식별 관계고 `PARENT`는 복합 기본 키를 사용한다.

`PARENT` 테이블을 보면 기본 키를 `PARENT_ID1` , `PARENT_ID2` 로 묶은 복합 키로 구성했다. 따라서 복합 키를 매핑하기 위해 식별자 클래스를 별도로 만들어야 한다.

==== 부모 클래스 ====

```
@Entity
@IdClass(ParentId.class)
public class Parent {

    @Id
    @Column(name = "PARENT_ID1")
    private String id1; //ParentId.id1과 연결

    @Id
    @Column(name = "PARENT_ID2")
    private String id2;  //ParentId.id2와 연결

    private String name;
    ...
}
```

`PARENT` 테이블을 매핑한 `Parent` 클래스를 분석해보자.

먼저 각각의 기본 키 컬럼을 `@Id` 로 매핑했다. 그리고 `@IdClass` 를 사용해서 `ParentId` 클래스를 식별자 클래스로 지정했다.

==== 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() {...}
}
```

`@IdClass`를 사용할 때 식별자 클래스는 다음 조건을 만족해야 한다.

- **식별자 클래스의 속성명과 엔티티에서 사용하는 식별자의 속성명이 같아야 한다.** 예제의 `Parent.id1`과 `ParentId.id1`, 그리고 `Parent.id2`과 `ParentId.id2`이 같다.
- `Serializable` 인터페이스를 구현해야 한다.
- `equals`, `hashCode`를 구현해야 한다.
- 기본 생성자가 있어야 한다.
- 식별자 클래스는 `public`이어야 한다.

그럼 실제 어떻게 사용하는지 알아보자.

==== 저장 코드 ====

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

저장 코드를 보면 식별자 클래스인 `ParentId`가 보이지 않는데, `em.persist()`를 호출하면 영속성 컨텍스트에 엔티티를 등록하기 직전에 내부에서 `Parent.id1`, `Parent.id2` 값을 사용해서 식별자 클래스인 `ParentId`를 생성하고 영속성 컨텍스트의 키로 사용한다.

==== 조회 코드 ====

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

조회 코드를 보면 식별자 클래스인 `ParentId`를 사용해서 엔티티를 조회한다.

자식 클래스

이제 자식 클래스를 추가해보자.

==== 자식 클래스 ====

```
@Entity
public class Child {

    @Id
    private String id;

    @ManyToOne
    @JoinColumns({
            @JoinColumn(name = "PARENT_ID1", referencedColumnName = "PARENT_ID1"
            @JoinColumn(name = "PARENT_ID2", referencedColumnName = "PARENT_ID2"
    })
    private Parent parent;
```

부모 테이블의 기본 키 컬럼이 복합 키이므로 자식 테이블의 외래 키도 복합 키다. 따라서 외래 키 매핑 시 여러 컬럼을 매핑해야 하므로 `@JoinColumns` 어노테이션을 사용하고 각각의 외래 키 컬럼을 `@JoinColumn` 으로 매핑한다.

참고로 예제처럼 `@JoinColumn` 의 `name` 속성과 `referencedColumnName` 속성의 값이 같으면 `referencedColumnName` 은 생략해도 된다.

- @EmbeddedId

`@IdClass` 가 데이터베이스에 맞춘 방법이라면 `@EmbeddedId` 는 좀 더 객체 지향적인 방법이다. 예제로 알아보자.

```
@Entity
public class Parent {

    @EmbeddedId
    private ParentId id;

    private String name;
    ...
}
```

`Parent` 엔티티에서 식별자 클래스를 직접 사용하고 `@EmbeddedId` 어노테이션을 적어주면 된다. 바로 식별자 클래스를 보자.

```
@Embeddable
public class ParentId implements Serializable {

    @Column(name = "PARENT_ID1")
```

```
    private String id1;

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

    //equals and hashCode 구현
    ...
}
```

`@IdClass`와는 다르게 `@EmbeddedId`를 적용한 식별자 클래스는 식별자 클래스에 기본 키를 직접 매핑한다.

`@EmbeddedId`를 적용한 식별자 클래스는 다음 조건을 만족해야 한다.

- `@Embeddable` 어노테이션을 붙여주어야 한다.
- `Serializable` 인터페이스를 구현해야 한다.
- `equals`, `hashCode`를 구현해야 한다.
- 기본 생성자가 있어야 한다.
- 식별자 클래스는 `public`이어야 한다.

`@EmbeddedId`를 사용하는 코드를 보자.

==== 저장 코드 ====

```
Parent parent = new Parent();
ParentId parentId = new ParentId("myId1","myId2");
parent.setId(parentId);
parent.setName("parentName");
em.persist(parent);
```

저장하는 코드를 보면 식별자 클래스 `parentId`를 직접 생성해서 사용한다.

==== 조회 코드 ====

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

조회 코드도 식별자 클래스 `parentId`를 직접 사용한다.

- 복합 키는 `equals`와 `hashCode`를 필수로 구현해야 한다.

