JPA 값 타입 & 비교

- 기본값 타입
- 인베디드 타입 (복합 값 타입)
- 값 타입과 불변 객체
- 값 타입의 비교
- 값 타입 컬렉션

JPA의 데이터 타입 분류

- **엔티티 타입**
  - @Entity로 정의하는 객체
  - 데이터가 변해도 식별자로 지속해서 추적 가능
- **값 타입**
  - int, Integer, String처럼 단순히 값으로 사용하는 자바 기본 타입이나 객체
  - 식별자가 없고 값만 있으므로 변경시 추적 불가

  - 기본값 타입
    - 자바 기본 타입 (int, double)
    - 래퍼 클래스 (Integer, Long)
    - String
  - 임베디드 타입
    - embedded type
    - 복합 ;값 타입
  - 컬렉션 값 타입
    - collection value type

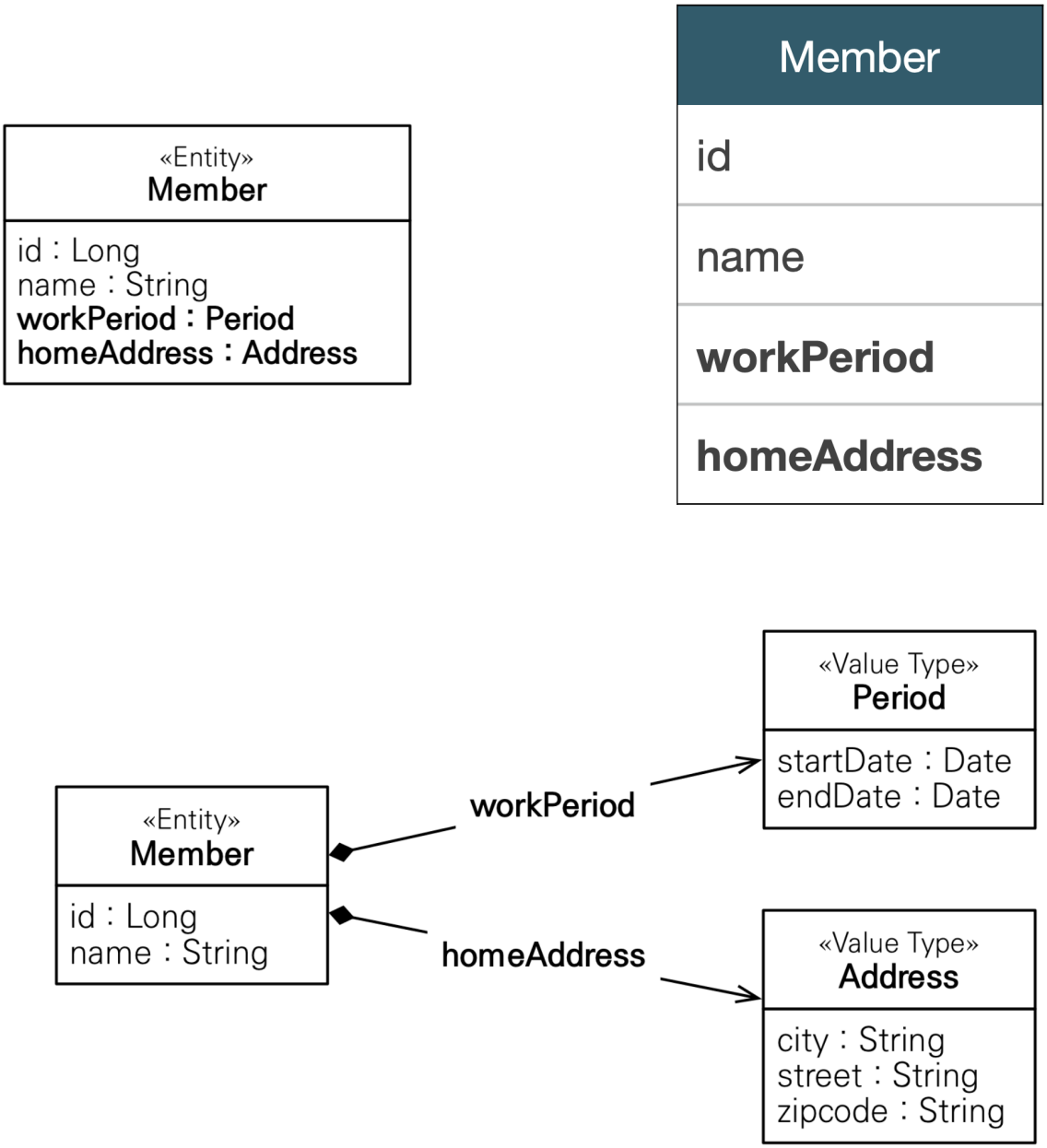
    ➜ 생명주기를 엔티티의 의존

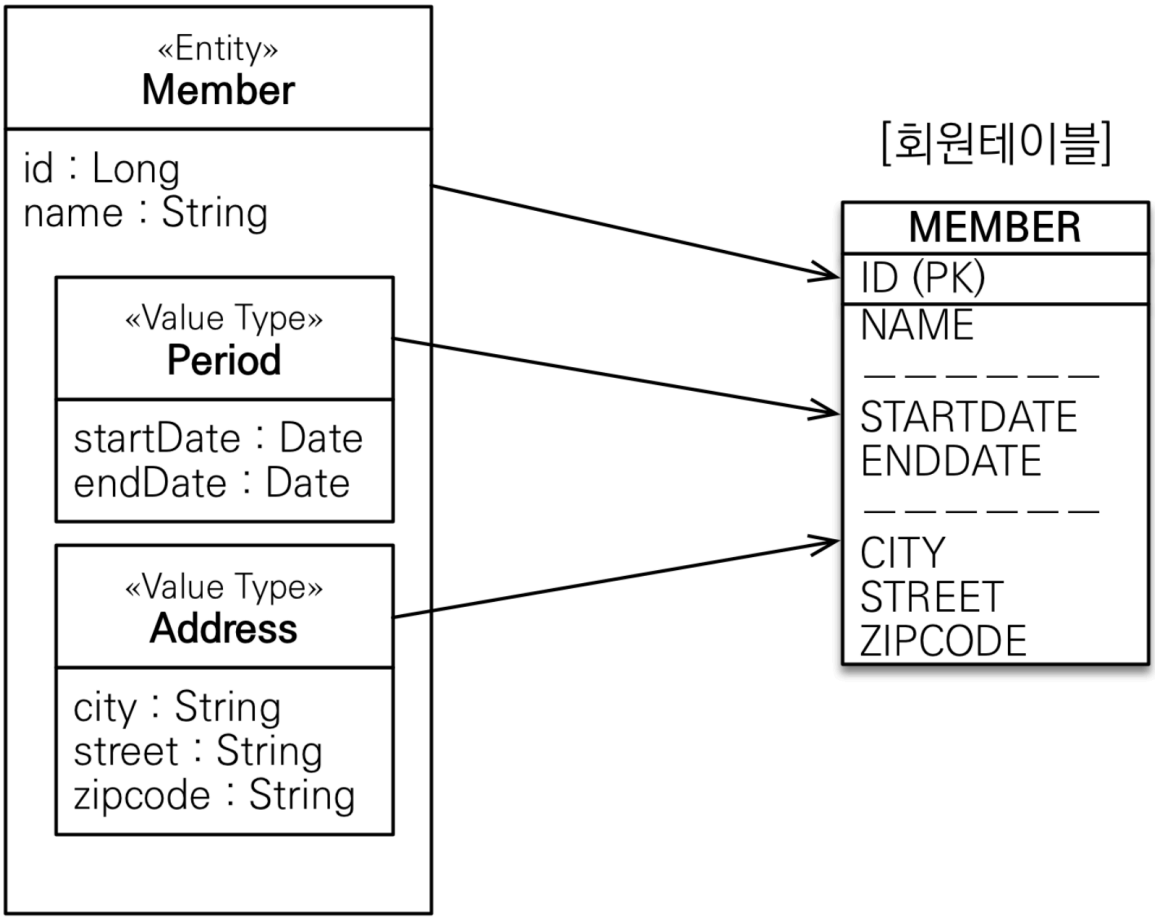
    ➜ 값 타입은 공유하면 하지 않는다. 다른 스택 메모리에 저장되기 때문

    ➜ Integer 같은 래퍼 클래스나 String 같은 특수한 클래스는 공유 가능한 객체이지만 변경하면 안된다. 참조 값이 넘어가기때문

- **임베디드 타입**
  - 새로운 값 타입을 직접 정의할 수 있음
  - JPA는 임베디드 타입(embedded type)이라 함
  - 주로 기본 값 타입을 모아서 만들어서 복합 값 타입이라고도 함

