

JPA

# JPA Setter 사용하지 않기

# Setter 메소드는 의도를 갖기 힘들다.

```
public Account updateMyAccount(long id, AccountDto.MyAccountReq dto) {  
    final Account account = findById(id);  
    account.setAddress("value");  
    account.setFistName("value");  
    account.setLastName("value");  
    return account;  
}
```

- 위의 코드는 회원 정보의 성, 이름, 주소를 변경하는 코드로 여러 **setter** 메소드들이 나열돼있습니다.
- **setter**들은 회원 정보를 변경하기 위한 나열이라서 **메소드들의 의도가 명확히 드러나지 않습니다.**

# Setter 메소드는 의도를 갖기 힘들다.

```
public static class MyAccountReq {  
    private Address address;  
    private String firstName;  
    private String lastName;  
}
```

```
public Account updateMyAccount(long id, AccountDto.MyAccountReq dto) {  
    final Account account = findById(id);  
    account.updateMyAccount(dto);  
    return account;  
}  
// Account 도메인 클래스  
public void updateMyAccount(AccountDto.MyAccountReq dto) {  
    this.address = dto.getAddress();  
    this.firstName = dto.getFirstName();  
    this.lastName = dto.getLastName();  
}
```

- **Account** 도메인 클래스에 **updateMyAccount** 메소드를 통해서 회원정보업데이트를 진행했습니다. 위의 코드보다 의도가 명확히 드러납니다.
- 위는 **MyAccountReq** 클래스입니다. 회원 정보 수정에 필요한 값 즉 변경될 값에 대한 명확한 명세가 있어 DTO를 두는 것이 바람직합니다.

# 객체의 일관성을 유지하기 어렵다

```
public Account updateMyAccount(long id, AccountDto.MyAccountReq dto) {  
    final Account account = findById(id);  
    account.setEmail("value");  
    return account;  
}
```

- **setter** 메소드가 있을 때 객체에 언제든지 변경할 수 있게 됩니다.
- 회원 변경 메소드뿐만 아니라 모든 곳에서 이메일 변경이 가능하게 됩니다.
- 변경이 불가능 한 항목에 **setter** 메서드를 두지 않는다는 방법도 있지만 관례로 **setter**는 모든 멤버필드에 대해서 만들기도 하거니와 실수 조금이라도 덜 할 수 있게 하는 것 이 바람직한 구조라고 생각합니다.

# JPA 연관관계 설정

# OneToOne 관계 설정 팁

```
public class Coupon {  
    @Id  
    @GeneratedValue  
    private long id;  
  
    @Column(name = "discount_amount")  
    private double discountAmount;  
  
    @Column(name = "use")  
    private boolean use;  
  
    @OneToOne()  
    private Order order;  
}  
  
public class Order {  
    @Id  
    @GeneratedValue  
    private long id;  
  
    @Column(name = "price")  
    private double price;  
  
    @OneToOne  
    @JoinColumn()  
    private Coupon coupon;  
}
```

- 주문과 쿠폰 엔티티가 있다.
- 주문 시 쿠폰을 적용해서 할인받을 수 있다.
- 주문과 쿠폰 관계는 1:1 관계 즉 OneToOne 관계이다.
- 주의 깊게 살펴볼 내용은 다음과 같습니다.
  - 외래 키는 어디 테이블에 두는 것이 좋은가?
  - 양방향 연관 관계 편의 메소드
  - 제약 조건으로 인한 안정성 및 성능 향상

# 외래 키는 어디 테이블에 두는 것이 좋은가?

```
// Order가 연관관계의 주인일 경우
@OneToOne
@JoinColumn(name = "coupon_id", referencedColumnName = "id")
private Coupon coupon;

@OneToOne(mappedBy = "coupon")
private Order order;

// coupon이 연관관계의 주인일 경우
@OneToOne(mappedBy = "order")
private Coupon coupon;

@OneToOne
@JoinColumn(name = "order_id", referencedColumnName = "id")
private Order order;
```

- 일대다 관계에서는 다 쪽에서 외래 키를 관리하게 되지만 상대적으로 일대일 관계 설정에는 외래 키를 어느 곳에 두어야 하는지를 생각을 해야 합니다.
- JPA 상에서는 외래 키가 갖는 쪽이 연관 관계의 주인이 되고 **연관 관계의 주인만이 데이터베이스 연관 관계와 맵핑되고 외래 키를 관리(등록, 수정, 삭제)할 수 있기 때문입니다.**

# Order가 주인일 경우 장점 : INSERT SQL이 한번 실행

```
Tests passed: 1 of 1 test – 148 ms
OrderServiceTes: 148 ms
order_쿠폰할인: 148 ms

Hibernate: drop table delivery if exists
Hibernate: drop table delivery_log if exists
Hibernate: drop table orders if exists
Hibernate: create table account (id bigint generated by default as identit
Hibernate: create table coupon (id bigint generated by default as identit
Hibernate: create table delivery (id bigint generated by default as ident
Hibernate: create table delivery_log (id bigint generated by default as i
Hibernate: create table orders (id bigint generated by default as identit
Hibernate: alter table account add constraint UK_q0uja26qgu1atulenwup9rxy
Hibernate: alter table delivery_log add constraint FKfs49km0ea809mth3oq4s
Hibernate: alter table orders add constraint FKa5ei0aklq6wrjl8vrr7ied3bx
Hibernate: select coupon0_.id as id1_1_0_, coupon0_.discount_amount as di
Hibernate: insert into orders (id, coupon_id, price) values (null, ?, ?)
couponId : 1
Hibernate: drop table account if exists
Hibernate: drop table coupon if exists
Hibernate: drop table delivery if exists
Hibernate: drop table delivery_log if exists
Hibernate: drop table orders if exists

Process finished with exit code 0
```

