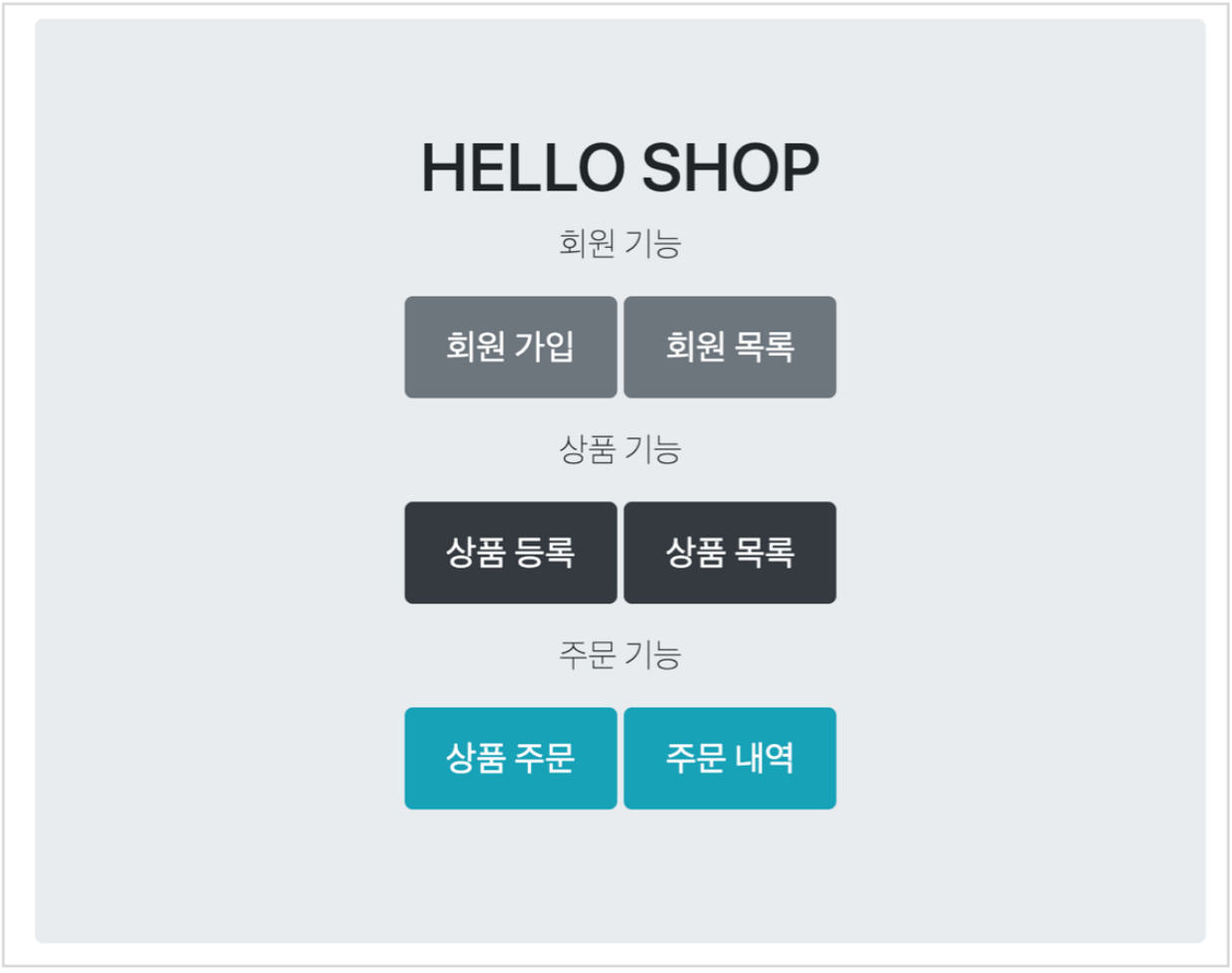


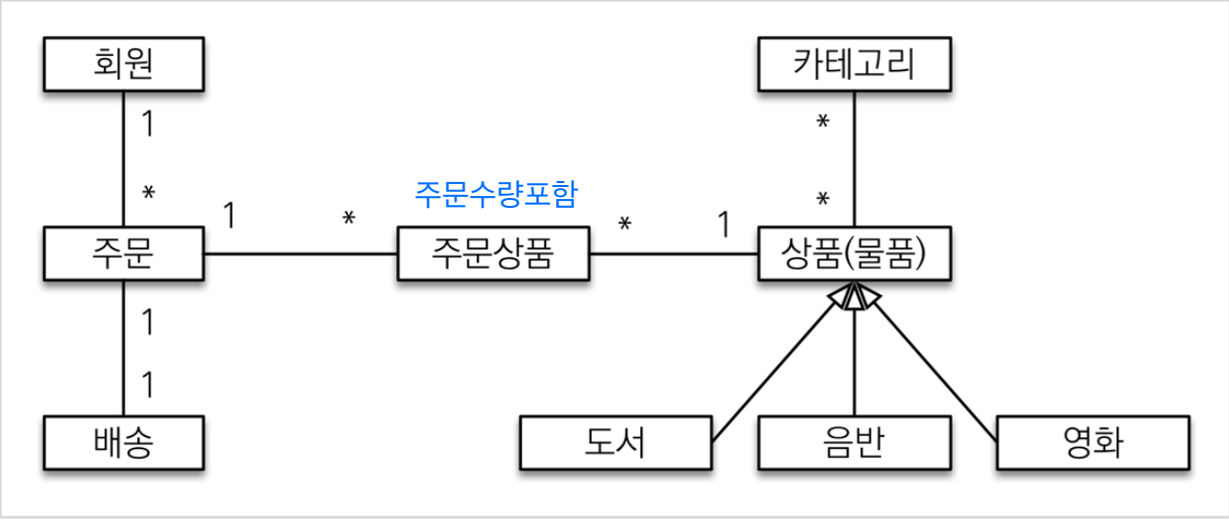
실제 동작하는 화면을 먼저 확인한다.

기능 목록

- 회원 기능
  - 회원 등록
  - 회원 조회
- 상품 기능
  - 상품 등록
  - 상품 수정
  - 상품 조회
- 주문 기능
  - 상품 주문
  - 주문 내역 조회
  - 주문 취소
- 기타 요구사항
  - 상품은 재고 관리가 필요하다.
  - 상품의 종류는 도서, 음반, 영화가 있다.
  - 상품을 카테고리로 구분할 수 있다.

- 상품 주문시 배송 정보를 입력할 수 있다.