- int, String과 같은 값 타입



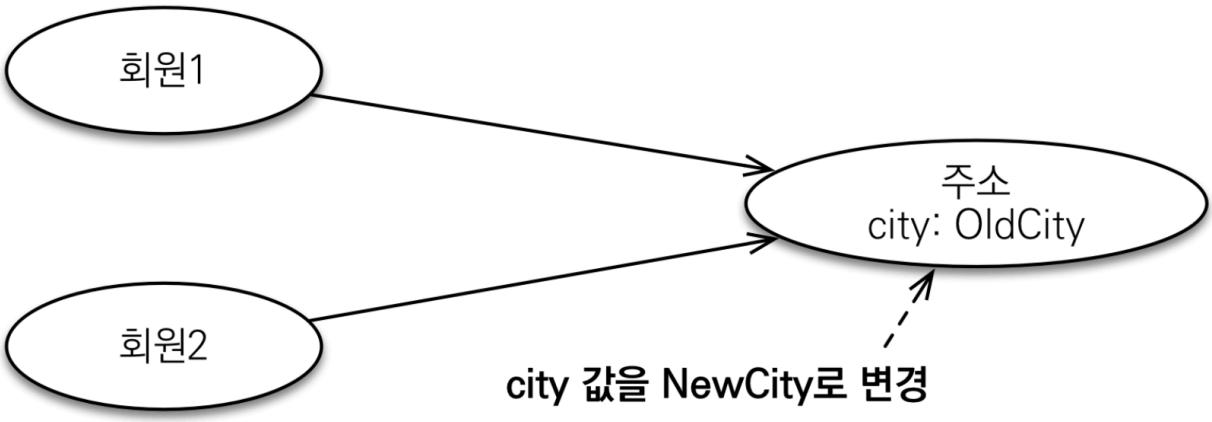


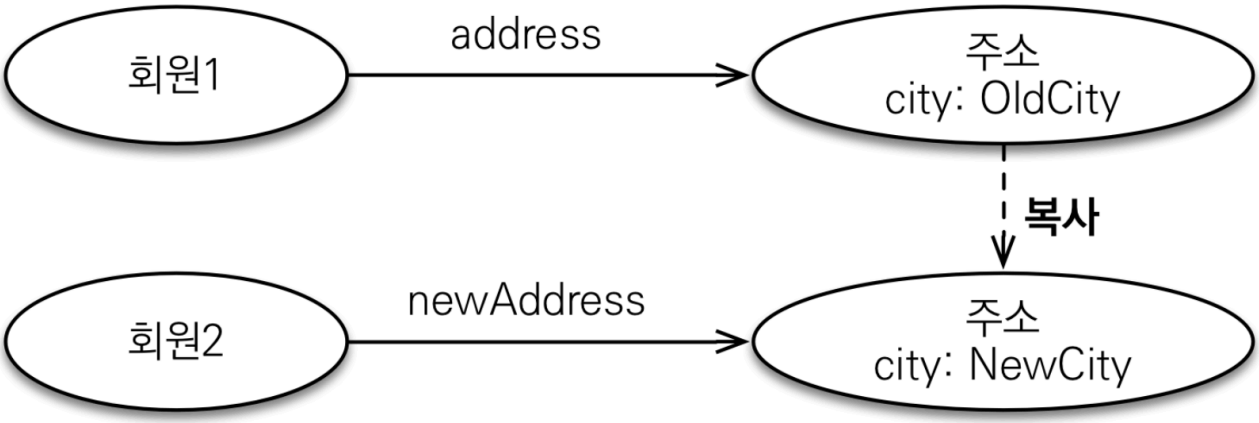
- 재사용 가능
- 높은 응집도
- 드를 만들 수 있음
- Period.isWork()처럼 해당 값 타입만 사용하는 의미 있는 메소드 만들 수 있다.
- 임베디드 타입을 포함한 모든 값 타입은, 값 타입을 소유한 엔티티에 생명주기를 의존함
- 임베디드 타입은 엔티티의 값일 뿐이다.
- 임베디드 타입으로 따로 빼면 객체별 기능 구현을 할 수 있어 유용하다.
- @Embeddable 기본 생성자 필수
- 중복되는 객체로 다른 필드명으로 저장할 때 사용