- order 테이블에 coupon\_id 칼럼을 저장하기 때문에 주문 SQL은 한 번만 실행됩니다.
- 반면에 coupon이 연관 관계의 주인일 경우에는 coupon에 order의 외래 키가 있으니 order INSERT SQL 한 번, coupon 테이블에 order\_id 칼럼 업데이트 쿼리 한번 총 2번의 쿼리가 실행됩니다.

```
// order가 연관 관계의 주인일 경우 SQL
insert into orders (id, coupon_id, price) values (null, ?, ?)

//coupon이 연관 관계의 주인일 경우 SQL
insert into orders (id, price) values (null, ?)
update coupon set discount_amount=?, order_id=?, use=? where id=?
```

# Order가 주인일 경우 단점 : 연관 관계 변경 시 취약

```
public class Coupon {  
    @Id  
    @GeneratedValue  
    private long id;  
  
    @Column(name = "discount_amount")  
    private double discountAmount;  
  
    @Column(name = "use")  
    private boolean use;  
  
    @OneToOne()  
    private Order order;  
}  
  
public class Order {  
    @Id  
    @GeneratedValue  
    private long id;  
  
    @Column(name = "price")  
    private double price;  
  
    @OneToOne  
    @JoinColumn()  
    private Coupon coupon;  
}
```

- 기존 요구사항은 주문 한 개에 쿠폰은 한 개만 적용이 가능 했기 때문에 **OneToOne** 연관 관계를 맺었지만 **하나의 주문에 여러 개의 쿠폰이 적용되는 기능이 추가되었을 때 변경하기 어렵다는 단점이 있습니다.**
  - **order** 테이블에 **coupon\_id** 칼럼을 갖고 있어서 여러 개의 쿠폰을 적용하기 위해서는 **coupon** 테이블에서 **order\_id** 칼럼을 가진 구조로 변경 해야 합니다.
- OneToMany** 관계에서는 연관 관계의 주인은 외래 키를 갖는 엔티티가 갖는 것이 바람직합니다. 비즈니스 로직 변경은 어려운 게 없으나 **데이터베이스 칼럼들을 이전 해야 하기 때문에 실제 서비스 중인 프로젝트에는 상당히 골치 아프게 됩니다.**

# 연관 관계의 주인 설정

- OneToOne 관계를 맺으면 외래 키를 어디에 둘 것인지, 즉 연관 관계의 주인을 어디에 둘 것인지는 많은 고민이 필요 합니다. 제 개인적인 생각으로는 OneToMany로 변경될 가능성이 있는지를 판단하고 변경이 될 가능성이 있다고 판단되면 Many가 될 엔티티가 관계의 주인이 되는 것이 좋다고 봅니다. 또 애초에 OneToMany를 고려해서 초기 관계 설정을 OneToMany로 가져가는 것도 좋다고 생각합니다.
- 연관 관계가 정말 OneToOne 관계인지 깊은 고민이 필요하고 해당 도메인에 대한 지식도 필요 하다고 생각합니다. 예를 들어 개인 송금 관계에서 입금 <-> 출금 관계를 가질 경우 반드시 하나의 입금 당 하나의 출금을 갖게 되니 이것은 OneToOne 관계로 맺어가도 무리가 없다고 판단됩니다. (물론 아닌 때도 있습니다. 그래서 해당 도메인에 대한 지식이 필요 한다고 생각합니다)
- 주인 설정이라고 하면 뭔가 더 중요한 것이 주인이 되어야 할 거 같다는 생각이 들지만 연관 관계의 주인이라는 것은 외래 키의 위치와 관련해서 정해야 하지 해당 도메인의 중요성과는 상관관계가 없습니다.

# 제약 조건으로 인한 안정성 및 성능 향상

```
public class Order {  
    ...  
  
    @OneToOne  
    @JoinColumn(name = "coupon_id", referencedColumnName = "id", nullable = false)  
    private Coupon coupon;  
}
```

- 모든 주문에 할인 쿠폰이 적용된다면 `@JoinColumn`의 **nullable** 옵션을 **false**로 주는 것이 좋습니다. NOT NULL 제약 조건을 준수해서 안전성이 보장됩니다.
- JPA 기반으로 DDL을 작성하지 않더라도 해당 도메인의 Spec으로 표시하는 편이 좋다고 생각합니다.

# 제약 조건으로 인한 안정성 및 성능 향상

The image shows two side-by-side IntelliJ IDEA debug consoles. Both are titled 'Debug - spring-jpa' and show the 'Variables' tab selected. The left console displays the following SQL query:

```
count2_1_1_, coupon1_.use as use3_1_1_ from orders order0_ left outer join coupon coupon1_ on order0_.coupon_id=coupon1_.id where order0_
count2_1_0_, coupon1_.use as use3_1_0_ from orders order0_ left outer join coupon coupon1_ on order0_.coupon_id=coupon1_.id where order0_
```

The right console displays a similar SQL query, partially cut off at the end:

```
_1_1_, coupon1_.discount_amount as discount2_1_1_, coupon1_.use as use3_1_1_ from orders order0_ inner join coupon coupon1_ on order0_.cou
_1_0_, coupon1_.discount_amount as discount2_1_0_, coupon1_.use as use3_1_0_ from orders order0_ inner join coupon coupon1_ on order0_.cou
```

- 외래 키에 NOT NULL 제약 조건을 설정하면 값이 있는 것을 보장합니다. 따라서 **JPA는 이때 내부조인을 통해서 SQL을 만들** 어 주고 이것은 외부 조인보다 성능과 최적화에 더 좋습니다.
- 물론 모든 경우에 적용할 수는 없고 반드시 외래 키가 NOT NULL인 조건에만 사용할 수 있습니다. 예를 들어 쿠폰과 회원 연관 관계가 있을 때 쿠폰은 반드시 회원의 외래 키를 참조하고 있어야 합니다. 이런 경우 유용하게 사용할 수 있습니다.

