04 엔티티 매핑

발표자 : 전예진

P.S 다애님에게 무한한 감사를...

01

다양한 매핑 어노테이션

@Table @Entity ..

02

필드와 컬럼 매핑

JPA가 제공하는 필드와 컬럼 매핑용 어노테이션을 배워보자
@Column @Lob @Temporal @Enumerated

03

DB 스키마 자동 생성

JPA가 제공하는 DB스키마 자동 생성 기능을 알아보고, 적용 방법과 옵션에 대해 알아본다

04

기본키 매핑

IDENTITY, AUTO, SEQUENCE, TABLE 같은 기본 키 매핑 방식을 알아본다

@Entity

@Entity 어노테이션을 붙인다 == 테이블과 매핑할 클래스다!

@Entity를 붙인 클래스는 JPA가 관리한다!

- JPA가 엔티티 객체를 생성할 때 기본 생성자를 사용하므로 **기본 생성자 필수!**
- **final 클래스, enum, interface, inner클래스**에는 사용할 수 없음
- 지정할 필드에 **final**을 사용하면 안됨 → 왜?

  - final 을 붙인다는 것은 한 번 값을 저장하면 다른 값으로 변경이 불가!
  - user.setName()같은 수정 쿼리가 불가능할 것!

@Table

: @Table을 통해 엔티티와 매핑할 테이블을 지정함

속성	기능
@Table(name = "~")	매핑할 테이블 이름을 지정함 (디폴트값은 엔티티 이름)
catalog	카탈로그 기능이 있는 데이터베이스에서 카탈로그를 매핑
schema	스키마 기능이 있는 데이터베이스에서 스키마를 매핑

1. @ Entity를 붙여줌으로써 클래스와
   테이블을 매핑

2. 기본 생성자가 필수라고 했는데..?
- @NoArgsConstructor을 사용했구나!
- 자바는 임의의 생성자를 하나 이상 만들면
  기본 생성자를 자동으로 만들지 않음!
  그러므로 꼭 기본 생성자를 만들어주자

3. @Table(name = "~")로 이름 지정
- 약간의 꿀팁

```java
@Getter
@NoArgsConstructor
@Table(name = "Songe")
@Entity
public class Character {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne(fetch =FetchType.LAZY)
    @JoinColumn(name = "user_id",  referencedColumnName = "id")
    private User user;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String quote;

    @Builder
    public Character(String name, String quote) {
        this.name = name;
        this.quote = quote;
    }
```

```java
@Entity
@Table(name="MEMBER")
public class Member {

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

    @Column(name = "NAME", nullable = false, length = 10) //추가
    private String username;

    private Integer age;

    @Enumerated(EnumType.STRING)
    private RoleType roleType;

    @Temporal(TemporalType.TIMESTAMP)
    private Date createdDate;

    @Temporal(TemporalType.TIMESTAMP)
    private Date lastModifiedDate;

    @Lob
    private String description;

    @Transient
    private String temp;

    //Getter, Setter

    ...
}
```

```java
package jpabook.start;

public enum RoleType {
    ADMIN, USER
}
```

요구 사항 1 : 일반회원과 관리자로 구분
- enum을 통해 회원을 구분
- enum을 사용하려면 @Enumerated로 매핑을 해주어야함!

요구 사항 2 : 가입일과 수정일이 존재
- 자바의 날짜 타입은 @Temporal을 사용해서 매핑

요구사항 3 : 회원을 설명할 필드가 존재 (길이제한X)
- 길이 제한이 없는 필드는 VARCHAR대신에 CLOB 타입으로 저장해야 함
- @Lob을 사용하면 CLOB, BLOB타입을 매핑 가능

@Column

: 필드를 테이블 컬럼에 매핑

```java
@Column(name = "NAME", nullable = false, length = 10)
private String username;
```

속성	기능
name	필드와 매핑할 테이블 컬럼의 이름을 지정 (디폴트는 필드 이름)
nullable	null값의 허용 여부를 설정 (디폴트는 true) false로 설정하면 DDL 생성 시 not null 제약 조건이 붙음
length	String타입에 문자 길이의 제약조건을 지정할 수 있음