```
parentId id1 = new parentId();
id1.setId1("myId1");
id1.setId2("myId2");

parentId id2 = new parentId();
id2.setId1("myId1");
id2.setId2("myId2");

id1.equals(id2) -> ?
```

이것은 순수한 자바 코드다. `id1` 과 `id2` 인스턴스 둘다 `myId1` , `myId2` 라는 같은 값을 가지고 있지만 인스턴스는 다르다. 그렇다면 마지막 줄에 있는 `id1.equals(id2)` 는 참일까 거짓일까?

`equals` 를 적절히 오버라이딩 했다면 참이겠지만 `equals` 를 적절히 오버라이딩 하지 않았다면 결과는 거짓이다. 왜냐하면 자바 `Object` 클래스가 제공하는 기본 `equals` 는 인스턴스 참조 값 비교인 `==` 비교(동일성 비교)를 하기 때문이다.

영속성 컨텍스트는 엔티티의 식별자를 키로 사용해서 엔티티를 관리한다. 그리고 식별자를 비교할 때 `equals` 와 `hashCode` 를 사용한다. 따라서 복합 키의 동등성(`equals` 비교)이 지켜지지 않으면 예상과 다른 엔티티가 조회되거나 엔티티를 찾을 수 없는 등 영속성 컨텍스트가 엔티티를 관리하는데 심각한 문제가 발생한다. 따라서 복합 키는 `equals` 와 `hashCode` 를 필수로 구현해야 한다.

- 정리

`@IdClass` 와 `@EmbeddedId` 는 각각 장단점이 있으므로 본인의 취향에 맞는 것을 사용하면 된다. `@EmbeddedId` 가 `@IdClass` 와 비교해서 더 객체 지향적이고 중복도 없어서 좋아보이긴 하지만 `JPQL` 이 조금 더 길어진다.

```
em.createQuery("select p.id.id1, p.id.id2 from Parent p"); // @EmbeddedId
em.createQuery("select p.id1, p.id2 from Parent p");       // @IdClass
```

> **참고** : 복합 키에는 `@GenerateValue` 를 사용할 수 없다. 복합 키를 구성하는 여러 컬럼중 하나에도 사용할 수 없다.

복합 키 - 식별 관계 매핑

복합 키와 식별 관계를 알아보자.

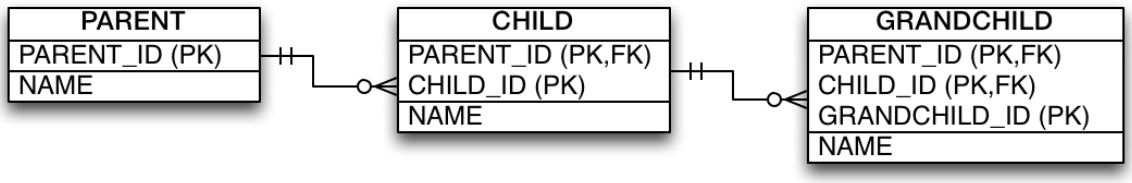

그림 - 식별 관계 구현

그림을 보면 부모, 자식, 손자까지 계속 기본 키를 전달하는 식별 관계다.

식별 관계에서 자식 테이블은 부모 테이블의 기본 키를 포함해서 복합 키를 구성해야 하므로 `@IdClass`나 `@EmbeddedId`를 사용해서 식별자를 매핑해야 한다. (뒤에서 설명할 일대일 관계는 약간 다르다.)

먼저 `@IdClass`로 식별 관계를 매핑해보자.

- @IdClass와 식별 관계

==== 부모 ====

```
@Entity
public class Parent {

    @Id @Column(name = "PARENT_ID")
    private String id;
    private String name;
    ...
}
```

==== 자식 ====

```
@Entity
@IdClass(ChildId.class)
public class Child {

    @Id
    @ManyToOne
    @JoinColumn(name = "PARENT_ID")
    public Parent parent;

    @Id @Column(name = "CHILD_ID")
```

```
    private String childId;

    private String name;
    ...
}
```

==== 자식ID ====

```
public class ChildId implements Serializable {

    private String parent; //Child.parent  매핑
    private String childId;//Child.childId 매핑

    //equals, hashCode
    ...
}
```

==== 손자 ====

```
@Entity
@IdClass(GrandChildId.class)
public class GrandChild {

    @Id
    @ManyToOne
    @JoinColumns({
            @JoinColumn(name = "PARENT_ID"),
            @JoinColumn(name = "CHILD_ID")
    })
    private Child child;

    @Id @Column(name = "GRANDCHILD_ID")
    private String id;

    private String name;
    ...
}
```

==== 손자ID ====

```
public class GrandChildId implements Serializable {

    private ChildId child; //GrandChild.child 매핑
```

```
    private String id;      //GrandChild.id 매핑

    //equals, hashCode
    ...
}
```

식별 관계는 기본 키와 외래 키를 같이 매핑해야 한다. 따라서 식별자 매핑인 `@Id`와 연관관계 매핑인 `@ManyToOne`을 같이 사용하면 된다.

```
@Id
@ManyToOne
@JoinColumn(name = "PARENT_ID")
public Parent parent;
```

`Child` 엔티티의 `parent` 필드를 보면 `@Id`로 기본 키를 매핑하면서 `@ManyToOne`과 `@JoinColumn`으로 외래 키를 같이 매핑한다.

- @EmbeddedId와 식별 관계

`@EmbeddedId`로 식별 관계를 구성할 때는 `@MapsId`를 사용해야 한다. 우선 코드부터 보자.

==== 부모 ====