  @AttributeOverrides, @AttributeOverride를 사용해서 컬러 명 속성을 재정의
- 임베디드 타입의 값이 null이면 매핑한 컬럼 값은 모두 null

➜ 임베디드 타입 같은 값 타입을 여러 엔티티에서 공유하면 위험함 부작용(side effect) 발생

➜ 대신 값(인스턴스)를 복사해서 사용





- **값 타입과 불변 객체**
  - 값 타입은 복잡한 객체 세상을 조금이라도 단순화하려고 만든 개념이다. 따라서 값 타입은 단순하고 안전하게 다 룰 수 있어야 한다.
  - 항상 값을 복사해서 사용하면 공유 참조로 인해 발생하는 부작용 을 피할 수 있다.
  - 문제는 임베디드 타입처럼 **직접 정의한 값 타입은 자바의 기본 타입이 아니라 객체 타입**이다.
  - 자바 기본 타입에 값을 대입하면 값을 복사한다.
  - 객체 타입은 참조 값을 직접 대입하는 것을 막을 방법이 없다
  - 객체의 공유 참조는 피할 수 없다
  -

기본 타입(primitive type)

```
int a = 10;
int b = a;//기본 타입은 값을 복사
b = 4;
```

객체 타입

```
Address a = new Address("Old");
Address b = a; //객체 타입은 참조를 전달
b. setCity("New")
```

그러므로 객체 타입은 불변 객체로 만들어야한다.

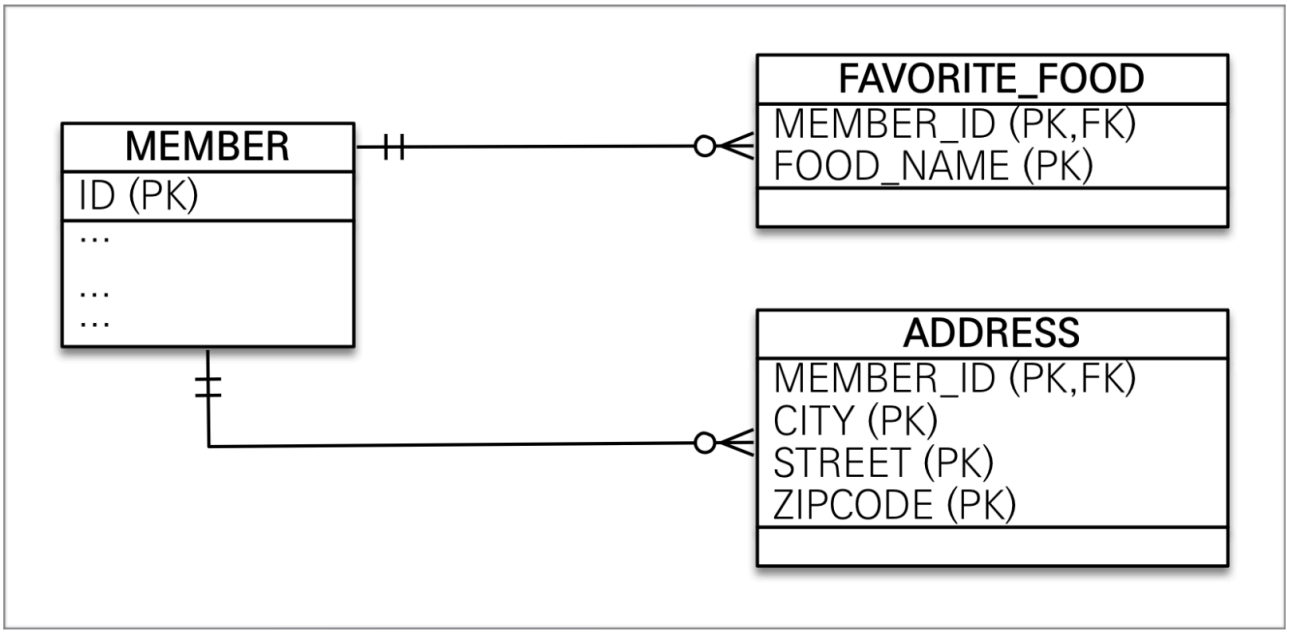
- 객체 타입을 수정할 수 없게 만들면 **부작용을 원천 차단
- 값 타입은 불변 객체(immutable object)로 설계해야함
- 불변 객체 생성 시점 이후 절대 값을 변경할 수 없는 객체
- 생성자로만 값을 설정하고 수정자(Setter)를 만들지 않으면 됨
- 참고: Integer, String은 자바가 제공하는 대표적인 불변 객체