# 양방향 연관관계 편의 메소드

```
// Order가 연관관계의 주인일 경우 예제
class Coupon {
    ...
    // 연관관계 편의 메소드
    public void use(final Order order) {
        this.order = order;
        this.use = true;
    }
}

class Order {
    private Coupon coupon; // (1)
    ...
    // 연관관계 편의 메소드
    public void applyCoupon(final Coupon coupon) {
        this.coupon = coupon;
        coupon.use(this);
        price -= coupon.getDiscountAmount();
    }
}

// 주문 생성시 1,000 할인 쿠폰 적용
public Order order() {
    final Order order = Order.builder().price(10_000).build(); // 10,000 상품주문
    Coupon coupon = couponService.findById(1); // 1,000 할인 쿠폰
    order.applyCoupon(coupon);
    return orderRepository.save(order);
}
```

- 양방향 연관 관계일 경우 위처럼 연관 관계 편의 메소드를 작성하는 것이 좋습니다. 위에서 말했듯이 연관 관계의 주인만이 외래 키를 관리 할 수 있으니 **applyCoupon** 메소드는 이해하는데 어렵지 않습니다.

# 양방향 연관관계 편의 메소드가 필요한 이유

```
public void use(final Order order) {  
    // this.order = order; 해당코드를 주석했을 때 테스트 코드  
    this.use = true;  
}  
  
@Test  
public void use_메서드에_order_set_필요이유() {  
    final Order order = orderService.order();  
    assertThat(order.getPrice(), is(9_000D)); // 1,000 할인 적용 확인  
    final Coupon coupon = order.getCoupon();  
    assertThat(coupon.getOrder(), is(notNullValue())); // 해당 검사는 실패한다.  
}
```

- order를 바인딩하는 코드를 주석하고 해당 코드를 돌려 보면 실패하게 됩니다.
- 일반적으로 생각했을 때 order 생성 시 1,000할인 쿠폰을 적용했기 때문에 해당 쿠폰에도 주문 객체가 들어갔을 거로 생각할 수 있습니다. 하지만 위의 주석시킨 코드가 그 기능을 담당했기 때문에 쿠폰 객체의 주문 값은 null인 상태입니다.
- 즉 순수한 객체까지 고려한 양방향 관계를 고려하는 것 이 바람직하고 그것이 안전합니다.

# OneToMany 관계 설정

```
@Entity
public class Delivery {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Embedded
    private Address address;

    @OneToMany(mappedBy = "delivery", cascade = CascadeType.PERSIST, orphanRemoval = true, fetch = FetchType.LAZY)
    private List<DeliveryLog> logs = new ArrayList<>();
}

@Entity
public class DeliveryLog {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Enumerated(EnumType.STRING)
    @Column(name = "status", nullable = false, updatable = false)
    private DeliveryStatus status;

    @ManyToOne
    @JoinColumn(name = "delivery_id", nullable = false, updatable = false)
    private Delivery delivery;
}
```

- 배송이 있고 배송의 상태를 갖는 배송 로그가 있습니다.
- 배송과 배송 상태는 1:N 관계를 갖는다.
- 배송은 배송 상태를 1개 이상 반드시 갖는다

# Delivery 저장

```
class Delivery {  
    public void addLog(DeliveryStatus status) {  
        this.logs.add(DeliveryLog.builder()  
            .status(status)  
            .delivery(this) // this를 통해서 Delivery를 넘겨준다.  
            .build());  
    }  
  
    class DeliveryLog {  
        public DeliveryLog(final DeliveryStatus status, final Delivery delivery) {  
            this.delivery = delivery;  
        }  
    }  
  
    class DeliveryService {  
        public Delivery create(DeliveryDto.CreationReq dto) {  
            final Delivery delivery = dto.toEntity();  
            delivery.addLog(DeliveryStatus.PENDING);  
            return deliveryRepository.save(delivery);  
        }  
    }  
}
```

- Delivery가 시작되면 DeliveryLog는 반드시 PENDING이어야 한다고 가정했을 경우 편의 메소드를 이용해서 두 객체에 모두 필요한 값을 바인딩시켜줍니다.

# Delivery 저장

```
class Delivery {  
    public void addLog(DeliveryStatus status) {  
        this.logs.add(DeliveryLog.builder()  
            .status(status)  
            .delivery(this) // this를 통해서 Delivery를 넘겨준다.  
            .build());  
    }  
}
```

## CaseCascade PERSIST 설정

```
// cascade 없는 경우  
Hibernate: insert into delivery (id, address1, address2, zip, created_at, update_at) values (null, ?, ?, ?, ?, ?)  
  
// cascade PERSIST 설정 했을 경우  
Hibernate: insert into delivery (id, address1, address2, zip, created_at, update_at) values (null, ?, ?, ?, ?, ?)  
Hibernate: insert into delivery_log (id, created_at, update_at, delivery_id, status) values (null, ?, ?, ?, ?)
```

```
class DeliveryService {  
    public Delivery create(DeliveryDto.CreationReq dto) {  
        final Delivery delivery = dto.toEntity();  
        delivery.addLog(DeliveryStatus.PENDING);  
        return deliveryRepository.save(delivery);  
    }  
}
```

- **CaseCascade PERSIST를 통해서 Delivery 엔티티에서 DeliveryLog를 생성할수 있게 설정합니다.**
- **CaseCascade PERSIST가 없을 때 실제 객체에는 저장되지만, 영속성 있는 데이터베이스에 저장에 필요한 insert query가 동작하지 않습니다.**

# 고아 객체 (orphanRemoval) : DeliveryLog 삭제

```
public Delivery removeLogs(long id) {  
    final Delivery delivery = findById(id);  
    delivery.getLogs().clear(); // DeliveryLog 전체 삭제  
    return delivery; // 실제 DeliveryLog 삭제 여부를 확인하기 위해 리턴  
}
```

```
// delete SQL  
Hibernate: delete from delivery_log where id=?
```

- JPA는 **부모 엔티티와 연관 관계가 끊어진 자식 엔티티**를 **자동으로 삭제하는 기능을 제공하는데 이것을 고아 객체 제거라 합니다.**
- 이 기능을 사용해서 부모 엔티티의 컬렉션에서 자식 엔티티의 참조만 제거하면 **자식 엔티티가 자동으로 삭제** 돼서 **개발의 편리함**이 있습니다.
- **Delivery 객체를 통해서 DeliveryLog를 위처럼 직관적으로 삭제 할 수 있습니다.**

