상품 기능 테스트

상품 테스트는 회원 테스트와 비슷하므로 생략

주문 도메인 개발

제일중요함!!

구현 기능

- 상품 주문
- 주문 내역 조회
- 주문 취소

순서

- 주문 엔티티, 주문상품 엔티티 개발
- 주문 리포지토리 개발
- 주문 서비스 개발
- 주문 검색 기능 개발
- 주문 기능 테스트

주문, 주문상품 엔티티 개발

주문 엔티티 개발

주문 엔티티 코드

```
package jpabook.jpashop.domain;

import lombok.Getter;
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")
private Member member; //주문 회원

@OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
private List<OrderItem> orderItems = new ArrayList<>();

@OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@JoinColumn(name = "delivery_id")
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);
}

//==생성 메서드==//
public static Order createOrder(Member member, Delivery delivery,
```

```
OrderItem... orderItems) {
        Order order = new Order();
        order.setMember(member);
        order.setDelivery(delivery);
        for (OrderItem orderItem : orderItems) {
            order.addOrderItem(orderItem);
        }
        order.setStatus(OrderStatus.ORDER);
        order.setOrderDate(LocalDateTime.now());
        return order;
    }

    //==비즈니스 로직==//
    /** 주문 취소 */
    public void cancel() {
        if (delivery.getStatus() == DeliveryStatus.COMP) {
            throw new IllegalStateException("이미 배송완료된 상품은 취소가 불가능합니
다.");
        }

        this.setStatus(OrderStatus.CANCEL);
        for (OrderItem orderItem : orderItems) {
            orderItem.cancel();
        }
    }

    //==조회 로직==//
    /** 전체 주문 가격 조회 */
    public int getTotalPrice() {
        int totalPrice = 0;
        for (OrderItem orderItem : orderItems) {
            totalPrice += orderItem.getTotalPrice();
        }
        return totalPrice;
    }

}
```

상태와 현재시간으로 세팅

기능 설명

- **생성 메서드**(`createOrder()`): 주문 엔티티를 생성할 때 사용한다. 주문 회원, 배송정보, 주문상품의 정보를 받아서 실제 주문 엔티티를 생성한다.
- **주문 취소**(`cancel()`): 주문 취소시 사용한다. 주문 상태를 취소로 변경하고 주문상품에 주문 취소를 알린다. 만약 이미 배송을 완료한 상품이면 주문을 취소하지 못하도록 예외를 발생시킨다.
- **전체 주문 가격 조회:** 주문 시 사용한 전체 주문 가격을 조회한다. 전체 주문 가격을 알려면 각각의 주문상품 가격을 알아야 한다. 로직을 보면 연관된 주문상품들의 가격을 조회해서 더한 값을 반환한다. (실무에서는 주로 주문에 전체 주문 가격 필드를 두고 역정규화 한다.)

주문상품 엔티티 개발

주문상품 엔티티 코드

```
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

    //==생성 메서드==//
    public static OrderItem createOrderItem(Item item, int orderPrice, int
count) {
        OrderItem orderItem = new OrderItem();
        orderItem.setItem(item);
        orderItem.setOrderPrice(orderPrice);
        orderItem.setCount(count);

        item.removeStock(count);
        return orderItem;
    }

    //==비즈니스 로직==//
    /** 주문 취소 */
    public void cancel() {
        getItem().addStock(count);
    }

    //==조회 로직==//
    /** 주문상품 전체 가격 조회 */
    public int getTotalPrice() {
        return getOrderPrice() * getCount();
    }

}
```

기능 설명

- **생성 메서드**(`createOrderItem()`): 주문 상품, 가격, 수량 정보를 사용해서 주문상품 엔티티를 생성한다. 그리고 `item.removeStock(count)`를 호출해서 주문한 수량만큼 상품의 재고를 줄인다.
- **주문 취소**(`cancel()`): `getItem().addStock(count)`를 호출해서 취소한 주문 수량만큼 상품의 재고를 증가시킨다.
- **주문 가격 조회**(`getTotalPrice()`): 주문 가격에 수량을 곱한 값을 반환한다.