값 비교

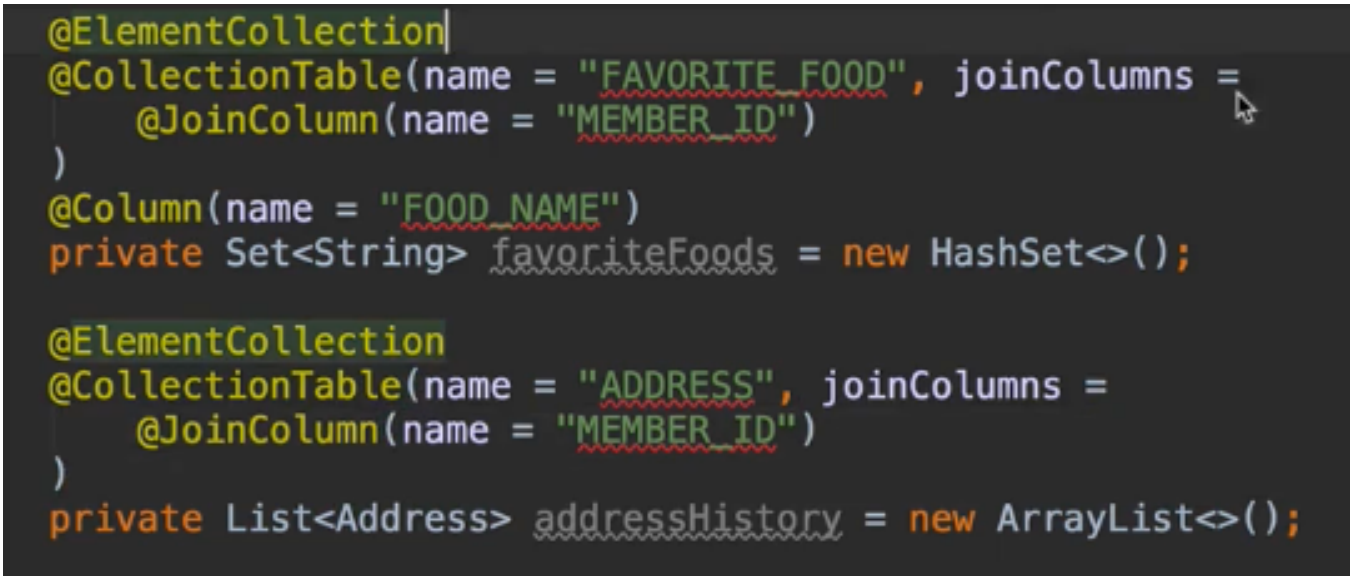
- 동일성(identity) 비교: 인스턴스의 참조 값을 비교, == 사용
- 동등성(equivalence) 비교: 인스턴스의 값을 비교, equals() 사용
  - equals()
  - equals() 만들면 hashCode도 맞게 구현해줘야한다.
- 값 타입은 a.equals(b)를 사용해서 동등성 비교를 해야 함
- 값 타입의 equals() 메소드를 적절하게 재정의(주로 모든 필드 사용)

값 타입 컬렉션

- 값 타입이기 때문에 라이플 사이클이 달라도 다른 테이블의 값을 저장 조회할 수 있다.



- 값 타입을 하나 이상 저장할 때 사용
- @ElementCollection, @CollectionTable 사용
- 데이터베이스는 컬렉션을 같은 테이블에 저장할 수 없다.
- 컬렉션을 저장하기 위한 별도의 테이블이 필요함



- 값 타입 컬렉션도 지연 로딩 전략 사용
- 참고: 값 타입 컬렉션은 영속성 전에(Cascade) + 고아 객체 제 거 기능을 필수로 가진다고 볼 수 있다.
- equals 랑 hashcode가 제대로 들어가 있어야 .remove() 할 때 equal 검색 후 일치하는 값이 삭제된다.