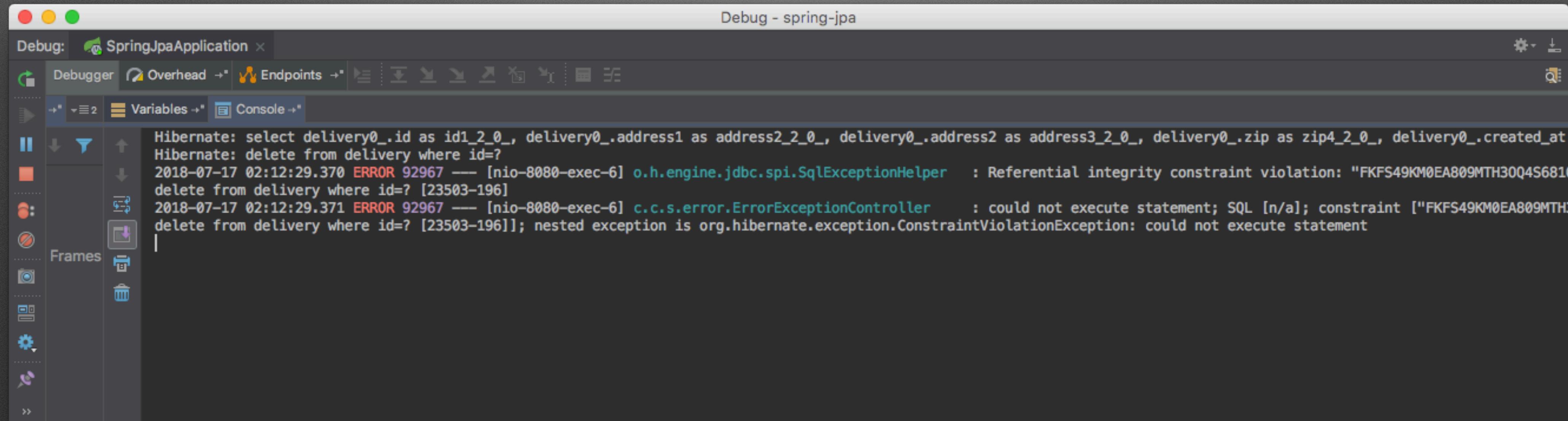
# 고아 객체 (orphanRemoval) : Delivery 삭제

```
public void remove(long id){  
    deliveryRepository.delete(id);  
}
```

```
// delete SQL  
Hibernate: delete from delivery_log where id=?  
Hibernate: delete from delivery where id=?
```

- **delivery, deliverylog 참조 관계를 맺고 있어 Delivery만 삭제할 수 없습니다.**
- **delete SQL을 보시다시피 delivery\_log 부터 제거 이후 delivery를 제거하는 것을 알 수 있습니다.**
- **이처럼 orphanRemoval 속성으로 더욱 쉽게 삭제 할 수 있습니다.**

# 고아 객체 (orphanRemoval) : 설정이 없는 경우



The screenshot shows the IntelliJ IDEA debugger interface with the title bar "Debug - spring-jpa". The "Console" tab is selected, displaying the following log output:

```
Hibernate: select delivery0_.id as id1_2_0_, delivery0_.address1 as address2_2_0_, delivery0_.address2 as address3_2_0_, delivery0_.zip as zip4_2_0_, delivery0_.created_at
Hibernate: delete from delivery where id=?
2018-07-17 02:12:29.370 ERROR 92967 --- [nio-8080-exec-6] o.h.engine.jdbc.spi.SqlExceptionHelper : Referential integrity constraint violation: "FKFS49KM0EA809MTH30Q4S6810
delete from delivery where id=? [23503-196]
2018-07-17 02:12:29.371 ERROR 92967 --- [nio-8080-exec-6] c.c.s.error.ErrorExceptionController : could not execute statement; SQL [n/a]; constraint ["FKFS49KM0EA809MTH30Q4S6810
delete from delivery where id=? [23503-196]]; nested exception is org.hibernate.exception.ConstraintViolationException: could not execute statement
```

- DeliveryLog 삭제 같은 경우에는 실제 객체에서는 clear() 메서드로 DeliveryLog가 삭제된 것처럼 보이지만 영속성 있는 데이터를 삭제하는 것은 아니기에 해당 Delivery를 조회하면 DeliveryLog가 그대로 조회됩니다.
- 실수하기 좋은 부분이기에 반드시 삭제하고 조회까지 해서 데이터베이스까지 확인하시는 것을 권장해 드립니다

# Embedded를 적극 활용

# 자료형의 통일

```
class Account {  
    // 단순 String  
    @email  
    @Column(name = "email", nullable = false)  
    private String email;  
  
    // Email 자료형  
    @Embedded  
    private Email email;  
}  
  
public Account findByEmail(final Email email) { //단순 문자열일 경우 (final String email)  
    final Account account = accountRepository.findByEmail(email);  
    if (account == null) throw new AccountNotFoundException(email);  
    return account;  
}
```

```
class Email {  
    @Email  
    @Column(name = "email", nullable = false, unique = true)  
    private String value;  
}
```

- **String** 자료형에서 **Email** 자료형으로 통일이 됩니다. 자료형이 통일되면 많은 더욱 안전성이 높아지는 효과가 있습니다.
- 이메일로 회원을 조회 할 때 단순 문자열일 경우에는 굳이 이메일 형식을 맞추지 않고도 단순 문자열을 통해서 조회할 수 있습니다. 이것은 편하게 느껴질지는 모르나 안전성에는 좋다고 생각하지 않습니다.
- 위처럼 단순 조회용뿐만이 아니라 Email에 관련된 모든 자료형을 단순 **String**에서 **Email**로 변경함으로써 얻을 수 있는 이점은 많습니다.