```
@Entity
public class Parent {

    @Id @Column(name = "PARENT_ID")
    private String id;

    private String name;
    ...
}
```

==== 자식 ====

```
@Entity
public class Child {

    @EmbeddedId
    private ChildId id;
```

```
    @MapsId("parentId") //ChildId.parentId 매핑
    @ManyToOne
    @JoinColumn(name = "PARENT_ID")
    public Parent parent;

    private String name;
    ...
}
```

==== 자식ID ====

```
@Embeddable
public class ChildId implements Serializable {

    private String parentId; //@MapsId("parentId")로 매핑

    @Column(name = "CHILD_ID")
    private String id;

    //equals, hashCode
    ...
}
```

==== 손자 ====

```
@Entity
public class GrandChild {

    @EmbeddedId
    private GrandChildId id;

    @MapsId("childId") //GrandChildId.childId 매핑
    @ManyToOne
    @JoinColumns({
            @JoinColumn(name = "PARENT_ID"),
            @JoinColumn(name = "CHILD_ID")
    })
    private Child child;

    private String name;
    ...
}
```

==== 손자ID ====

```
@Embeddable
public class GrandChildId implements Serializable {

    private ChildId childId; //@MapsId("childId")로 매핑

    @Column(name = "GRANDCHILD_ID")
    private String id;

    //equals, hashCode
    ...
}
```

`@EmbeddedId`는 식별 관계로 사용할 연관관계의 속성에 `@MapsId`를 사용하면 된다. `Child` 엔티티의 `parent` 필드를 보자.

```
@MapsId("parentId")
@ManyToOne
@JoinColumn(name = "PARENT_ID")
public Parent parent;
```

`@IdClass`와 다른 점은 `@Id` 대신에 `@MapsId`를 사용한 점이다. `@MapsId`는 외래 키와 매핑한 연관관계를 기본 키에도 매핑하겠다는 뜻이다. `@MapsId`의 속성 값은 `@EmbeddedId`를 사용한 식별자 클래스의 기본 키 필드를 지정하면 된다. 여기서는 `ChildId`의 `parentId` 필드를 선택했다.

비식별 관계로 구현

앞 장에서 다대다 관계를 설명하면서 식별 관계를 비 식별관 관계로 변경했던 예제처럼, 방금 예를 들었던 식별 관계 테이블을 비식별 관계로 변경해보자.

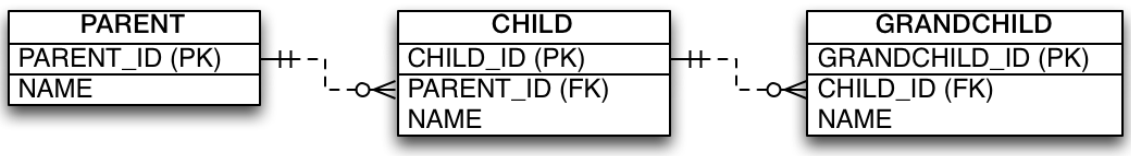

그림 - 비식별 관계로 변경

이렇게 비식별 관계로 만든 테이블을 매핑해보자.

==== 부모 ====

```
@Entity
public class Parent {

    @Id @GeneratedValue
    @Column(name = "PARENT_ID")
    private Long id;
    private String name;
    ...
}
```

==== 자식 ====

```
@Entity
public class Child {

    @Id @GeneratedValue
    @Column(name = "CHILD_ID")
    private Long id;
    private String name;

    @ManyToOne
    @JoinColumn(name = "PARENT_ID")
    private Parent parent;
    ...
}
```

==== 손자 ====

```
@Entity
public class GrandChild {

    @Id @GeneratedValue
    @Column(name = "GRANDCHILD_ID")
    private Long id;
    private String name;

    @ManyToOne
    @JoinColumn(name = "CHILD_ID")
    private Child child;
    ...
}
```

식별 관계의 복합 키를 사용한 코드와 비교하면 매핑도 쉽고 코드도 단순하다. 그리고 복합 키가 없으므로 복합 키 클래스를 만들지 않아도 된다.

일대일 식별 관계

일대일 식별 관계는 조금 특별하다. 바로 예제를 보자.

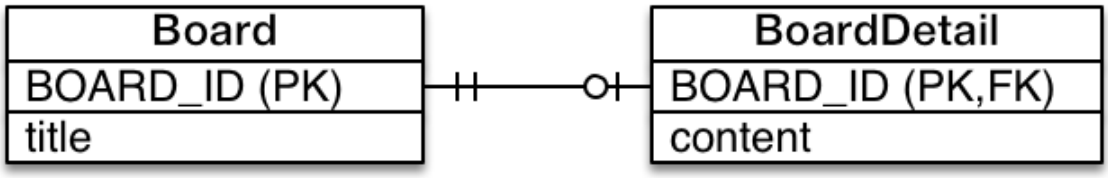

그림 - 식별 관계 일대일

일대일 식별 관계는 자식 테이블의 기본 키 값으로 부모 테이블의 기본 키 값만 사용한다. 그래서 부모 테이블의 기본 키가 복합 키가 아니면 자식 테이블의 기본 키는 복합 키로 구성하지 않아도 된다.

==== 부모 ====

```
@Entity
public class Board {

    @Id @GeneratedValue
    @Column(name = "BOARD_ID")
    private Long id;

    private String titie;

    @OneToOne(mappedBy = "board")
    private BoardDetail boardDetail;
    ...
}
```

==== 자식 ====