주문 리포지토리 개발

주문 리포지토리 코드

```
package jpabook.jpashop.repository;

import jpabook.jpashop.domain.Order;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Repository;

import javax.persistence.EntityManager;

@Repository
@RequiredArgsConstructor
public class OrderRepository {

    private final EntityManager em;

    public void save(Order order) {
        em.persist(order);
    }

    public Order findOne(Long id) {
        return em.find(Order.class, id);
    }

//    public List<Order> findAll(OrderSearch orderSearch) { ... }
}
```

주문 리포지토리에는 주문 엔티티를 저장하고 검색하는 기능이 있다. 마지막의 `findAll(OrderSearch orderSearch)` 메서드는 조금 뒤에 있는 주문 검색 기능에서 자세히 알아보자.

주문 서비스 개발

주문 서비스 코드

```java
package jpabook.jpashop.service;

import jpabook.jpashop.domain.Delivery;
import jpabook.jpashop.domain.Member;
import jpabook.jpashop.domain.Order;
import jpabook.jpashop.domain.OrderItem;
import jpabook.jpashop.domain.item.Item;
import jpabook.jpashop.repository.ItemRepository;
import jpabook.jpashop.repository.MemberRepository;
import jpabook.jpashop.repository.OrderRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
@Transactional(readOnly = true)
@RequiredArgsConstructor
public class OrderService {

    private final MemberRepository memberRepository;
    private final OrderRepository orderRepository;
    private final ItemRepository itemRepository;

    /** 주문 */
    @Transactional
    public Long order(Long memberId, Long itemId, int count) {

        //엔티티 조회
        Member member = memberRepository.findOne(memberId);
        Item item = itemRepository.findOne(itemId);

        //배송정보 생성
        Delivery delivery = new Delivery();
        delivery.setAddress(member.getAddress());
        delivery.setStatus(DeliveryStatus.READY);

        //주문상품 생성
        OrderItem orderItem = OrderItem.createOrderItem(item, item.getPrice(),
```

cascade옵션으로 orderitem,delivery가 저절로 persist가 된다.

```
count);
        //주문 생성
        Order order = Order.createOrder(member, delivery, orderItem);

                        new Order();할 수있으니 생성자를 protected로 막아
        //주문 저장       주자.
        orderRepository.save(order);
        return order.getId();
    }

    /** 주문 취소 */
    @Transactional
    public void cancelOrder(Long orderId) {

        //주문 엔티티 조회
        Order order = orderRepository.findOne(orderId);
        //주문 취소
        order.cancel();
    }

    /** 주문 검색 */
/*
    public List<Order> findOrders(OrderSearch orderSearch) {
        return orderRepository.findAll(orderSearch);
    }
*/
}
```

엔티티안에서 데이터를 바꾸면 JPA가 알아서 데이터베이스에서 업데이트 쿼리가 들어간다 ->JPA가 킹임

주문 서비스는 주문 엔티티와 주문 상품 엔티티의 비즈니스 로직을 활용해서 주문, 주문 취소, 주문 내역 검색 기능을 제공한다.

참고: 예제를 단순화하려고 한 번에 하나의 상품만 주문할 수 있다.

- **주문**(`order()`): 주문하는 회원 식별자, 상품 식별자, 주문 수량 정보를 받아서 실제 주문 엔티티를 생성한 후 저장한다.
- **주문 취소**(`cancelOrder()`): 주문 식별자를 받아서 주문 엔티티를 조회한 후 주문 엔티티에 주문 취소를 요청한다.
- **주문 검색**(`findOrders()`): `OrderSearch`라는 검색 조건을 가진 객체로 주문 엔티티를 검색한다. 자세한 내용은 다음에 나오는 주문 검색 기능에서 알아보자.

JPA는 도메인 모델패턴 스타일임.