# 풍부한 객체 (Rich Object)

```
public class Email {  
    ...  
    public String getHost() {  
        int index = value.indexOf("@");  
        return value.substring(index);  
    }  
  
    public String getId() {  
        int index = value.indexOf("@");  
        return value.substring(0, index);  
    }  
}
```

- 이메일 아이디와 호스트값을 추출해야 하는 기능이 필요해질 경우 기존 **String** 자료형일 경우에는 해당 로직을 **Account** 도메인 객체에 추가하든, 유틸성 클래스에 추가하든 해야 합니다.
- 도메인 객체에 추가할 때는 **Account** 도메인 클래스가 갖는 책임들이 많아집니다. 또 이메일은 어디서든지 사용할 수 있는데 **Account** 객체에서 이 기능을 정의하는 것은 올바르지 않습니다.
- 유틸성 클래스에 추가하는 것 또한 좋지 않아 보입니다. 일단 유틸성 클래스에 해당 기능이 있는지 알아봐야 하고 기능이 있음에도 불구하고 그것을 모르고 추가하여 중복 코드가 발생하는 일이 너무나도 흔하게 발생합니다.
- 이것을 **Email** 형으로 빼놓았다면 아래처럼 **Email** 객체를 사용하는 곳 어디든지 사용할 수 있습니다. 해당 기능은 **Email** 객체가 해야 하는 일이고 또 그 일을 가장 잘할 수 있는 객체입니다. 또한 코드가 아주 이해하기 쉽게 됩니다. 객체의 기능이 풍부해집니다.

# 재사용성

```
class Remittance {
    //자료형이 없는 경우
    @Column(name = "send_amount") private double sendAamount;
    @Column(name = "send_country") private String sendCountry;
    @Column(name = "send_currency") private String sendCurrency;

    @Column(name = "receive_amount") private double receiveAamount;
    @Column(name = "receive_country") private String receiveCountry;
    @Column(name = "receive_currency") private String receiveCurrency;

    //Money 자료형
    private Money snedMoney;
    private Money receiveMoney;
}

class Money {
    @Column(name = "amount", nullable = false, updatable = false) private double amount;
    @Column(name = "country", nullable = false, updatable = false) private Country country;
    @Column(name = "currency", nullable = false, updatable = false) private Currency currency;
}
```

- 위처럼 Money라는 자료형을 두고 금액, 나라, 통화를 두면 도메인을 이해하는데 한결 수월할 뿐만 아니라 수많은 곳에서 재사용 할 수 있습니다. 사용자에게 해당 통화로 금액을 보여줄 때 소수자리 몇 자리로 보여줄 것인지 등등 핵심 도메인일수록 재사용성을 높여 중복 코드를 제거하고 응집력을 높일 수 있습니다.

JPA N+1

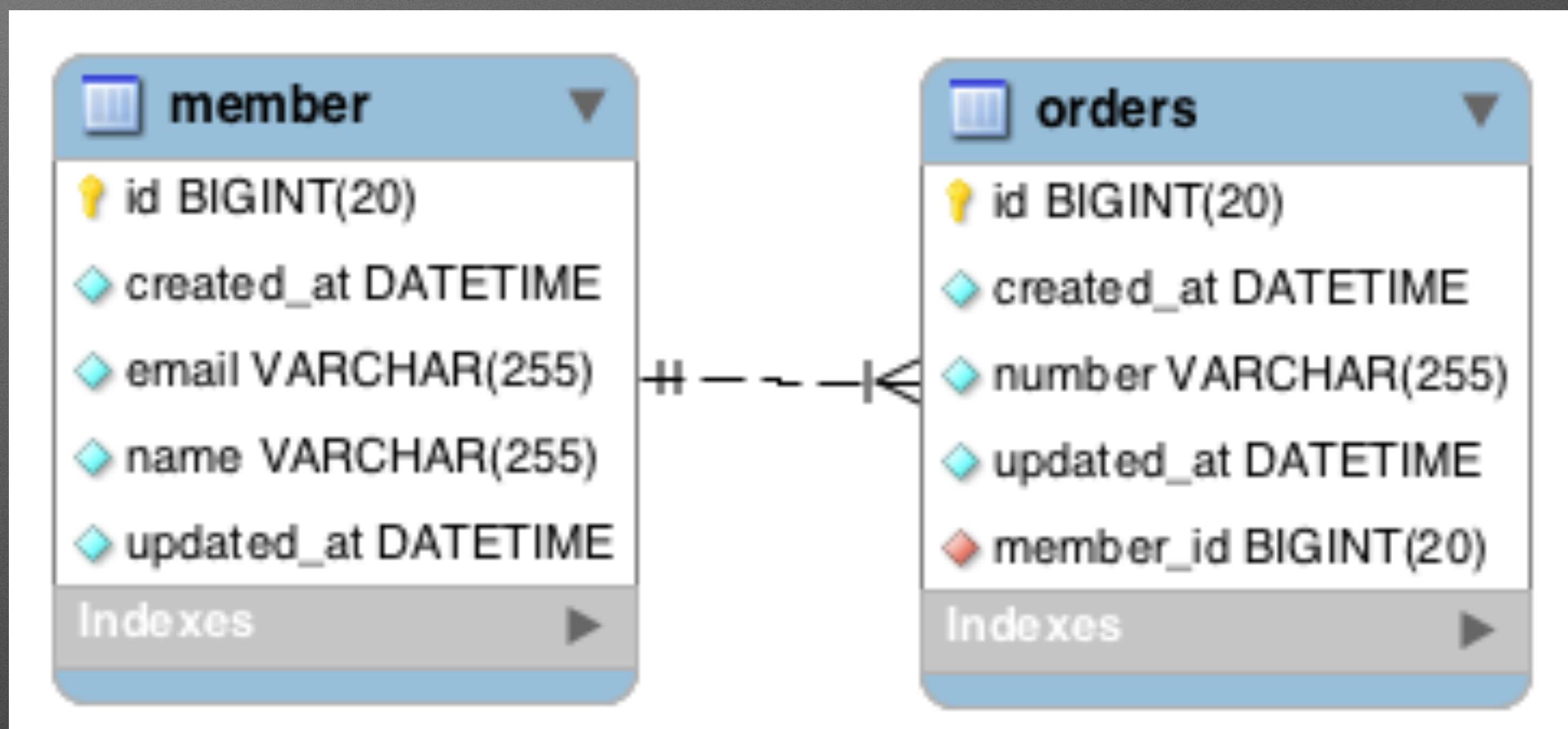
# ERD

```
@Entity
@Table(name = "member")
class Member private constructor() {

    ...
    @OneToMany(mappedBy = "member", fetch = FetchType.LAZY)
    var orders: Set<Order> = emptySet()
        private set
}

@Entity
@Table(name = "orders")
class Order private constructor() {
    ...

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "member_id", nullable = false, updatable = false)
    lateinit var member: Member
        private set
}
```