```
@Entity
public class BoardDetail {

    @Id
    private Long boardId;
```

```
    @MapsId //BoardDetail.boardId 매핑
    @OneToOne
    @JoinColumn(name="BOARD_ID")
    private Board board;

    private String content;
    ...
}
```

`BoardDetail`처럼 식별자가 단순히 컬럼 하나면 `@MapsId`를 사용하고 속성 값은 비워두면 된다. 이때 `@MapsId`는 `@Id`를 사용해서 식별자로 지정한 `BoardDetail.boardId`와 매핑된다.

이제 일대일 식별 관계를 사용하는 코드를 보자.

==== 저장 ====

```
public void save() {
    Board board = new Board();
    board.setTitie("제목");
    em.persist(board);

    BoardDetail boardDetail = new BoardDetail();
    boardDetail.setContent("내용");
    boardDetail.setBoard(board);
    em.persist(boardDetail);
}
```

식별, 비식별 관계의 장단점

데이터베이스 설계 관점에서 보면 다음과 같은 이유로 식별 관계보다는 비식별 관계를 선호한다.

- 식별 관계는 부모 테이블의 기본 키를 자식 테이블로 전파하면서 자식 테이블의 기본 키 컬럼이 점점 늘어난다. 예를 들어 부모 테이블은 기본 키 컬럼이 하나였지만 자식 테이블은 기본 키 컬럼이 2개, 손자 테이블은 기본 키 컬럼이 3개로 점점 늘어난다. 결국 조인할 때 SQL이 복잡해지고 기본 키 인덱스가 불필요하게 커질 수 있다.
- 식별 관계는 2개 이상의 컬럼을 합해서 복합 기본 키를 만들어야 하는 경우가 많다.
- 식별 관계를 사용할 때 기본 키로 비즈니스 의미가 있는 자연 키 컬럼을 조합하는 경우가 많다. 반면에 비식별 관계의 기본 키는 비즈니스와 전혀 관계없는 대리 키를 주로 사용한다. 비즈니스 요구사항은 시간이 지남에 따라 언젠가는 변한다. 식별 관계의 자연 키 컬럼들이 자식에 손자까지 전파되면 변경하기 힘들다.

- 식별 관계는 부모 테이블의 기본 키를 자식 테이블의 기본 키로 사용하므로 비식별 관계보다 테이블 구조가 유연하지 못하다.

객체 관계 매핑의 관점에서 보면 다음과 같은 이유로 비식별 관계를 선호한다.

- 일대일 관계를 제외하고 식별 관계는 2개 이상의 컬럼을 묶은 복합 기본 키를 사용한다. JPA에서 복합 키는 별도의 복합 키 클래스를 만들어서 사용해야 한다. 따라서 컬럼이 하나인 기본 키를 매핑하는 것보다 많은 노력이 필요하다.
- 비식별 관계의 기본 키는 주로 대리 키를 사용하는데 JPA는 `@GenerateValue`처럼 대리 키를 생성하기 위한 편리한 방법을 제공한다.

물론 식별 관계가 가지는 장점도 있다. 기본 키 인덱스를 활용하기 좋고, 상위 테이블들의 기본 키 컬럼을 자식, 손자 테이블들이 가지고 있으므로 특정 상황에 조인 없이 하위 테이블만으로 검색을 완료할 수 있다.

기본 키 인덱스를 활용하는 예를 보자.

부모 아이디가 A인 모든 자식 조회