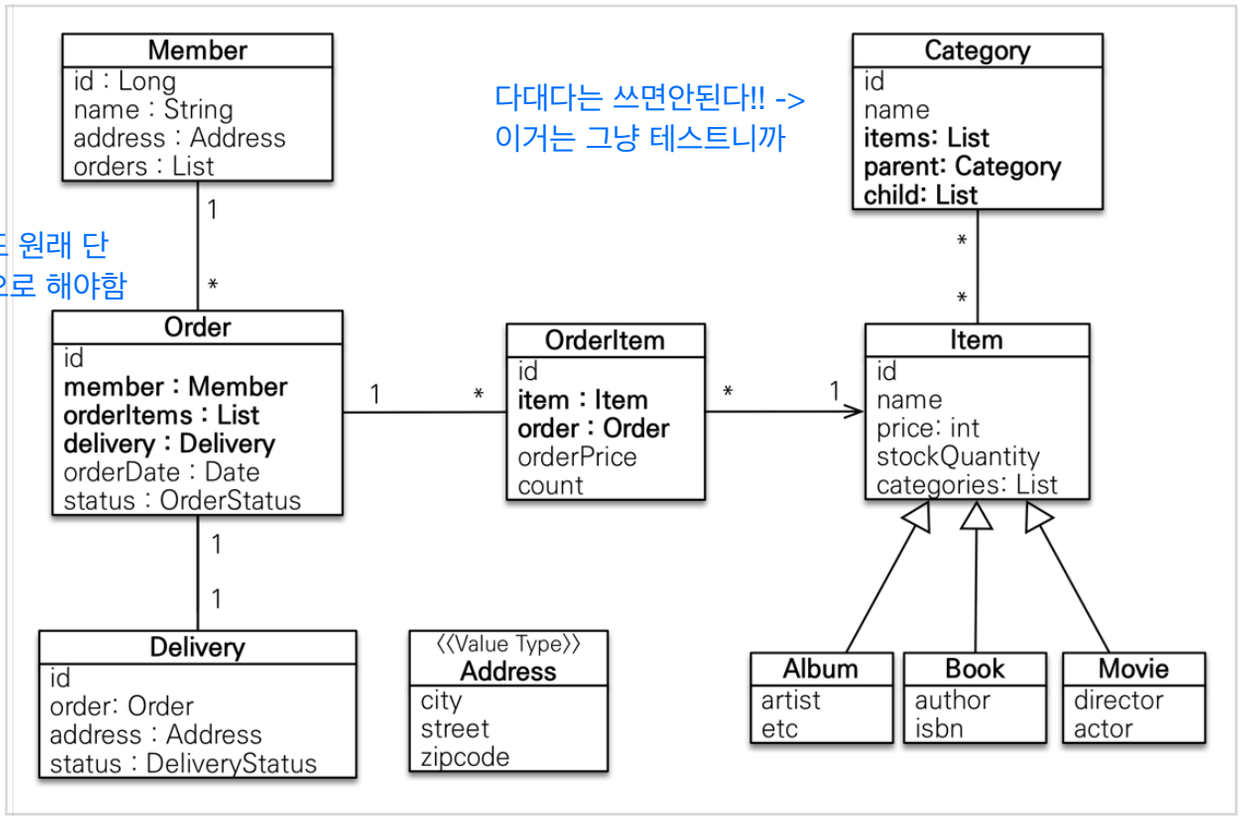
도메인 모델과 테이블 설계



회원, 주문, 상품의 관계: 회원은 여러 상품을 주문할 수 있다. 그리고 한 번 주문할 때 여러 상품을 선택할 수 있으므로 주문과 상품은 다대다 관계다. 하지만 이런 다대다 관계는 관계형 데이터베이스는 물론이고 엔티티에서도 거의 사용하지 않는다. 따라서 그림처럼 주문상품이라는 엔티티를 추가해서 다대다 관계를 일대다, 다대일 관계로 풀어냈다.

상품 분류: 상품은 도서, 음반, 영화로 구분되는데 상품이라는 공통 속성을 사용하므로 상속 구조로 표현했다.

회원 엔티티 분석



회원(Member): 이름과 임베디드 타입인 주소(`Address`), 그리고 주문(`orders`) 리스트를 가진다.

주문(Order): 한 번 주문시 여러 상품을 주문할 수 있으므로 주문과 주문상품(`OrderItem`)은 일대다 관계다. 주문은 상품을 주문한 회원과 배송 정보, 주문 날짜, 주문 상태(`status`)를 가지고 있다. 주문 상태는 열거형을 사용했는데 주문(`ORDER`), 취소(`CANCEL`)을 표현할 수 있다.

주문상품(OrderItem): 주문한 상품 정보와 주문 금액(`orderPrice`), 주문 수량(`count`) 정보를 가지고 있다. (보통 `OrderLine` , `LineItem` 으로 많이 표현한다.)

상품(Item): 이름, 가격, 재고수량(`stockQuantity`)을 가지고 있다. 상품을 주문하면 재고수량이 줄어든다. 상품의 종류로는 도서, 음반, 영화가 있는데 각각은 사용하는 속성이 조금씩 다르다.

배송(Delivery): 주문시 하나의 배송 정보를 생성한다. 주문과 배송은 일대일 관계다.