# 즉시 로딩 N+1

```
Hibernate:  
    select  
        orders0_.member_id as member_i5_1_0_,  
        orders0_.id as id1_1_0_,  
        orders0_.id as id1_1_1_,  
        orders0_.created_at as created_2_1_1_,  
        orders0_.member_id as member_i5_1_1_,  
        orders0_.number as number3_1_1_,  
        orders0_.updated_at as updated_4_1_1_  
    from  
        orders orders0_  
    where  
        orders0_.member_id=?  
order size: 10  
Hibernate:  
    select  
        orders0_.member_id as member_i5_1_0_,  
        orders0_.id as id1_1_0_,  
        orders0_.id as id1_1_1_,  
        orders0_.created_at as created_2_1_1_,  
        orders0_.member_id as member_i5_1_1_,  
        orders0_.number as number3_1_1_,  
        orders0_.updated_at as updated_4_1_1_  
    from  
        orders orders0_  
    where  
        orders0_.member_id=?  
order size: 10  
Hibernate:  
    select  
        orders0_.member_id as member_i5_1_0_,  
        orders0_.id as id1_1_0_,  
        orders0_.id as id1_1_1_,  
        orders0_.created_at as created_2_1_1_,  
        orders0_.member_id as member_i5_1_1_,  
        orders0_.number as number3_1_1_,  
        orders0_.updated_at as updated_4_1_1_  
    from  
        orders orders0_  
    where  
        orders0_.member_id=?  
order size: 10  
Hibernate:  
    select
```

```
@Test  
internal fun `즉시 로딩 n+1`() {  
    // fetch = FetchType.EAGER 의 경우  
    val members = memberRepository.findAll()  
}
```

# 지연로딩과 N+1

```
activeProfiles = '{}', propertySourceLocations = '{}', propertySo
.PropertyMappingContextCustomizer@0, org.springframework.boot.tes
.DuplicateJsonObjectContextCustomizerFactory$DuplicateJsonObjectC
'src/main/webapp', contextLoader = 'org.springframework.boot.test
.ServletTestExecutionListener.populatedRequestContextHolder' -> t
Hibernate:
/* select
   generatedAlias0
from
  Member as generatedAlias0 */ select
  member0_.id as id1_0_,
  member0_.created_at as created_2_0_,
  member0_.email as email3_0_,
  member0_.name as name4_0_,
  member0_.updated_at as updated_5_0_
from
  member member0_
2019-10-26 23:48:34.175  INFO 1245 --- [           main] o.s.t.c.t
  경우$jpa_n_plus_1@MemberJpaTest, testException = [null], mergedCo
activeProfiles = '{}', propertySourceLocations = '{}', propertySo
.PropertyMappingContextCustomizer@0, org.springframework.boot.tes
.DuplicateJsonObjectContextCustomizerFactory$DuplicateJsonObjectC
'src/main/webapp', contextLoader = 'org.springframework.boot.test
.ServletTestExecutionListener.populatedRequestContextHolder' -> t
```

```
@Test
internal fun `지연 로딩 n+1`() {
    // fetch = FetchType.LAZY 의 경우
    val members = memberRepository.findAll()
}
```

# 지연로딩과 N+1

```
'org.springframework.test.context.web.ServletTestExecutionListener.resetReq
Hibernate:
/* select
   generatedAlias0
from
  Member as generatedAlias0 */ select
  member0_.id as id1_0_,
  member0_.created_at as created_2_0_,
  member0_.email as email3_0_,
  member0_.name as name4_0_,
  member0_.updated_at as updated_5_0_
from
  member member0_
Hibernate:
select
  orders0_.member_id as member_i5_1_0_,
  orders0_.id as id1_1_0_,
  orders0_.id as id1_1_1_,
  orders0_.created_at as created_2_1_1_,
  orders0_.member_id as member_i5_1_1_,
  orders0_.number as number3_1_1_,
  orders0_.updated_at as updated_4_1_1_
from
  orders orders0_
where
  orders0_.member_id=?
order size : 10
2019-10-27 00:24:26.266 INFO 2388 --- [           main] o.s.t.c.transaction.
```

```
@Test
internal fun `지연로딩인 n+1`() {
    val members = memberRepository.findAll()

    // 회원 한명에 대한 조회는 문제가 없다
    val firstMember = members[0]
    println(firstMember.orders.size)
}
```