```
SELECT * FROM CHILD
WHERE PARENT_ID = 'A'
```

부모 아이디가 A고 자식 아이디가 B 자식 조회

```
SELECT * FROM CHILD
WHERE PARENT_ID = 'A' AND CHILD_ID = 'B'
```

두 경우 모두 `CHILD` 테이블의 기본 키 인덱스를 `PARENT_ID` + `CHILD_ID`로 구성하면 별도의 인덱스를 생성할 필요 없이 기본 키 인덱스만 사용해도 된다.

이처럼 식별 관계가 가지는 장점도 있으므로 꼭 필요한 곳에는 적절하게 사용하는 것이 데이터베이스 테이블 설계의 묘를 살리는 방법이다.

비식별 관계를 사용하세요.

ORM 신규 프로젝트 진행시 추천하는 방법은 될 수 있으면 **비식별 관계를 사용하고 기본 키는 `Long` 타입의 대리 키를 사용**하는 것이다. 대리 키는 비즈니스와 아무 관련이 없다. 따라서 비즈니스가 변경되어도 유연한 대처가 가능하다는 장점이 있다. JPA는 `@GenerateValue`를 통해 간편하게 대리 키를 생

성할 수 있다. 그리고 식별자 컬럼이 하나여서 쉽게 매핑할 수 있다. 식별자의 데이터 타입은 `Long`을 추천하는데, 자바에서 `Int`는 20억 정도면 끝나버리므로 데이터를 많이 저장하면 문제가 발생할 수 있다. 반면에 `Long`은 아주 커서 안전하다.

그리고 선택적 비식별 관계보다는 필수적 비식별 관계를 사용하는 것이 좋은데, 선택적인 비식별 관계는 `NULL`을 허용하므로 조인할 때에 외부 조인을 사용해야 한다. 반면에 필수적 관계는 `NOT NULL`로 항상 관계가 있다는 것을 보장하므로 내부 조인만 사용해도 된다.

조인 테이블

데이터베이스 테이블의 연관관계를 설계하는 방법은 크게 2가지가 있다.

- 조인 컬럼 사용 (외래 키)
- 조인 테이블 사용 (테이블 사용)

조인 컬럼 사용

테이블간에 관계는 주로 **조인 컬럼**이라 부르는 외래 키 컬럼을 사용해서 관리한다.

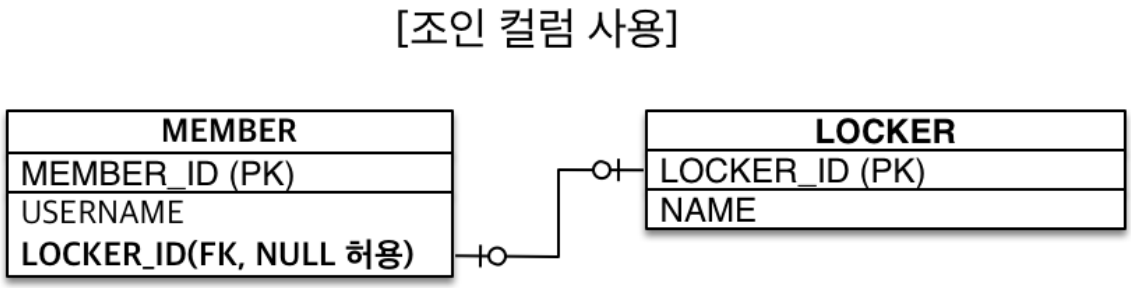

그림 - 조인 컬럼 사용

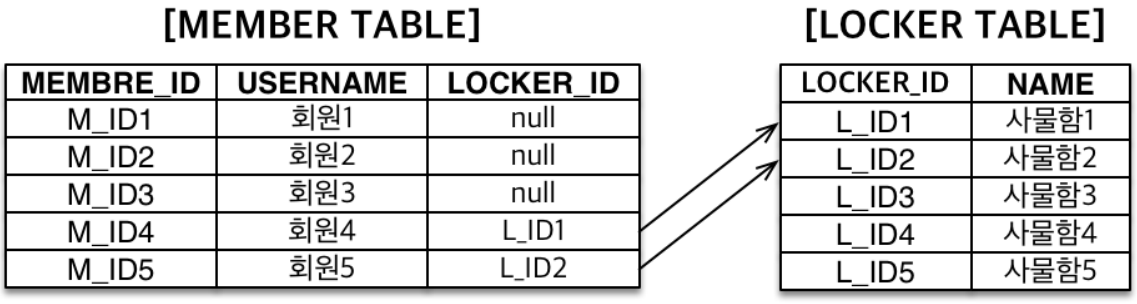

그림 - 조인 컬럼 데이터

[그림 - 조인 컬럼 사용]을 보자. 예를 들어 회원과 사물함이 있는데 각각 테이블에 데이터를 등록했다가 회원이 원할 때 사물함을 선택할 수 있다고 가정해보자. 회원이 사물함을 사용하기 전까지는 아직 둘 사이에 관계가 없으므로 `MEMBER` 테이블의 `LOCKER_ID` 외래 키에 `null`을 입력해두어야 한다. 이렇게 외래 키에 `null`을 허용하는 관계를 선택적 비식별 관계라 한다.