> 참고: 주문 서비스의 주문과 주문 취소 메서드를 보면 비즈니스 로직 대부분이 엔티티에 있다. 서비스 계층은 단순히 엔티티에 필요한 요청을 위임하는 역할을 한다. 이처럼 엔티티가 비즈니스 로직을 가지고 객체 지향의 특성을 적극 활용하는 것을 도메인 모델 패턴(http://martinfowler.com/eaaCatalog/domainModel.html)이라 한다. 반대로 엔티티에는 비즈니스 로직이 거의 없고 서비스 계층에서 대부분의 비즈니스 로직을 처리하는 것을 트랜잭션 스크립트 패턴(http://martinfowler.com/eaaCatalog/transactionScript.html)이라 한다.

주문 기능 테스트

테스트 요구사항

- 상품 주문이 성공해야 한다.
- 상품을 주문할 때 재고 수량을 초과하면 안 된다.
- 주문 취소가 성공해야 한다.

상품 주문 테스트 코드

```
package jpabook.jpashop.service;

import jpabook.jpashop.domain.Address;
import jpabook.jpashop.domain.Member;
import jpabook.jpashop.domain.Order;
import jpabook.jpashop.domain.OrderStatus;
import jpabook.jpashop.domain.item.Book;
import jpabook.jpashop.domain.item.Item;
import jpabook.jpashop.exception.NotEnoughStockException;
import jpabook.jpashop.repository.OrderRepository;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Transactional;
```

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;

@RunWith(SpringRunner.class)
@SpringBootTest
@Transactional
public class OrderServiceTest {

    @PersistenceContext
    EntityManager em;

    @Autowired OrderService orderService;
    @Autowired OrderRepository orderRepository;

    @Test
    public void 상품주문() throws Exception {

        //Given
        Member member = createMember();
        Item item = createBook("시골 JPA", 10000, 10); //이름, 가격, 재고
        int orderCount = 2;

        //When
        Long orderId = orderService.order(member.getId(), item.getId(),
orderCount);

        //Then
        Order getOrder = orderRepository.findOne(orderId);

        assertEquals("상품 주문시 상태는 ORDER",OrderStatus.ORDER,
getOrder.getStatus());
        assertEquals("주문한 상품 종류 수가 정확해야 한다.",1,
getOrder.getOrderItems().size());
        assertEquals("주문 가격은 가격 * 수량이다.", 10000 * 2,
getOrder.getTotalPrice());
        assertEquals("주문 수량만큼 재고가 줄어야 한다.",8, item.getStockQuantity());
```

```
    }

    @Test(expected = NotEnoughStockException.class)
    public void 상품주문_재고수량초과() throws Exception {
        //...
    }



    @Test
    public void 주문취소() {
        //...
    }

    private Member createMember() {
        Member member = new Member();
        member.setName("회원1");
        member.setAddress(new Address("서울", "강가", "123-123"));
        em.persist(member);
        return member;
    }

    private Book createBook(String name, int price, int stockQuantity) {
        Book book = new Book();
        book.setName(name);
        book.setStockQuantity(stockQuantity);
        book.setPrice(price);
        em.persist(book);
        return book;
    }
}
```

상품주문이 정상 동작하는지 확인하는 테스트다. Given 절에서 테스트를 위한 회원과 상품을 만들고 When 절에서 실제 상품을 주문하고 Then 절에서 주문 가격이 올바른지, 주문 후 재고 수량이 정확히 줄었는지 검증한다.

재고 수량 초과 테스트

재고 수량을 초과해서 상품을 주문해보자. 이때는 `NotEnoughStockException` 예외가 발생해야 한다.

재고 수량 초과 테스트 코드

```
@Test(expected = NotEnoughStockException.class)
public void 상품주문_재고수량초과() throws Exception {

    //Given
    Member member = createMember();
    Item item = createBook("시골 JPA", 10000, 10); //이름, 가격, 재고

    int orderCount = 11; //재고보다 많은 수량

    //When
    orderService.order(member.getId(), item.getId(), orderCount);

    //Then
    fail("재고 수량 부족 예외가 발생해야 한다.");
}
```

코드를 보면 재고는 10권인데 `orderCount = 11`로 재고보다 1권 더 많은 수량을 주문했다. 주문 초과로 다음 로직에서 예외가 발생한다.