@Column 생략

```
int data1; //@Column 생략. 자바 기본 타입
data1 integer not null // 생성된 DDL

Integer data2; //@Column생략. 객체 타입
data2 integer

@Column
int data3;
data3 integer
```

1. 자바 기본 타입에는 null값을 입력할 수 없으며 객체 타입일 때만 null값이 허용

2. 그러므로 JPA는 DDL 생성 시에 기본 타입에는 not null 제약 조건을 추가

3. 반면 객체타입이면 null이 입력될 수 있으므로 not null 제약 조건을 설정하지 않음

4. @Column을 사용하면 기본 값이 nullable = true 이므로 기본타입에 not null제약조건을 설정하지 않음따라서 기본 타입에 @Column을 상ㅇ하는 경우nullable = false로 지정하는 것이 안전

@Enumerated

```java
@Enumerated(EnumType.STRING)
private RoleType roleType;
```

: enum 타입을 매핑할 때사용

EnumType.ORDINAL

- 순서를 데이터베이스에 저장 (0, 1, 2...)
- ADIMN은 0, USER는 1로 저장
- 저장되는 데이터 크기가 작음
- 이미 저장된 enum의 순서를 바꿀 수 없음

만약 ADMIN(0), USER(1) 사이에 enum이 추가되어
ADMIN(0), NEW(1), USER(2)로 설정되었다면 ?
수정 이후 부터는 USER는 2로 저장되나 기존에 DB에
저장된 값은 여전히 1로 남아있음
-> 그냥 String으로 쓰자!

EnumType.STRING

- 저장된 enum의 순서가 바뀌거나 추가되어도 안전
- 데이터베이스에 저장되는 데이터 크기가 ORDINAL에 비해 더 큼

@Temporal

```java
@Temporal(TemporalType.TIMESTAMP)
private Date createdDate;

@Temporal(TemporalType.TIMESTAMP)
private Date lastModifiedDate;
```

: 날짜 타입을 매핑할 때사용

@Temporal(TemporalType.DATE)
: 데이터베이스의 date 타입과 매핑. 날짜를 의미 (ex . 2022-05-28)

@Temporal(TemporalType.TIME)
: 데이터베이스의 time 타입과 매핑. 시간을 의미 (ex. 13:11:12)

@Temporal(TemporalType.TIMESTAMP)
: 데이터베이스의 timestamp와 매핑 날짜와 시간 (ex. 2022-05-28 13:11:12)

@Lob

```java
@Lob
private String description;
```

: 데이터베이스 BLOB, CLOB 타입과 매핑

- Lob는 지정할 수 없는 속성이 없음
- VARCHAR보다 큰 값 지정 시 사용

- 매핑하는 필드 타입이 문자면 CLOB로, 나머지는 BLOB 로 매핑
  - CLOB : String, char[], java.sql.CLOB
  - BLOB : byte[], java.sql.BLOB

@Access

: JPA가 엔티티에 접근하는 방식을 지정

@Access(AccessType.FIELD)
: 필드에 직접 접근하는 방법
필드 접근 권한이 private이여도 접근 가능

@Access(AccessType.PROPERTY)
: 접근자 getter를 사용하여 접근하는 방법

@Access(AccessType.FIELD)

```java
@Access(AccessType.FIELD)
@Entity
public class Member {
    @Id
    private String id;
    // @Id가 필드에 있으므로 @Access(AccessType.FIELD)로 설정한 것과 같음 따라서 생략 가능
}
```

@Access(AccessType.PROPERTY)

```java
@Access(AccessType.PROPERTY)
@Entity
public class Member {
    private String id;

    @Id
    public String getId() {
        return id;
    }
}
```

DB 스키마 자동생성

JPA는 데이터베이스 스키마를 자동으로 생성하는 기능을 제공
- 매핑정보와 데이터베이스 방언을 이용해서 데이터베이스 스키마를 생성

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.hibernate.ddl-auto=create
```

show.sql= true : 콘솔에 테이블 생성 DDL 출력가능
hibernate.ddl-auto=create : 애플리케이션 실행 시점에 DB 테이블을 자동생성

DB 스키마 자동생성