선택적 비식별 관계는 외래 키에 `null`을 허용하므로 회원과 사물함을 조인할 때 외부 조인(OUTER JOIN)을 사용해야 한다. 실수로 내부 조인을 사용하면 사물함과 관계가 없는 회원은 조회되지 않는다. 그리고 회원과 사물함이 아주 가끔 관계를 맺는다면 외래 키 값 대부분이 `null`로 저장되는 단점이 있다.

조인 테이블 사용

이번에는 조인 컬럼을 사용하는 대신에 조인 테이블을 사용해서 연관관계를 관리해보자.

그림 - 조인 테이블 사용

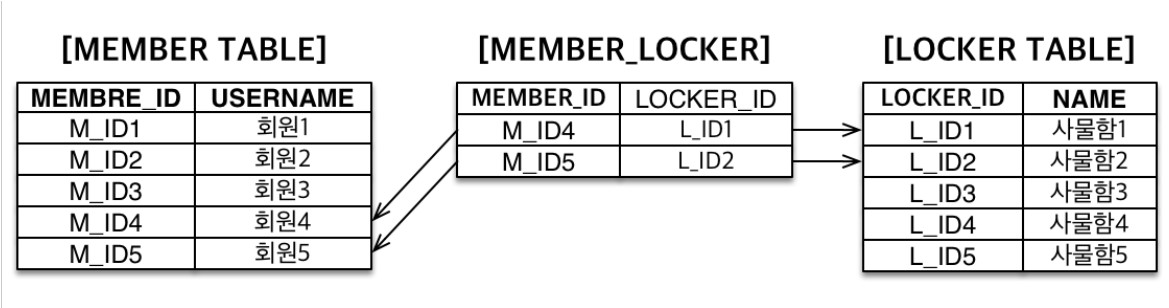

그림 - 조인 테이블 데이터

이 방법은 조인 테이블이라는 별도의 테이블을 사용해서 연관관계를 관리한다. [그림 - 조인 컬럼 사용]과 [그림 - 조인 테이블 사용]을 비교해보면 조인 컬럼을 사용하는 방법과 조인 테이블을 사용하는 방법의 차이를 알 수 있다. 조인 컬럼을 사용하는 방법은 단순히 외래 키 컬럼만 추가해서 연관관계를 맺지만 조인 테이블을 사용하는 방법은 연관관계를 관리하는 조인 테이블(`MEMBER_LOCKER`)을 추가하고 여기서 두 테이블의 외래 키를 가지고 연관관계를 관리한다. 따라서 `MEMBER`와 `LOCKER`에는 연관관계를 관리하기 위한 외래 키 컬럼이 없다.

[그림 - 조인 테이블 데이터]를 보면 회원과 사물함 데이터를 각각 등록했다가 회원이 원할 때 사물함을 선택하면 `MEMBER_LOCKER` 테이블에만 값을 추가하면 된다.

조인 테이블의 가장 큰 단점은 테이블을 하나 추가해야 한다는 점이다. 따라서 관리해야 하는 테이블이 늘어나고 회원과 사물함 두 테이블을 조인하려면 `MEMBER_LOCKER` 테이블까지 추가로 조인해야 한다. 따라서 기본은 조인 컬럼을 사용하고 필요하다고 판단되면 조인 테이블을 사용하자.

조인 테이블에 대해 앞으로 설명할 내용은 다음과 같다.

- 객체와 테이블을 매핑할 때 조인 컬럼은 `@JoinColumn`으로 매핑하고 조인 테이블은 `@JoinTable`로 매핑한다.
- 조인 테이블은 주로 다대다 관계를 일대다, 다대일 관계로 풀어내기 위해 사용하지만 일대일, 일대다, 다대일 관계에서도 사용한다.

지금부터 일대일, 일대다, 다대일, 다대다 관계를 조인 테이블로 매핑해보자.

> **참고:** 조인 테이블을 연결 테이블, 링크 테이블로도 부른다.

- 일대일 조인테이블

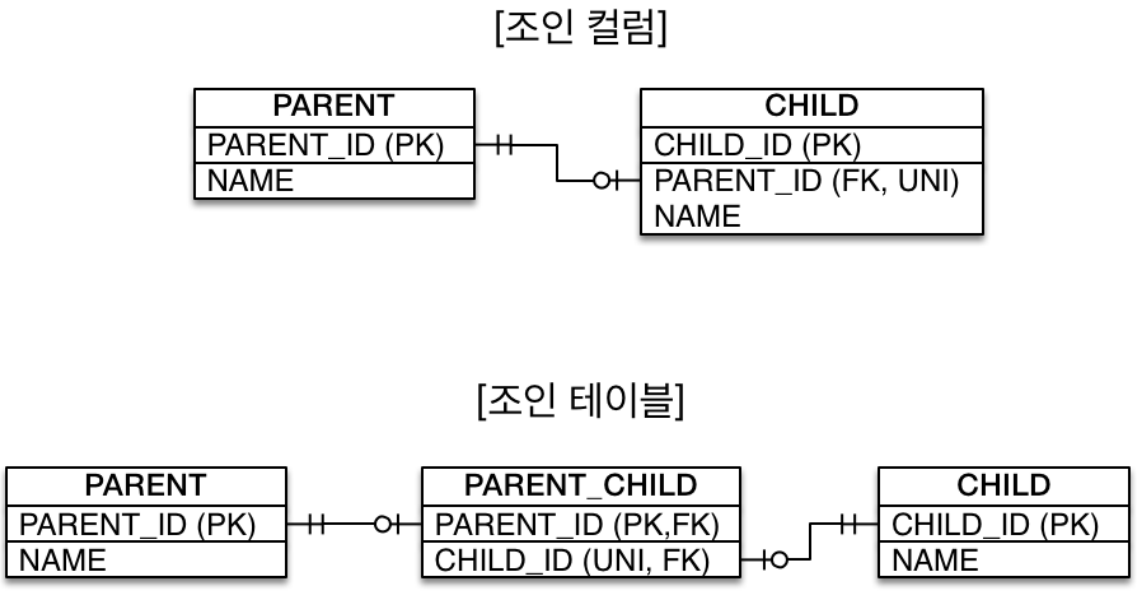

그림 - 조인테이블 일대일

[그림 - 조인테이블 일대일]에서 조인테이블을 보자. 일대일 관계를 만들려면 조인 테이블의 외래 키 컬럼 각각에 총 2개의 유니크 제약조건을 걸어야 한다. (`PARENT_ID`는 기본 키이므로 유니크 제약조건이 걸려있다.)

==== 부모 ====

