JPA 연관관계 맵핑

양방향 연관관계 매핑시 무한루프를 조심해라!

toString(), lombok, JSON

lombok에서 toString() 만드는 것은 웬만하면 빼고 써라!

엔티티는 Dto로 변환해서 반환해라.

단방향 매핑만으로도 이미 연관관계 매핑은 완료!!

양방향 매핑은 반대방향으로 조회 기능이 추가된 것 뿐.

JPQL에서 역방향 조회할 일이 많기는 한데, 역방향 조회시 단방향만 매핑해놓고 양방향 매핑은 필요할 때 추가해도 됨. (테이블에는 영향 주지 않음)

연관관계의 주인은 외래키의 위치를 기준으로 정해야한다.

연관관계 매핑시 고려사항 3가지

- 다중성
- 단방향, 양방향
- 연관관계의 주인

상속관계 매핑

- 관계형 데이터베이스에는 상속관계가 없다.
- 슈퍼타입 서브타입 관계라는 모델링 기법이 객체 상속과 유사하다.

슈퍼타입 서브타입 논리 모델을 실제 물리 모델로 구현하는 방법

- 각각 테이블로 변환 > 조인 전략
- 통합 테이블로 변환 > 단일 테이블 전략
- 서브타입 테이블로 변환 > 구현 클래스마다 테이블 전략

@Inheritance(strategy=InheritanceType.~~~)

- JOINED : 조인전략
- SINGLE_TABLE : 단일 테이블 전략
- TABLE_PER_CLASS : 구현 클래스마다 테이블 전략

SuperClass

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)   // JOIN전략을 사용해서 상속하겠다.
@DiscriminatorColumn(name="DType")   // Item 테이블에 저장되는 데이터가 어떤 객체타입인지 표시
public class Item {
  @Id @GeneratedValue
  private Long id;

  private String name;
  private int price;
}
```

SubClass

```
@Entity
@DiscriminatorValue("B")   // Dtype컬럼에 저장될 이름, default값은 엔티티 이름
public class Book extends Item{
  private String author;
  private String isbn;
}
```

상속 전략의 디폴트는 단일테이블 전략이다.

Table_per_class 전략에서는 Discriminator가 의미없음

▼ **각 전략의 장단점**

JOIN 전략 - 가장 객체지향적인 전략

▼ 장점

- 테이블 정규화
- 외래 키 참조 무결성 제약조건 활용가능
- 저장공간 효율화

▼ 단점

- 조회시 조인을 많이 사용하게 되어 성능 저하가 발생함
- 조회 쿼리가 복잡함
- 데이터 저장시 INSERT SQL 2번 호출함

단일 테이블 전략

▼ 장점

- 조인이 필요없으므로 일반적으로 조회 성능이 빠름
- 조회 쿼리가 단순

▼ 단점

- 자식 엔티티가 매핑한 컬럼은 모두 null 허용됨 (무결성 이슈 발
- 단일 테이블에 모두 저장하다보니 테이블이 커지면 조회성능이 느려질 수 있다.

구현클래스마다 테이블 전략

▼ 데이터베이스 설계자와 ORM 전문가 둘다 추천하지 않는 전략

치명적 단점 : 여러 자식 테이블을 함께 조회할 때 성능이 느림(UNION SQL)

쓰지마라!

@MappedSuperclass

공통 매핑정보가 필요할때 사용함!

(귀찮음을 줄이기 위해 사용)

```
// MappedSuperclass는 엔티티가 아님, 테이블과 매핑하지 않음, 조회/검색불가
// 직접 생성해서 사용할 일이 없으므로 추상클래스로 사용하는 것을 권장

@MappedSuperclass  // 매핑 속성정보만 상속하는 super class (인터페이스 같은 느낌?)
public class BaseEntity {
  private LocalDateTime createdDate;
  private LocalDateTime lastModifiedDate;
}

@Entity
public class Member extends BaseEntity {
  @Id
  @GeneratedValue
  @Column(name= "MEMBER_ID")
  private Long id;
}