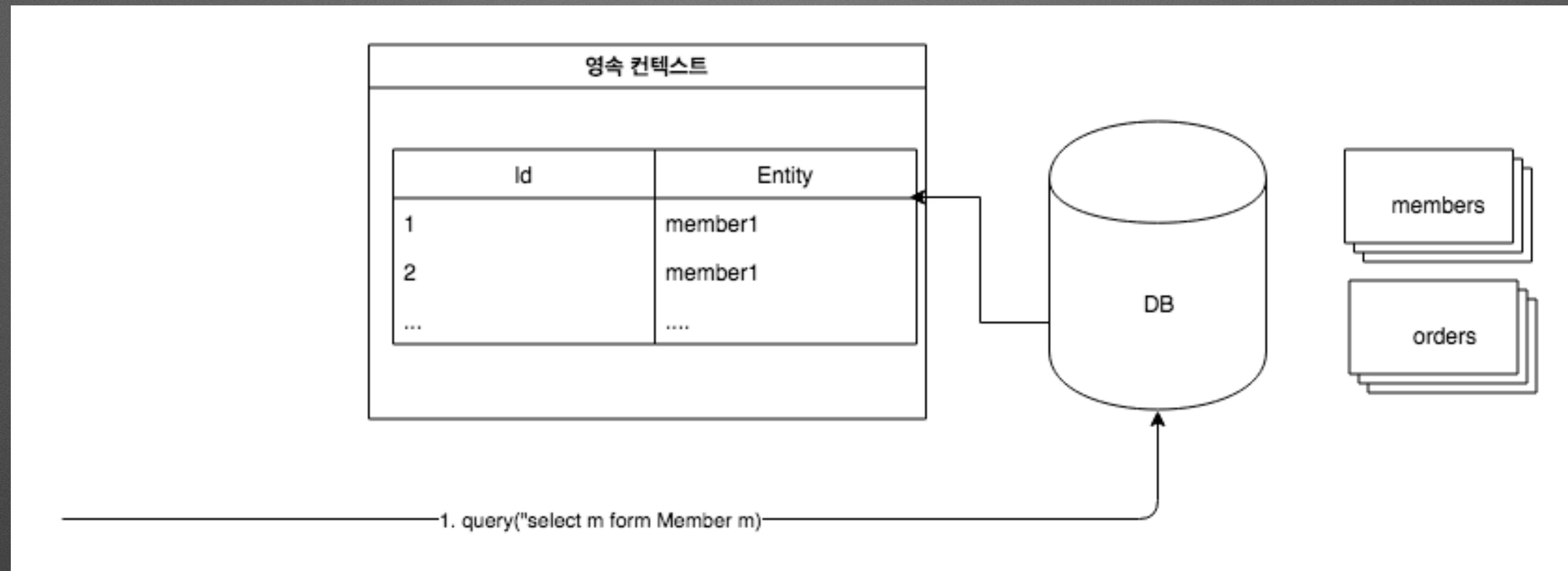
# 지연로딩과 N+1

```
Hibernate:  
    select  
        orders0_.member_id as member_i5_1_0_,  
        orders0_.id as id1_1_0_,  
        orders0_.id as id1_1_1_,  
        orders0_.created_at as created_2_1_1_,  
        orders0_.member_id as member_i5_1_1_,  
        orders0_.number as number3_1_1_,  
        orders0_.updated_at as updated_4_1_1_  
    from  
        orders orders0_  
    where  
        orders0_.member_id=?  
order size: 10  
Hibernate:  
    select  
        orders0_.member_id as member_i5_1_0_,  
        orders0_.id as id1_1_0_,  
        orders0_.id as id1_1_1_,  
        orders0_.created_at as created_2_1_1_,  
        orders0_.member_id as member_i5_1_1_,  
        orders0_.number as number3_1_1_,  
        orders0_.updated_at as updated_4_1_1_  
    from  
        orders orders0_  
    where  
        orders0_.member_id=?  
order size: 10  
Hibernate:  
    select  
        orders0_.member_id as member_i5_1_0_,  
        orders0_.id as id1_1_0_,  
        orders0_.id as id1_1_1_,  
        orders0_.created_at as created_2_1_1_,  
        orders0_.member_id as member_i5_1_1_,  
        orders0_.number as number3_1_1_,  
        orders0_.updated_at as updated_4_1_1_  
    from  
        orders orders0_  
    where  
        orders0_.member_id=?  
order size: 10  
Hibernate:  
    select
```

```
@Test  
internal fun `지연로딩인 n+1`() {  
    val members = memberRepository.findAll()  
  
    // 회원 한명에 대한 조회는 문제가 없다  
    val firstMember = members[0]  
    println("order size : ${firstMember.orders.size}")  
  
    // 조회한 모든 회원에 대해서 조회하는 경우 문제 발생  
    for(member in members){  
        println("order size: ${member.orders.size}")  
    }  
}
```

- 모든 Member에 대해서 주문을 조회하는 경우 N+1문제가 발생합니다.

# N+1원인



- JPQL 특징이 있습니다. `findById()` 같은 경우에는 엔티티를 영속성 컨텍스트에서 먼저 찾고 영속성 컨텍스트에 없는 경우에 데이터베이스에 찾는 반면 JPQL은 항상 데이터베이스에 SQL을 실행해서 결과를 조회합니다. 그리고 아래와 같은 작업을 진행하게 됩니다.
  1. JPQL을 호출하면 데이터베이스에 우선적으로 조회한다.
  2. 조회한 값을 영속성 컨텍스트에 저장한다.
  3. 영속성 컨텍스트에 조회할 때 이미 존재하는 데이터가 있다면(같은 영속성 컨텍스트에서 이미 조회한 유저가 있는 경우) 데이터를 버린다.

# N+1원인

## 즉시 로딩인 경우

```
val members = memberRepository.findAll()
```

JPQL에서 동작한 쿼리를 통해서 members에 데이터가 바인딩 됩니다. 그 이후 JPA에서는 글로벌 패치 전략(즉시 로딩)을 받아들여 해당 member의 연관관계인 order에 대해서 추가적인 페이지 로딩이 진행되어 N+1을 발생시킵니다.

## 지연 로딩인 경우

```
val members = memberRepository.findAll()
```

JPQL에서 동작한 쿼리를 통해서 members에 데이터가 바인딩 됩니다. JPA가 글로벌 패치 전략을 받아들이지만 지연 로딩이기 때문에 추가적인 SQL을 발생시키지 않습니다. 하지만 위에서 본 예제처럼 페이지로 딩으로 추가적인 작업을 진행하게되면 결국 N+1 문제가 발생하게 됩니다.

- JPQL을 실행하면 JPA는 이것을 분석해서 SQL을 생성합니다. JPQL 입장에서는 즉시 로딩, 지연 로딩과 같은 글로벌 패치 전략을 무시하고 JPQL만 사용해서 SQL을 생성합니다.