```
Hibernate:

    create table songe (
       id bigint not null auto_increment,
        name varchar(255) not null,
        quote varchar(255) not null,
        user_id bigint,
        primary key (id)
    ) engine=InnoDB
```

1. 자동 생성되는 DDL은 지정한
데이터 베이스 **방언**에 따라 달라짐

- 오라클 : varchar2, number타입 사용
- MySQL : varchar(255)

2. 운영 환경에 사용할 만큼 완벽하지 않으므로
개발 환경에서 사용하거나 매핑을 어떻게
해야 하는지 참고하는 용도로만 사용하자

DB 스키마 자동생성

옵션	설명
create	기존 테이블을 삭제하고 새로 생성 (DROP + CREATE)
create - drop	create 속성과 비슷하나 애플리케이션 종료 시 생성한 DDL 삭제 (DROP + CREATE + DROP)
update	데이터베이스 테이블과 엔티티 매핑 정보를 비교하여 변경 사항만 수정 -> **필드를 추가하고 drop table하고 싶지 않을 때 사용!**
validate	데이터베이스 테이블과 엔티티 매핑 정보를 비교하여 변경 사항이 있으면 경고를 남기고 애플리케이션을 실행시키지 않음. DDL 수정도 하지 않음! -> DDL을 수정하지 않으니까 필드 하나 추가하면 차이가 발생 -> **엔티티와 테이블이 정상 매핑 되었는지 확인만 할 때 사용**
none	자동 생성 기능을 사용하지 않음

기본 키 매핑

```java
@Id
@Column(name = "ID")
private String id;
```

직접할당 방식 : 기본 키를 애플리케이션에서 직접 할당

자동생성 방식 : 대리 키 사용 방식 @GeneratedValue(strategy = "GeneratedType.~")

- IDENTITY : 기본 키 생성을 데이터베이스에 위임
- SEQUENCE : 데이터베이스 시퀀스를 사용해서 기본 키를 할당
- TABLE : 키 생성 테이블을 사용
- AUTO : 데이터베이스 방언에 따라 설정

기본 키 매핑 - 직접 할당

em.persist()로 엔티티 저장 전에 기본 키를 직접 할당하는 방식

@ID 적용 가능 자바 타입 : 기본형, Wrapper형, String, Date .. 등

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

User user = new User();
user.setId("id1"); // 기본 키 직접 할당
em.persist(user);
```

식별자 값 없이 저장하면 예외발생!

IDENTITY 전략

기본 키 생성을 데이터베이스에 위임하는 전략
- MySQL의 **auto_increment**기능은 데이터베이스가 기본키를 자동으로 생성
- 이 전략은 데이터베이스에 값을 **저장**하고 기본 키 값을 구할 수 있을 때 사용
- em.persist()를 호출해서 엔티티를 저장한 후에 식별자 값을 출력
- 그러므로 엔티티에 식별자 값을 할당하려면 JPA는 추가로 데이터베이스를 조회해야 함

```java
@Id
@GeneratedValue(strategy = GenrationType.IDENTITY)
private Long id;

User user = new User();
em.persist(user);
System.out.println(user.getId()); // 1이 출력됨
```

```
Hibernate:

    create table songe (
       id bigint not null auto_increment,
        name varchar(255) not null,
        quote varchar(255) not null,
        user_id bigint,
        primary key (id)
    ) engine=InnoDB
```

SEQUENCE 전략

데이터베이스 시퀀스는 유일한 값을 순서대로 생성하는데, 이 시퀀스를 통해 기본키를 생성함

1. 시퀀스를 사용하려면 먼저 시퀀스 생성를 하자

```
CREATE TABLE BOARD {
    ID BIGINT NOT NNULL RIMARY KEY,
    DATA VARCHAR(255)
}

CREATE SEQUENCE BOARD_SEQ START WITH 1 INCREMENT BY 1;
```