// main에서 아래와 같이 사용가능

try{
  Member member = new Member();
  member.setCreatedDate(LocalDateTime.now());
  em.persist(member);

  tx.commit();
}catch(Exception e){
  tx.rollback();
}finally{
  em.close();
}
```

JPA의 데이터 타입 분류

▼ 엔티티 타입

- @Entity로 정의하는 객체
- 데이터가 변해도 식별자로 지속해서 추적 가능

▼ 값 타입

- int, Integer, String처럼 단순히 값으로 사용하는 자바 기본타입이나 객체
- 식별자가 없고 값만 있으므로 변경시 추적불가
- 값 타입에는 기본값 타입 / 임베디드 타입 / 컬렉션 값 타입 3가지가 있다.

값 타입

- 기본값 타입 (자바 기본 타입, 래퍼클래스, String)

생명주기를 엔티티에 의존한다.

값 타입은 절대 공유해서는 안된다. (회원 이름 변경시 다른 회원의 이름도 변경되어서는 안됨)

(기본타입은 절대 공유되지 않기때문에 안전 - 기본 타입은 항상 값을 복사하기 때문)

- 임베디드 타입

임베디드 타입과 컬렉션 값 타입은 JPA에서 정의해서 쓴다.

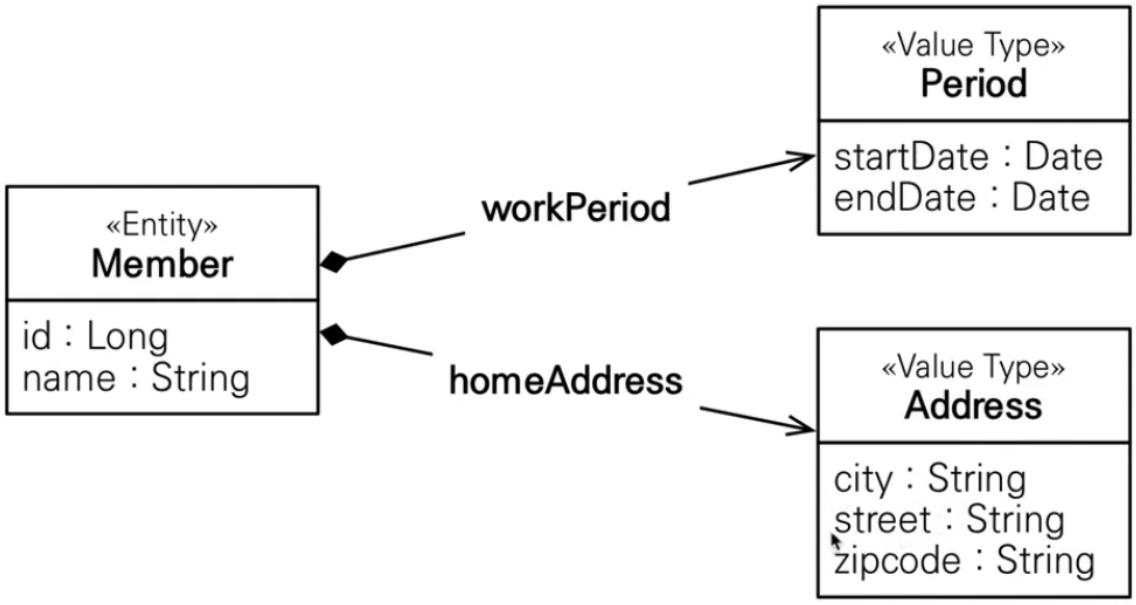
- 컬렉션 값 타입

컬렉션에 기본값타입이나 임베디드 타입을 넣어 쓸 수 있는 것을 컬렉션 값 타입이라 한다.

임베디드 타입

- 새로운 값 타입을 직접 정의할 수 있음
- JPA는 임베디드 타입(embedded type)이라 함
- 주로 기본 값 타입을 모아서 만들었기 때문에 복합 값 타입이라고도 함
- int, String과 같은 값 타입

그림으로 보면 바로 이해가 가능하다



임베디드 타입의 장점

- 재사용이 가능하다
- 높은 응집도

- Period.isWork()처럼 해당 값 타입만 사용하는 의미있는 메서드를 만들 수 있다.
- 임베디드 타입을 포함한 모든 값 타입은 값 타입을 소유한 엔티티에 생명주기를 의존함.

임베디드 타입 사용법!!!

- @Embeddable : 값 타입을 정의하는 곳에 표시하기!!!
- @Embedded : 값 타입을 사용하는 곳에 표시하기!!!
- **기본 생성자 필수!!!**

임베디드 타입은 엔티티의 값일 뿐

임베디드 타입을 사용하기 전과 후에 매핑하는 테이블은 같다!

객체와 테이블을 아주 세밀하게 매핑하는 것이 가능!

잘 설계한 ORM 어플리케이션은 매핑한 테이블의 수보다 클래스의 수가 더 많음!!

DTO에 담을것

error code(HTTP response code), message, data

에러 발생시 에러코드, 메세지만 포함해서 넘김 / 데이터는 x