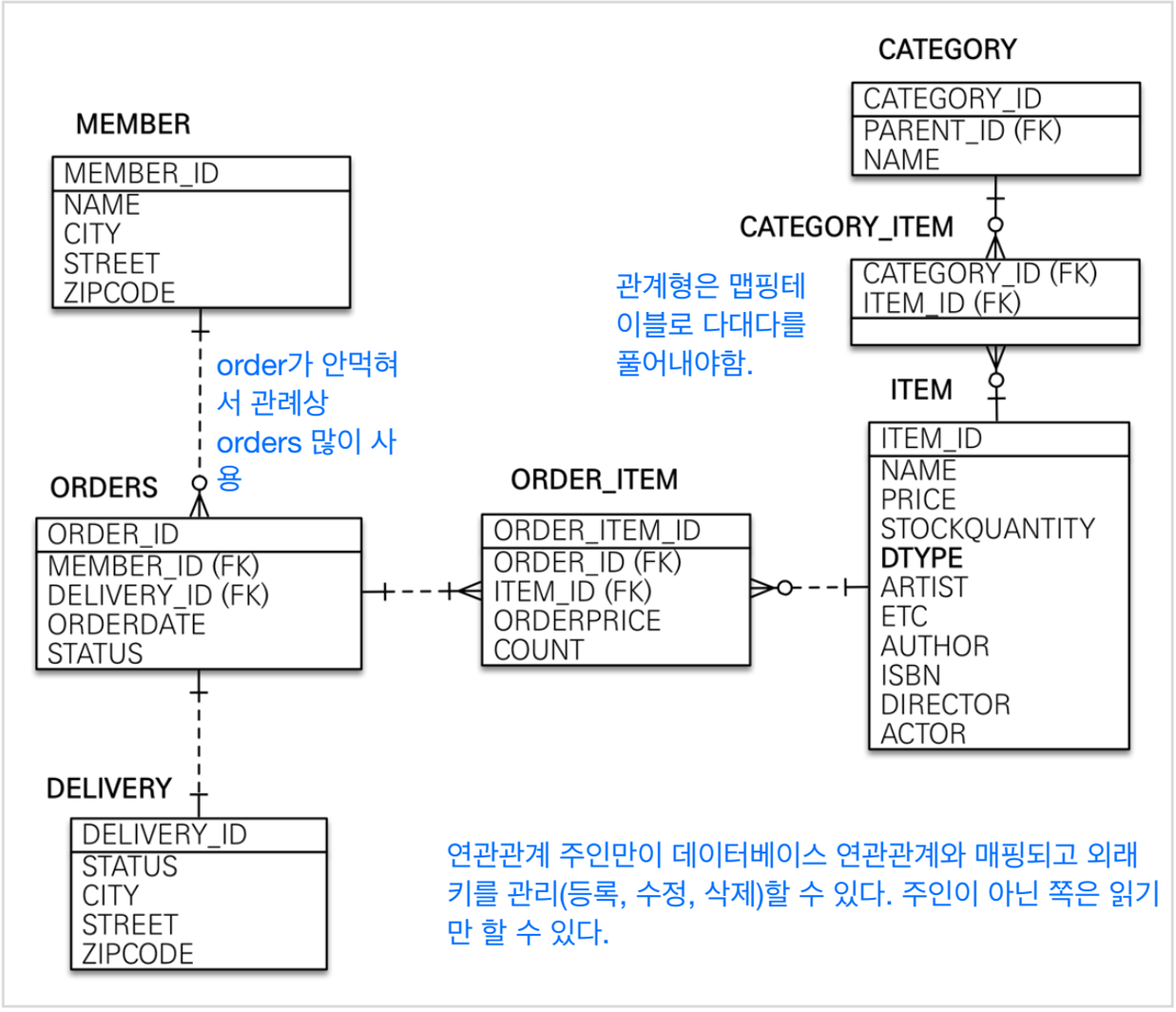
카테고리(Category): 상품과 다대다 관계를 맺는다. `parent` , `child` 로 부모, 자식 카테고리를 연결한다.

주소(Address): 값 타입(임베디드 타입)이다. 회원과 배송(Delivery)에서 사용한다.

> 참고: 회원 엔티티 분석 그림에서 Order와 Delivery가 단방향 관계로 잘못 그려져 있다. 양방향 관계가 맞다.