제약 사항

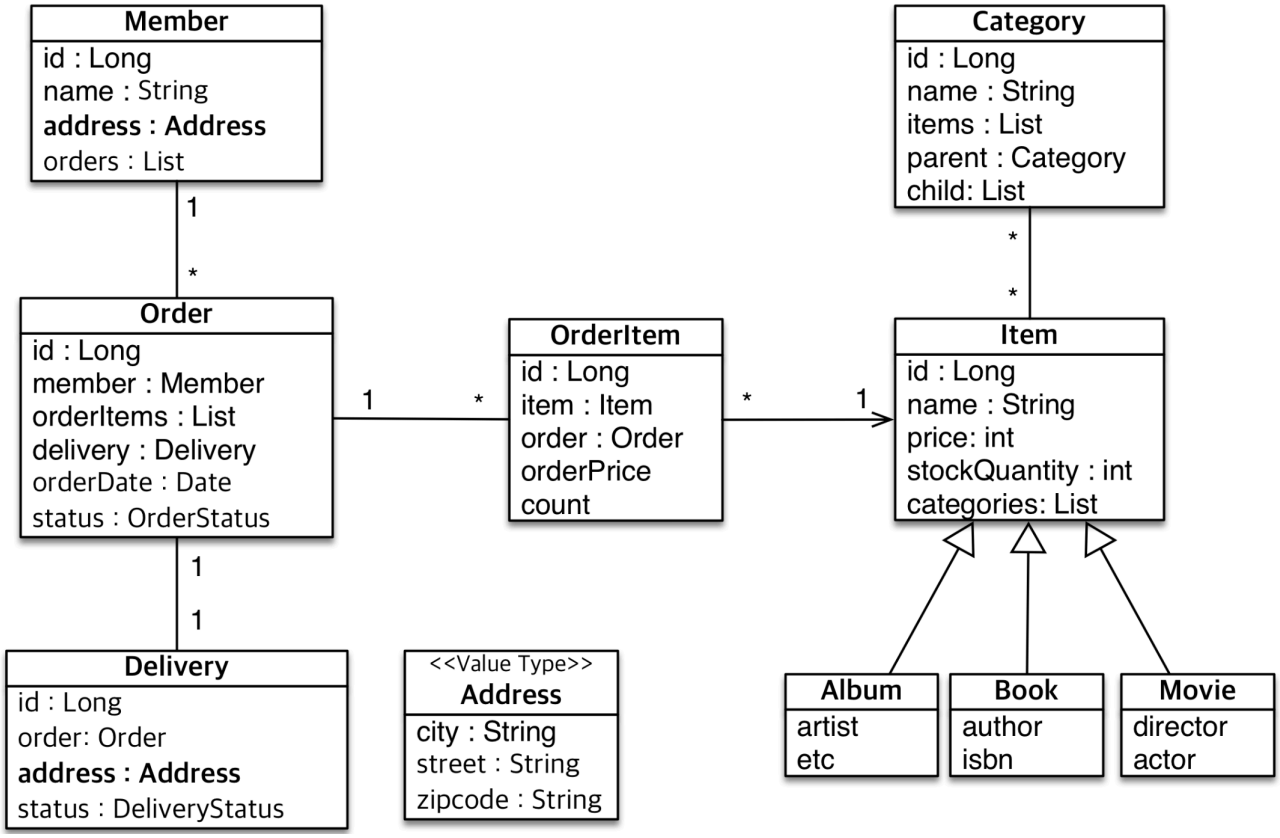
- 값 타입은 엔티티와 다르게 식별자 개념이 없다.
- 값은 변경하면 추적이 어렵다.
- 값 타입 컬렉션에 변경 사항이 발생하면,
  주인 엔티티와 연관된 모든 데이터를 삭제하고,
  값 타입 컬렉션에 있는 현재 값을 모두 다시 저장한다.
- 값 타입 컬렉션을 매핑하는 테이블은 모든 컬럼을 묶어서 기본 키를 구성해야 함:
  - null 입력X, 중복 저장 X

값 타입 컬렉션 대안

- 실무에서는 상황에 따라 **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 컬 렉션 처럼 사용

- 값 타입은 정말 값 타입이라 판단될 때만 사용
- 엔티티와 값 타입을 혼동해서 엔티티를 값 타입으로 만들면 안됨
- 식별자가 필요하고, 지속해서 값을 추적, 변경해야 한다면 그것 은 값 타입이 아닌 엔티티

- **엔티티 타입의 특징**
  - 식별자O
  - 생명 주기 관리
  - 공유
- **값 타입의 특징**
  - 식별자X
  - 생명 주기를 엔티티에 의존
  - 공유하지 않는 것이 안전(복사해서 사용)
  - 불변 객체로 만드는 것이 안전



#dev/jpa/tutorial