SEQUENCE 전략

데이터베이스 시퀀스는 유일한 값을 순서대로 생성하는데, 이 시퀀스를 통해 기본키를 생성함

2. 사용할 데이터 베이스 시퀀스를 매핑하자

```
@Entity
@SequencGenerator (
    name = "BOARD_SEQ_GENERATOR",
    sequenceName = "BOARD_SEC", //매핑할 데이터베이스 시퀀스 이름
    initialValue = 1, allocationSize = 1 )

public class Board {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "BOARD_SEQ_GENERATOR") //시퀀스 생성기 선택
    private Long id

}
```

1) @SequenceGenerator를 통해 시퀀스 생성기를 등록
2) sequenceName 속성으로 매핑할 실제 데이터베이스 시퀀스 이름을 지정
3) 키생성전략을 SEQUENCE로 설정하고 generator를 방금 등록한 시퀀스 생성기로 선택
4) 이제부터 id값은BOARD_SEQ_GENERATOR 시퀀스 생성기가 할당!

IDENTITY vs SEQUENCE

: 사용코드는 동일하나 내부 동작 방식은 다르다!!

SEQUENCE 전략은 em.persist()를 호출할 때 먼저 데이터베이스 시퀀스를 사용해서 식별자를 조회하고, 조회한 식별자를 엔티티에 할당 후에 엔티티를 영속성 컨텍스트에 저장!
이후에 트랜잭션을 커밋해서 플러시가 일어나면 엔티티를 데이터베이스에 저장

반면 IDENTITY 전략은 먼저 엔티티를 데이터베이스에 저장한 후에 식별자를 조회해서 엔티티의 식별자에 할당! 즉, em.persist()으로 저장이 먼저 이루어지고, 이후에 재조회를 통해 식별자를 할당

Table 전략

: 키 생성 전용 테이블을 하나 만들고 여기에 이름과 값으로 사용할 컬럼을 만들어
데이터베이스 시퀀스를 흉내내는 방식 -> 모든 데이터베이스에 적용할 수 있다는 장점

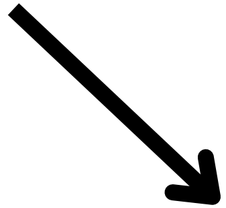
```
create table MY_SEQUENCES {
    sequence_name varchar(255) not null,
    next_val bigint;
    primary key (sequence_name)
}
```

```
@Entity
@TableGenerator (
    name = "BOARD_SEQ_GENERATOR",
    table = "MY_SEQUENCES",
    pkColuValue = "BOARD_SEQ", allocationSize = 1 )

public class Board {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE,
                    generator = "BOARD_SEQ_GENERATOR")
    private Long id

}
```

실 전 예 제

요구사항 분석

1. 회원은 상품을 주문할 수 있다
2. 주문 시 여러 종류의 상품을 선택할 수 있다



<요구사항 분석을 통해 기능 도출>

1. 회원 기능 (등록, 수정)
2. 상품 기능 (등록, 수정, 조회)
3. 주문 기능 (주문, 주문내역 조회, 주문 취소)