```
public abstract class Item {

    //...

    public void removeStock(int orderQuantity) {
        int restStock = this.stockQuantity - orderQuantity;
        if (restStock < 0) {
            throw new NotEnoughStockException("need more stock");
        }
        this.stockQuantity = restStock;
    }
}
```

주문 취소 테스트

주문 취소 테스트 코드를 작성하자. 주문을 취소하면 그만큼 재고가 증가해야 한다.

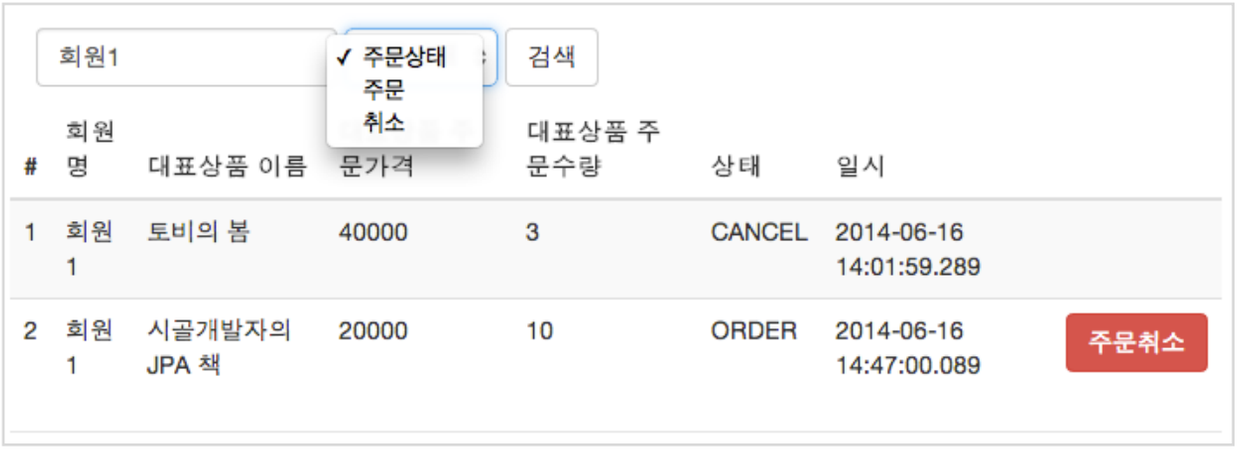
주문 취소 테스트 코드

```
@Test
public void 주문취소() {

    //Given
    Member member = createMember();
    Item item = createBook("시골 JPA", 10000, 10); //이름, 가격, 재고
    int orderCount = 2;

    Long orderId = orderService.order(member.getId(), item.getId(),
orderCount);

    //When
    orderService.cancelOrder(orderId);

    //Then
    Order getOrder = orderRepository.findOne(orderId);

    assertEquals("주문 취소시 상태는 CANCEL 이다.",OrderStatus.CANCEL,
getOrder.getStatus());
    assertEquals("주문이 취소된 상품은 그만큼 재고가 증가해야 한다.", 10,
item.getStockQuantity());
}
```

주문을 취소하려면 먼저 주문을 해야 한다. Given 절에서 주문하고 When 절에서 해당 주문을 취소했다. Then 절에서 주문상태가 주문 취소 상태인지(`CANCEL`), 취소한 만큼 재고가 증가했는지 검증한다.

주문 검색 기능 개발

JPA에서 **동적 쿼리**를 어떻게 해결해야 하는가?



*검색 조건 파라미터 `OrderSearch` *

```
package jpabook.jpashop.domain;

public class OrderSearch {

    private String memberName;      //회원 이름
    private OrderStatus orderStatus;//주문 상태[ORDER, CANCEL]