> 참고: 회원이 주문을 하기 때문에, 회원이 주문리스트를 가지는 것은 얼핏 보면 잘 설계한 것 같지만, 객체 세상은 실제 세계와는 다르다. 실무에서는 회원이 주문을 참조하지 않고, 주문이 회원을 참조하는 것으로 충분하다. 여기서는 일대다, 다대일의 양방향 연관관계를 설명하기 위해서 추가했다.

회원 테이블 분석



MEMBER: 회원 엔티티의 `Address` 임베디드 타입 정보가 회원 테이블에 그대로 들어갔다. 이것은 `DELIVERY` 테이블도 마찬가지다.

ITEM: 앨범, 도서, 영화 타입을 통합해서 하나의 테이블로 만들었다. `DTYPE` 컬럼으로 타입을 구분한다.

> 참고: 테이블명이 `ORDER` 가 아니라 `ORDERS` 인 것은 데이터베이스가 `order by` 때문에 예약어로 잡고 있는 경우가 많다. 그래서 관례상 `ORDERS` 를 많이 사용한다.

> **참고: 실제 코드에서는 DB에 소문자 + _(언더스코어) 스타일을 사용하겠다.**
> 데이터베이스 테이블명, 컬럼명에 대한 관례는 회사마다 다르다. 보통은 대문자 + _(언더스코어)나 소문자 + _(언더스코어) 방식 중에 하나를 지정해서 일관성 있게 사용한다. 강의에서 설명할 때는 객체와 차이를 나

타내기 위해 데이터베이스 테이블, 컬럼명은 대문자를 사용했지만, **실제 코드에서는 소문자 + _(언더스코어) 스타일을 사용하겠다.**

연관관계 매핑 분석

회원과 주문: 일대다 , 다대일의 양방향 관계다. 따라서 연관관계의 주인을 정해야 하는데, 외래 키가 있는 주문을 연관관계의 주인으로 정하는 것이 좋다. 그러므로 `Order.member` 를 `ORDERS.MEMBER_ID` 외래 키와 매핑한다.

주인쪽으로 값을 세팅해야 값을 변경가능 반대쪽은 단순히 조회용

주문상품과 주문: 다대일 양방향 관계다. 외래 키가 주문상품에 있으므로 주문상품이 연관관계의 주인이다. 그러므로 `OrderItem.order` 를 `ORDER_ITEM.ORDER_ID` 외래 키와 매핑한다.

주문상품과 상품: 다대일 단방향 관계다. `OrderItem.item` 을 `ORDER_ITEM.ITEM_ID` 외래 키와 매핑한다.

주문과 배송: 일대일 양방향 관계다. `Order.delivery` 를 `ORDERS.DELIVERY_ID` 외래 키와 매핑한다.

카테고리와 상품: `@ManyToMany` 를 사용해서 매핑한다.(실무에서 @ManyToMany는 사용하지 말자. 여기서는 다대다 관계를 예제로 보여주기 위해 추가했을 뿐이다)

참고: 외래 키가 있는 곳을 연관관계의 주인으로 정해라.
연관관계의 주인은 단순히 외래 키를 누가 관리하냐의 문제이지 비즈니스상 우위에 있다고 주인으로 정하면 안된다.. 예를 들어서 자동차와 바퀴가 있으면, 일대다 관계에서 항상 다쪽에 외래 키가 있으므로 외래 키가 있는 바퀴를 연관관계의 주인으로 정하면 된다. 물론 자동차를 연관관계의 주인으로 정하는 것이 불가능 한 것은 아니지만, 자동차를 연관관계의 주인으로 정하면 자동차가 관리하지 않는 바퀴 테이블의 외래 키 값이 업데이트 되므로 관리와 유지보수가 어렵고, 추가적으로 별도의 업데이트 쿼리가 발생하는 성능 문제도 있다. 자세한 내용은 JPA 기본편을 참고하자.

항상 외래키가 있는곳이 주인-> 정석이다!

엔티티 클래스 개발

객체가 양방향이면 연관관계 주인을 정해야한다!!