도메인 모델 분석

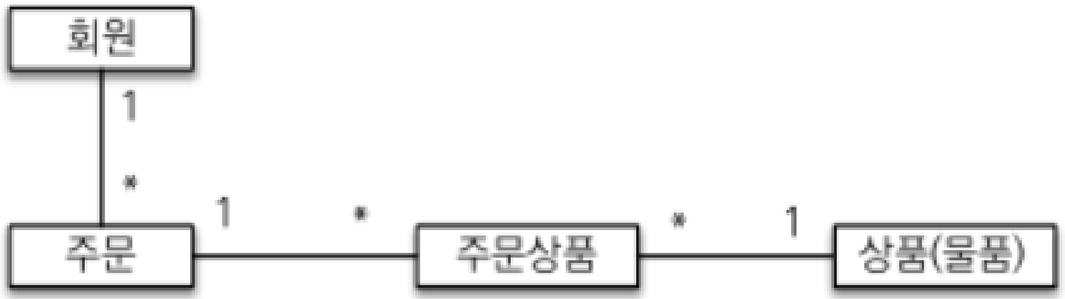
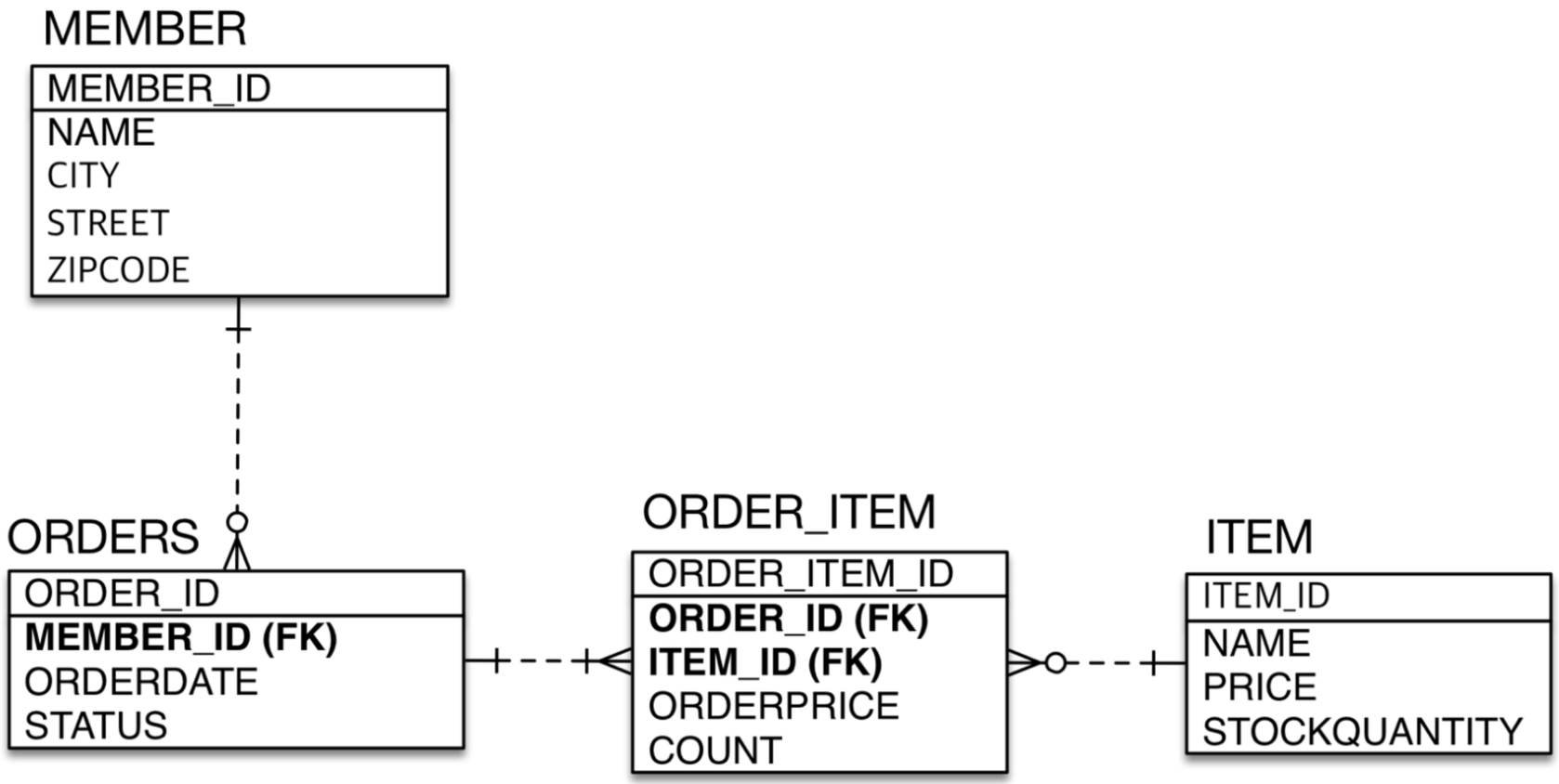


그림 4.2 실전 예제 1 UML

1. 회원은 여러번 주문 할 수 있으므로 둘의 관계는 **일대다 관계**
2. 주문 시 여러 종류의 상품을 선택할 수 있고, 같은 상품도 여러번 주문될 수 있으므로 **다대다 관계**
   하지만 다대다 관계는 거의 사용하지않음!
   그래서 주문 상품이라는 엔티티를 만들어서 다대 일관계로 풀어내자!