```
@Entity
public class Parent {

    @Id @GeneratedValue
    @Column(name = "PARENT_ID")
    private Long id;
    private String name;

    @OneToOne
    @JoinTable(name = "PARENT_CHILD",
            joinColumns = @JoinColumn(name = "PARENT_ID"),
            inverseJoinColumns = @JoinColumn(name = "CHILD_ID")
    )
    private Child child;
    ...
}
```

==== 자식 ====

```
@Entity
public class Child {

    @Id @GeneratedValue
    @Column(name = "CHILD_ID")
    private Long id;
    private String name;
    ...
}
```

부모 엔티티를 보면 `@JoinColumn` 대신에 `@JoinTable`을 사용했다.

`@JoinTable`의 속성

- `name` : 매핑할 조인 테이블 이름
- `joinColumns` : 현재 엔티티를 참조하는 외래 키
- `inverseJoinColumns` : 반대방향 엔티티를 참조하는 외래 키

양방향 매핑

양방향으로 매핑하려면 다음 코드를 추가하면 된다.

====== 양방향 매핑 추가 ======

```
public class Child {
    ...
    @OneToOne(mappedBy="child")
    private Parent parent;
}
```

- 일대다 조인테이블

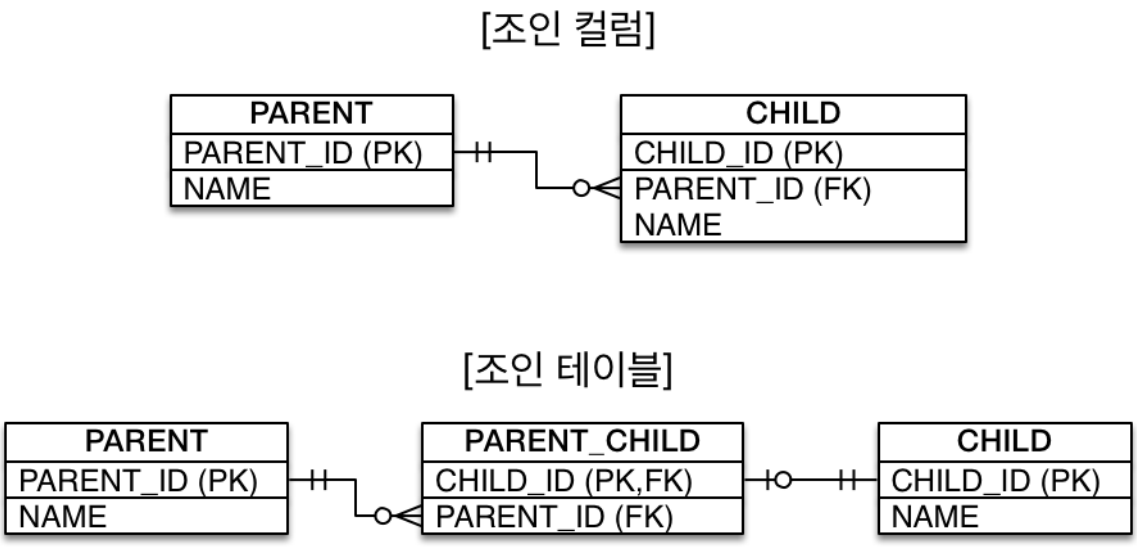

그림 - 조인테이블 일대다, 다대일

그림은 일대다 관계다. 일대다 관계를 만들려면 조인 테이블의 컬럼중 다(N)와 관련된 컬럼인 `CHILD_ID`에 유니크 제약조건을 걸어야 한다. (`CHILD_ID` 는 기본 키이므로 유니크 제약조건이 걸려 있다.)

일대다 단방향 관계로 매핑해보자.

==== 부모 ====

```
@Entity
public class Parent {

    @Id @GeneratedValue
    @Column(name = "PARENT_ID")
    private Long id;
    private String name;

    @OneToMany
    @JoinTable(name = "PARENT_CHILD",
```

```
            joinColumns = @JoinColumn(name = "PARENT_ID"),
            inverseJoinColumns = @JoinColumn(name = "CHILD_ID")
    )
    private List<Child> child = new ArrayList<Child>();
    ...
}
```

==== 자식 ====

```
@Entity
public class Child {

    @Id @GeneratedValue
    @Column(name = "CHILD_ID")
    private Long id;
    private String name;
    ...
}
```

- 다대일 조인테이블

다대일은 일대다에서 방향만 반대이므로 조인 테이블 모양은 일대다에서 설명한 [그림 - 조인테이블 일대다, 다대일]과 같다.

다대일, 일대다 양방향 관계로 매핑해보자.

==== 부모 ====

```
@Entity
public class Parent {

    @Id @GeneratedValue
    @Column(name = "PARENT_ID")
    private Long id;
    private String name;

    @OneToMany(mappedBy = "parent")
    private List<Child> child = new ArrayList<Child>();
    ...
}
```

==== 자식 ====

```
@Entity
public class Child {

    @Id @GeneratedValue
    @Column(name = "CHILD_ID")
    private Long id;
    private String name;

    @ManyToOne(optional = false)
    @JoinTable(name = "PARENT_CHILD",
            joinColumns = @JoinColumn(name = "CHILD_ID"),
            inverseJoinColumns = @JoinColumn(name = "PARENT_ID")
    )
    private Parent parent;
    ...
}
```

- 다대다 조인테이블

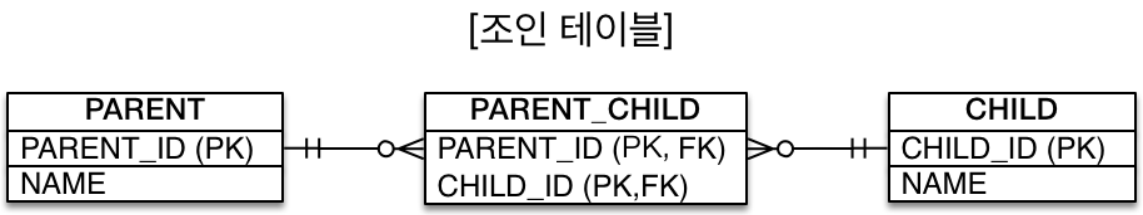

그림 - 조인테이블 다대다

그림은 다대다 관계다. 다대다 관계를 만들려면 조인 테이블의 두 컬럼을 합해서 하나의 복합 유니크 제약조건을 걸어야 한다. (`PARENT_ID` , `CHILD_ID` 는 복합 기본 키이므로 유니크 제약조건이 걸려있다.)

==== 부모 ====

```
@Entity
public class Parent {

    @Id @GeneratedValue
    @Column(name = "PARENT_ID")
    private Long id;
    private String name;

    @ManyToMany
    @JoinTable(name = "PARENT_CHILD",
            joinColumns = @JoinColumn(name = "PARENT_ID"),
```