- 예제에서는 설명을 쉽게하기 위해 엔티티 클래스에 Getter, Setter를 모두 열고, 최대한 단순하게 설계
- 실무에서는 가급적 Getter는 열어두고, Setter는 꼭 필요한 경우에만 사용하는 것을 추천

영한님은 실무에서 Setter를 닫는다고 함

참고: 이론적으로 Getter, Setter 모두 제공하지 않고, 꼭 필요한 별도의 메서드를 제공하는게 가장 이상적이다. 하지만 실무에서 엔티티의 데이터는 조회할 일이 너무 많으므로, Getter의 경우 모두 열어두는 것이 편리하다. Getter는 아무리 호출해도 호출 하는 것 만으로 어떤 일이 발생하지는 않는다. 하지만 Setter는 문제가 다르다. Setter를 호출하면 데이터가 변한다. Setter를 막 열어두면 가까운 미래에 엔티티에가 도대체 왜 변경되는지 추적하기 점점 힘들어진다. 그래서 엔티티를 변경할 때는 Setter 대신에 변경 지점이 명확

@Embedded 설명-https://velog.io/@conatuseus/JPA-%EC%9E%84%EB%B2%A0%EB%94%94%EB%93%9C-%ED%83%80%EC%9E%85embedded-type-8ak3ygq8wo

> 하도록 변경을 위한 비즈니스 메서드를 별도로 제공해야 한다.

회원 엔티티

```
package jpabook.jpashop.domain;

import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Getter @Setter
public class Member {

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

    private String name;

    @Embedded
    private Address address;

    서로의 관계
    @OneToMany(mappedBy = "member")
    private List<Order> orders = new ArrayList<>();

}
```

mappedBy->이게 있으면 member필드에 의해 맵핑된거야! 주인이 아닐때 씀

> 참고: 엔티티의 식별자는 `id`를 사용하고 PK 컬럼명은 `member_id`를 사용했다. 엔티티는 타입(여기서는 `Member`)이 있으므로 `id` 필드만으로 쉽게 구분할 수 있다. 테이블은 타입이 없으므로 구분이 어렵다. 그리고 테이블은 관례상 `테이블명 + id`를 많이 사용한다. 참고로 객체에서 `id` 대신에 `memberId`를 사용해도 된다. 중요한 것은 일관성이다.

주문 엔티티

```
package jpabook.jpashop.domain;

import lombok.Getter;
```

```java
import lombok.Setter;

import javax.persistence.*;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "orders")
@Getter @Setter
public class Order {

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

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "member_id") FK이름
    private Member member; //주문 회원

                                    cascadeALL->엔티티는 각각 persist해야하는데
                                    order하면 알아서 다 persist해줌
    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
    private List<OrderItem> orderItems = new ArrayList<>();

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "delivery_id") 외래키를 연관관계의 주인으로 관리
    private Delivery delivery; //배송정보

    private LocalDateTime orderDate; //주문시간

    @Enumerated(EnumType.STRING)
    private OrderStatus status; //주문상태 [ORDER, CANCEL]

    //==연관관계 메서드==//
    public void setMember(Member member) {
        this.member = member;
        member.getOrders().add(this);
    }

    public void addOrderItem(OrderItem orderItem) {
```

```
        orderItems.add(orderItem);
        orderItem.setOrder(this);
    }

    public void setDelivery(Delivery delivery) {
        this.delivery = delivery;
        delivery.setOrder(this);
    }

}
```

주문상태

```
package jpabook.jpashop.domain;

public enum OrderStatus {
    ORDER, CANCEL
}
```

주문상품 엔티티

```
package jpabook.jpashop.domain;

import lombok.Getter;
import lombok.Setter;

import jpabook.jpashop.domain.item.Item;
import javax.persistence.*;

@Entity
@Table(name = "order_item")
@Getter @Setter
public class OrderItem {

    @Id @GeneratedValue
    @Column(name = "order_item_id")
```

```
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "item_id")
    private Item item;      //주문 상품

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "order_id")
    private Order order;    //주문

    private int orderPrice; //주문 가격
    private int count;      //주문 수량

}
```

상품 엔티티

```
package jpabook.jpashop.domain.item;

import lombok.Getter;
import lombok.Setter;

import jpabook.jpashop.domain.Category;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "dtype")
@Getter @Setter
public abstract class Item {

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

    private String name;
```

상속관계맵핑은 상속관계전략을 정해줘야 함.SINGLE_TABLE->한테이블에 다 때려박는것.