# 해결 방법 : Batch Size

```
@Entity  
@Table(name = "member")  
class Member private constructor() {  
    ...  
  
    @BatchSize(size = 5) // Batch size를 지정한다  
    @OneToMany(mappedBy = "member", fetch = FetchType.EAGER) // 즉시 로딩으로 설정  
    var orders: List<Order> = emptyList()  
    private set  
}
```

```
Hibernate:  
/* load one-to-many com.example.jpanplus1.member.Member.orders */ select  
    orders0_.member_id as member_i5_1_1_,  
    orders0_.id as id1_1_1_,  
    orders0_.id as id1_1_0_,  
    orders0_.created_at as created_2_1_0_,  
    orders0_.member_id as member_i5_1_0_,  
    orders0_.number as number3_1_0_,  
    orders0_.updated_at as updated_4_1_0_  
from  
    orders orders0_  
where  
    orders0_.member_id in (  
        ?, ?, ?, ?, ?  
    )  
Hibernate:  
/* load one-to-many com.example.jpanplus1.member.Member.orders */ select  
    orders0_.member_id as member_i5_1_1_,  
    orders0_.id as id1_1_1_,  
    orders0_.id as id1_1_0_,  
    orders0_.created_at as created_2_1_0_,  
    orders0_.member_id as member_i5_1_0_,  
    orders0_.number as number3_1_0_,  
    orders0_.updated_at as updated_4_1_0_  
from  
    orders orders0_  
where  
    orders0_.member_id in (  
        ?, ?, ?, ?, ?  
    )
```

- **@BatchSize(size = 5)** 어노테이션을 통해서 설정한 size 만큼 데이터를 미리 로딩 한다. 즉 연관된 엔티티를 조회할때 size 만큼 where in 쿼리를 통해서 조회하게되고 size를 넘어가게 되면 추가로 where in 쿼리를 진행합니다.
- **spring.jpa.properties.hibernate.default\_batch\_fetch\_size=1000** 설정으로 properties 설정으로 글로벌하게 설정 가능

# 해결 방법 : Fetch 조인 사용

```
interface MemberRepository : JpaRepository<Member, Long> {

    @Query(
        "select m from Member m left join fetch m.orders"
    )
    fun findAllWithFetch(): List<Member>
}

@Test
internal fun `페치 조인 사용`() {
    val members = memberRepository.findAllWithFetch()

    // 조회한 모든 회원에 대해서 조회하는 경우에도 N+1 문제가 발생하지 않음
    for (member in members) {
        println("order size: ${member.orders.size}")
    }
}
```

```
hibernate:
  /* select
     m
   from
     Member m
   left join
     fetch m.orders */ select
       member0_.id as id1_0_0_,
       orders1_.id as id1_1_1_,
       member0_.created_at as created_2_0_0_,
       member0_.email as email3_0_0_,
       member0_.name as name4_0_0_,
       member0_.updated_at as updated_5_0_0_,
       orders1_.created_at as created_2_1_1_,
       orders1_.member_id as member_i5_1_1_,
       orders1_.number as number3_1_1_,
       orders1_.updated_at as updated_4_1_1_,
       orders1_.member_id as member_i5_1_0_,
       orders1_.id as id1_1_0_
   from
     member member0_
   left outer join
     orders orders1_
   on member0_.id=orders1_.member_id
order size: 10
```

- 가장 많이 사용하는 방법인 `fetch`을 통해서 조인 쿼리를 진행하는 것입니다. `fetch` 키워드를 사용하게 되면 연관된 엔티티나 컬렉션을 한 번에 같이 조회할 수 있습니다.
- 즉 페치 조인을 사용하게 되면 연관된 엔티티는 프록시가 아닌 실제 엔티티를 조회하게 되므로 연관관계 객체까지 한 번의 쿼리로 가져올 수 있습니다.

# Fetch 조인과 일반 조인 차이

```
interface MemberRepository : JpaRepository<Member, Long> {

    @Query(
        "select m from Member m join m.orders"
    )
    fun findAllWithFetch(): List<Member>
}

@Test
internal fun `페치 조인 키워드 제거`() {
    val members = memberRepository.findAllWithFetch() // 패치 타입 Lazy 경우

    // 패치 조인하지 않은 상태에서는 N+1 문제 발생
    for (member in members) {
        println("order size: ${member.orders.size}")
    }
}
```

```
Hibernate:
/* select
   m
  from
  Member m
 join
  m.orders */ select
    member0_.id as id1_0_,
    member0_.created_at as created_2_0_,
    member0_.email as email3_0_,
    member0_.name as name4_0_,
    member0_.updated_at as updated_5_0_
  from
    member member0_
   inner join
    orders orders1_
      on member0_.id=orders1_.member_id
Hibernate:
select
  orders0_.member_id as member_i5_1_0_,
  orders0_.id as id1_1_0_,
  orders0_.id as id1_1_1_,
  orders0_.created_at as created_2_1_1_,
  orders0_.member_id as member_i5_1_1_,
  orders0_.number as number3_1_1_,
  orders0_.updated_at as updated_4_1_1_
  from
    orders orders0_
   where
     orders0_.member_id=?
order size: 10
Hibernate:
select
  orders0_.member_id as member_i5_1_0_,
  orders0_.id as id1_1_0_,
  orders0_.id as id1_1_1_,
  orders0_.created_at as created_2_1_1_,
  orders0_.member_id as member_i5_1_1_,
  orders0_.number as number3_1_1_,
  orders0_.updated_at as updated_4_1_1_
  from
    orders orders0_
   where
     orders0_.member_id=?
order size: 10
```

- 출력되는 SQL을 보면 조인을 통해서 연관관계 컬렉션까지 함께 조회되는 것으로 생각할 수 있습니다. 하지만 JPQL은 결과를 반환할 때 연관관계 까지 고려하지 않고 select 절에 지정한 엔티티만 조회하게 됩니다.
- 따라서 컬렉션은 초기화하지 않은 컬렉션 레퍼를 반환하게 되고 컬렉션이 없기 때문에 Lazy 로딩이 발생하게 되고 결과적으로 N+1 문제가 발생하게 됩니다.