```
            inverseJoinColumns = @JoinColumn(name = "CHILD_ID")
    )
    private List<Child> child = new ArrayList<Child>();
    ...
}
```

==== 자식 ====

```
@Entity
public class Child {

    @Id @GeneratedValue
    @Column(name = "CHILD_ID")
    private Long id;
    private String name;
    ...
}
```

> **참고:** 조인 테이블에 컬럼을 추가하면 `@JoinTable` 전략을 사용할 수 없다. 대신에 새로운 엔티티를 만들어서 조인 테이블과 매핑해야 한다.

엔티티 하나에 여러 테이블 매핑하기

잘 사용하지는 않지만 `@SecondaryTable`을 사용하면 한 엔티티에 여러 테이블을 매핑할 수 있다.

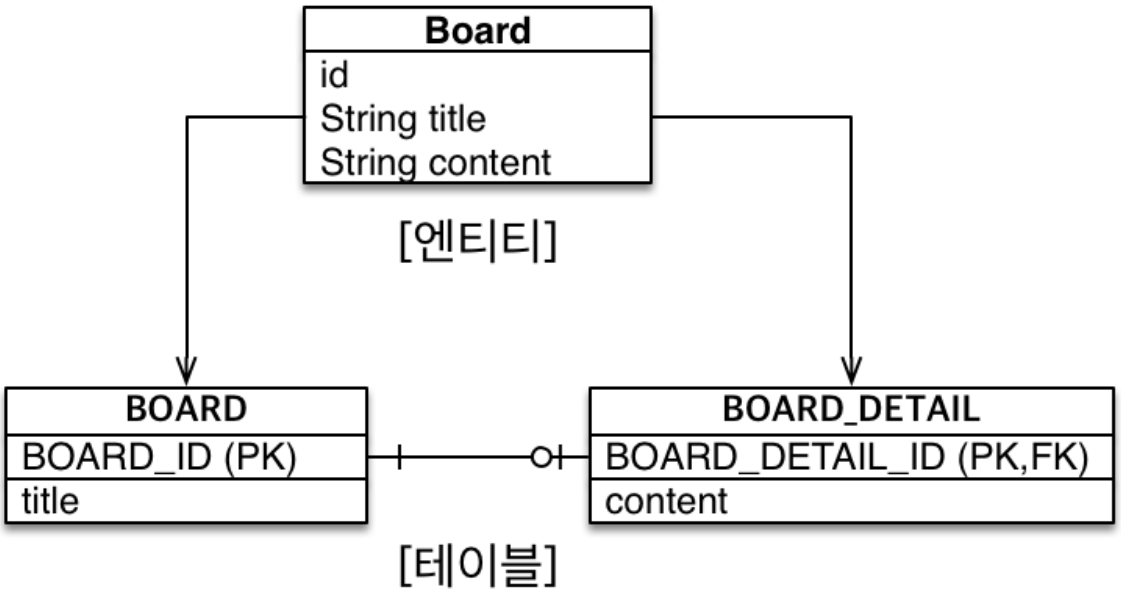

그림 - 하나의 엔티티에 여러 테이블 매핑하기

==== 하나의 엔티티에 여러 테이블 매핑하기 예제 ====

```
@Entity
@Table(name="BOARD")
@SecondaryTable(name = "BOARD_DETAIL",
        pkJoinColumns = @PrimaryKeyJoinColumn(name = "BOARD_DETAIL_ID"))
public class Board {

    @Id @GeneratedValue
    @Column(name = "BOARD_ID")
    private Long id;

    private String title;

    @Column(table = "BOARD_DETAIL")
    private String content;
    ...
}
```

`Board` 엔티티는 `@Table`을 사용해서 `BOARD` 테이블과 매핑했다. 그리고 `@SecondaryTable`을 사용해서 `BOARD_DETAIL` 테이블을 추가로 매핑했다.

`@SecondaryTable` 속성

- `@SecondaryTable.name` : 매핑할 다른 테이블의 이름, 예제에서는 테이블명을 `BOARD_DETAIL`로 지정했다.

- `@SecondaryTable.pkJoinColumns` : 매핑할 다른 테이블의 기본 키 컬럼 속성, 예제에서는 기본 키 컬럼명을 `BOARD_DETAIL_ID` 로 지정했다.

```
@Column(table = "BOARD_DETAIL")
private String content;
```

`content` 필드는 `@Column(table = "BOARD_DETAIL")` 을 사용해서 `BOARD_DETAIL` 테이블의 컬럼에 매핑했다. `title` 필드처럼 테이블을 지정하지 않으면 기본 테이블인 `BOARD` 에 매핑된다.

더 많은 테이블을 매핑하려면 `@SecondaryTables` 를 사용하면 된다.

====== `@SecondaryTables` 예제 ======

```
@SecondaryTables({
          @SecondaryTable(name="BOARD_DETAIL"),
          @SecondaryTable(name="BOARD_FILE")
      })
```

참고로 `@SecondaryTable` 을 사용해서 두 테이블을 하나의 엔티티에 매핑하는 방법보다는 테이블당 엔티티를 각각 만들어서 일대일 매핑하는 것을 권장한다. 이 방법은 항상 두 테이블을 조회하므로 최적화하기 어렵다. 반면에 일대일 매핑은 원하는 부분만 조회할 수 있고 필요하면 둘을 함께 조회하면 된다.