@Discrim~기본값은 클래스네임임 중요하진 않은 듯

```
    private int price;
    private int stockQuantity;

    @ManyToMany(mappedBy = "items")
    private List<Category> categories = new ArrayList<Category>();

}
```

상품 - 도서 엔티티

```
package jpabook.jpashop.domain.item;

import lombok.Getter;
import lombok.Setter;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("B")
@Getter @Setter
public class Book extends Item {

    private String author;
    private String isbn;

}
```

상품 - 음반 엔티티

```
package jpabook.jpashop.domain.item;

import lombok.Getter;
import lombok.Setter;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
```

```
@Entity
@DiscriminatorValue("A")
@Getter @Setter
public class Album extends Item {

    private String artist;
    private String etc;

}
```

상품 - 영화 엔티티

```
package jpabook.jpashop.domain.item;

import lombok.Getter;
import lombok.Setter;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("M")
@Getter @Setter
public class Movie extends Item {

    private String director;
    private String actor;

}
```

배송 엔티티

```
package jpabook.jpashop.domain;

import lombok.Getter;
import lombok.Setter;
```

```java
import javax.persistence.*;

@Entity
@Getter @Setter
public class Delivery {

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

    @OneToOne(mappedBy = "delivery", fetch = FetchType.LAZY)
    private Order order;

    @Embedded
    private Address address;

    @Enumerated(EnumType.STRING)  ORDINAL은 숫자대로인데 장애나기가 쉬움
    private DeliveryStatus status; //ENUM [READY(준비), COMP(배송)]

}
```

배송 상태

```java
package jpabook.jpashop.domain;

public enum DeliveryStatus {
    READY, COMP
}
```

카테고리 엔티티

```java
package jpabook.jpashop.domain;

import lombok.Getter;
import lombok.Setter;
```

```java
import jpabook.jpashop.domain.item.Item;

import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Getter @Setter
public class Category {

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

    private String name;

                                        다대다는 안좋음 이건 예제라 하는것
    @ManyToMany
    @JoinTable(name = "category_item",  다대다에서 중간테이블을 연결해줘야함
            joinColumns = @JoinColumn(name = "category_id"),
            inverseJoinColumns = @JoinColumn(name = "item_id"))
    private List<Item> items = new ArrayList<>();

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "parent_id")
    private Category parent;

    @OneToMany(mappedBy = "parent")
    private List<Category> child = new ArrayList<>();

    //==연관관계 메서드==//
    public void addChildCategory(Category child) {
        this.child.add(child);
        child.setParent(this);
    }

}
```

> 참고: 실무에서는 `@ManyToMany`를 사용하지 말자

`@ManyToMany`는 편리한 것 같지만, 중간 테이블(`CATEGORY_ITEM`)에 컬럼을 추가할 수 없고, 세밀하게 쿼리를 실행하기 어렵기 때문에 실무에서 사용하기에는 한계가 있다. 중간 엔티티(`CategoryItem`를 만들고 `@ManyToOne`, `@OneToMany`로 매핑해서 사용하자. 정리하면 대다대 매핑을 일대다, 다대일 매핑으로 풀어내서 사용하자.

주소 값 타입

```
package jpabook.jpashop.domain;

import lombok.Getter;
import lombok.Setter;

import javax.persistence.Embeddable;

@Embeddable JPA의 내장타입->어딘가에 내장이 될 수 있다.
@Getter
public class Address {

    private String city;
    private String street;
    private String zipcode;

    protected Address() {
    }                          protected까진 허용해줌

    public Address(String city, String street, String zipcode) {
        this.city = city;
        this.street = street;
        this.zipcode = zipcode;
    }
}
```

참고: 값 타입은 변경 불가능하게 설계해야 한다.
`@Setter`를 제거하고, 생성자에서 값을 모두 초기화해서 변경 불가능한 클래스를 만들자. JPA 스펙상 엔티티나 임베디드 타입(`@Embeddable`)은 자바 기본 생성자(default constructor)를 `public` 또는 `protected`로 설정해야 한다. `public`으로 두는 것 보다는 `protected`로 설정하는 것이 그나마 더 안전하다.
JPA가 이런 제약을 두는 이유는 JPA 구현 라이브러리가 객체를 생성할 때 리플랙션 같은 기술을 사용할 수

있도록 지원해야 하기 때문이다.