테이블 설계


MEMBER
MEMBER_ID
NAME
CITY
STREET
ZIPCODE
ORDERS
ORDER_ID
MEMBER_ID (FK)
ORDERDATE
STATUS
ORDER_ITEM
ORDER_ITEM_ID
ORDER_ID (FK)
ITEM_ID (FK)
ORDERPRICE
COUNT
ITEM
ITEM_ID
NAME
PRICE
STOCKQUANTITY

엔티티 설계와 매핑

설계한 테이블 기반으로 실제 엔티티를 설계하자!

회원 엔티티

```java
@Setter
@Getter
@Entity
public class Member {

    @Id @GeneratedValue
    @Column(name = "MEMBER_ID")
    private Long id;
    private String name;
    private String city;
    private String street;
    private String zipcode;
}
```

- 회원은 이름과 주소 정보를 가짐
- 주소는 city, street, zipcode로 표현
- @Id와 @GeneratedValue를 사용해서 데이터베이스에서 자동 생성하도록 함
- @GenartedValue의 기본 생성 전략은 AUTO
- 그러므로 데이터베이스 방언에 따라 전략이 선택됨
- 여기서는 H2 DB를 사용하며 전략은 SEQUENCE

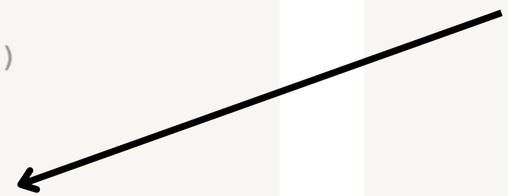
주문 엔티티

```java
@Getter @Setter
@Entity
@Table(name = "ORDERS")
public class Order {

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

    @Column(name = "MEMBER_ID")
    private Long memberId; // 상품을 주문할 회원의 외래키 값

    @Temporal(TemporalType.TIMESTAMP)
    private Date orderDate; // 주문 날짜

    @Enumerated(EnumType.STRING)
    private OrderStatus status; // 주문 상태

}

public enum OrderStatus { // 주문 상태
    ORDER, CANCEL
}
```

- 주문은 상품을 주문한 회원의 외래키 값과 주문 날짜, 주문 상태를 가짐
- 주문 날짜는 Date 사용. 년월일 시분초를 모두 사용하므로 @Temporal에 timestamp속성을 사용
- 주문 상태는 열거형을 사용하므로 @Enumerated로 매핑하고 String속성을 주어 열거형 이름이 그대로 저장 되도록 함.

주문 상품, 상품 엔티티

```java
@Getter @Setter
@Entity
public class OrderItem {

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

    @Column(name = "ORDER_ID")
    private Long ordeId;

    @Column(name = "ITEM_ID")
    private Long itemId;

    private int orderPrice;
    private int count;
}
```



```java
@Getter @Setter
@Entity
public class Item {

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

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

데이터 중심 설계의 매핑

이 방식의 문제점 ?

1. 테이블의 외래키를 객체에 그대로 가져왔다!
   - 관계형 DB는 연관된 객체를 찾을 때 외래키를 사용하여 조인하면 됨
   - 하지만 객체에는 조인 기능이 없고 참조를 통해서만 연관객체를 찾을 수 있음

2. 외래키만 그대로 가지고 있으면 외래키로 데이터베이스를 다시 조회해야함
   ex) Order order = em.find(Order.class, orderId);
   Member member = em.find(Member.class, order.getMemberId());

3. 참조를 사용한다면?
   ex) Order order = em.find(Order.class, orderId);
   Member member = order.getMember();