    //Getter, Setter
}
```

검색을 추가한 주문 리포지토리 코드

```
package jpabook.jpashop.repository;

@Repository
public class OrderRepository {

    @PersistenceContext
    EntityManager em;

    public void save(Order order) {
        em.persist(order);
    }

    public Order findOne(Long id) {
```

```
        return em.find(Order.class, id);
    }

    public List<Order> findAll(OrderSearch orderSearch) {
        //... 검색 로직
    }
}
```

`findAll(OrderSearch orderSearch)` 메서드는 검색 조건에 동적으로 쿼리를 생성해서 주문 엔티티를 조회한다.

JPQL로 처리

```
public List<Order> findAllByString(OrderSearch orderSearch) {
    //language=JPAQL
    String jpql = "select o From Order o join o.member m";
    boolean isFirstCondition = true;

    //주문 상태 검색
    if (orderSearch.getOrderStatus() != null) {
        if (isFirstCondition) {
            jpql += " where";
            isFirstCondition = false;
        } else {
            jpql += " and";
        }
        jpql += " o.status = :status";
    }
    //회원 이름 검색
    if (StringUtils.hasText(orderSearch.getMemberName())) {
        if (isFirstCondition) {
            jpql += " where";
            isFirstCondition = false;
        } else {
            jpql += " and";
        }
        jpql += " m.name like :name";
    }
```

JPQL을 문자로 처리는 정말정말 비효율적->안씀

```
    TypedQuery<Order> query = em.createQuery(jpql, Order.class)
            .setMaxResults(1000); //최대 1000건

    if (orderSearch.getOrderStatus() != null) {
        query = query.setParameter("status", orderSearch.getOrderStatus());
    }
    if (StringUtils.hasText(orderSearch.getMemberName())) {
        query = query.setParameter("name", orderSearch.getMemberName());
    }

    return query.getResultList();
}
```

JPQL 쿼리를 문자로 생성하기는 번거롭고, 실수로 인한 버그가 충분히 발생할 수 있다.

JPA Criteria로 처리 이것도 권장하지 않음

```
public List<Order> findAllByCriteria(OrderSearch orderSearch) {

    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Order> cq = cb.createQuery(Order.class);
    Root<Order> o = cq.from(Order.class);
    Join<Order, Member> m = o.join("member", JoinType.INNER); //회원과 조인

    List<Predicate> criteria = new ArrayList<>();

    //주문 상태 검색
    if (orderSearch.getOrderStatus() != null) {
        Predicate status = cb.equal(o.get("status"),
orderSearch.getOrderStatus());
        criteria.add(status);
    }
    //회원 이름 검색
    if (StringUtils.hasText(orderSearch.getMemberName())) {
        Predicate name =
                cb.like(m.<String>get("name"), "%" +
orderSearch.getMemberName() + "%");
        criteria.add(name);
```

```
    }

    cq.where(cb.and(criteria.toArray(new Predicate[criteria.size()])));
    TypedQuery<Order> query = em.createQuery(cq).setMaxResults(1000); //최대
1000건
    return query.getResultList();
}
```

JPA Criteria는 JPA 표준 스펙이지만 실무에서 사용하기에 너무 복잡하다. 결국 다른 대안이 필요하다. 많은 개발자가 비슷한 고민을 했지만, 가장 멋진 해결책은 Querydsl이 제시했다. Querydsl 소개장에서 간단히 언급하겠다. 지금은 이대로 진행하자.

> 참고: JPA Criteria에 대한 자세한 내용은 자바 ORM 표준 JPA 프로그래밍 책을 참고하자

웹 계층 개발

- 홈 화면
- 회원 기능
  - 회원 등록
  - 회원 조회
- 상품 기능
  - 상품 등록
  - 상품 수정
  - 상품 조회
- 주문 기능
  - 상품 주문
  - 주문 내역 조회
  - 주문 취소

상품 등록
상품 목록
상품 수정
변경 감지와 병합
상품 주문

홈 화면과 레이아웃