Application이 엔티티와 다른위치에 있으면 테이블이 생성되지 않는다.패키지위치를 같이해주거나 상위에 놓자!

일대일 관계에서는 access가 많은 엔티티에 FK를 둔다.

엔티티 설계시 주의점

엔티티에는 가급적 Setter를 사용하지 말자

Setter가 모두 열려있다. 변경 포인트가 너무 많아서, 유지보수가 어렵다. 나중에 리펙토링으로 Setter 제거

모든 연관관계는 지연로딩으로 설정! 엄청중요하다!!!

- 즉시로딩(`EAGER`)은 예측이 어렵고, 어떤 SQL이 실행될지 추적하기 어렵다. 특히 JPQL을 실행할 때 N+1 문제가 자주 발생한다. 절대쓰면안됨
- 실무에서 모든 연관관계는 지연로딩(`LAZY`)으로 설정해야 한다.
- 연관된 엔티티를 함께 DB에서 조회해야 하면, fetch join 또는 엔티티 그래프 기능을 사용한다.
- @XToOne(OneToOne, ManyToOne) 관계는 기본이 즉시로딩이므로 직접 지연로딩으로 설정해야 한다. command+shift+F

컬렉션은 필드에서 초기화 하자.

컬렉션은 필드에서 바로 초기화 하는 것이 안전하다.

선언하면서 new로 !

진짜 웬만하면 바꾸지말자!

- `null` 문제에서 안전하다.
- 하이버네이트는 엔티티를 영속화 할 때, 컬랙션을 감싸서 하이버네이트가 제공하는 내장 컬렉션으로 변경한다. 만약 `getOrders()` 처럼 임의의 메서드에서 컬력션을 잘못 생성하면 하이버네이트 내부 메커니즘에 문제가 발생할 수 있다. 따라서 필드레벨에서 생성하는 것이 가장 안전하고, 코드도 간결하다.

```
Member member = new Member();
System.out.println(member.getOrders().getClass());
em.persist(team);
System.out.println(member.getOrders().getClass());

//출력 결과
class java.util.ArrayList
class org.hibernate.collection.internal.PersistentBag
```

테이블, 컬럼명 생성 전략

스프링 부트에서 하이버네이트 기본 매핑 전략을 변경해서 실제 테이블 필드명은 다름

- https://docs.spring.io/spring-boot/docs/2.1.3.RELEASE/reference/htmlsingle/#howto-configure-hibernate-naming-strategy
- http://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#naming

하이버네이트 기존 구현: 엔티티의 필드명을 그대로 테이블의 컬럼명으로 사용
(`SpringPhysicalNamingStrategy`)

스프링 부트 신규 설정 (엔티티(필드) ➜ 테이블(컬럼))

1. 카멜 케이스 ➜ 언더스코어(memberPoint ➜ member_point)
2. .(점) ➜ _(언더스코어)
3. 대문자 ➜ 소문자

적용 2 단계

1. 논리명 생성: 명시적으로 컬럼, 테이블명을 직접 적지 않으면 ImplicitNamingStrategy 사용
   `spring.jpa.hibernate.naming.implicit-strategy` : 테이블이나, 컬럼명을 명시하지 않을 때 논리명 적용,
2. 물리명 적용:
   `spring.jpa.hibernate.naming.physical-strategy` : 모든 논리명에 적용됨, 실제 테이블에 적용 (username ➜ usernm 등으로 회사 룰로 바꿀 수 있음)

스프링 부트 기본 설정

```
spring.jpa.hibernate.naming.implicit-strategy:
org.springframework.boot.orm.jpa.hibernate.SpringImplicitNamingStrategy
spring.jpa.hibernate.naming.physical-strategy:
org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy
```

애플리케이션 구현 준비

구현 요구사항