

JPA

1. EJB라는 ORM을 사용했었는데, 너무 안좋아서 금융권 개발자가 퇴근 후 하이버네이트라는 ORM을 만듦 ⇒ 표준이라는 EJB 보다 하이버네이트가 더 많이 사용되다보니 자바 진영에서 반성하고, 금융권 개발자와 새로운 ORM을 만듦 ⇒ 자바 표준 ORM인 JPA 탄생
2. 컬렉션 동작 방식 지향
3. 패러다임 불일치 문제 해결
 - a. 객체를 다루는 자바, 관계를 다루는 테이블과 행, 이 둘의 불일치 문제를 해결해 줌
4. 부모 객체만 저장해도 연관 관계의 엔티티들도 자동 저장

```
Album album = jpa.persist(albumId);  
  
==>  
INSERT INTO ITEM ...  
INSERT INTO ALBUM ...
```

5. 캐싱
 - a. 같은 트랜잭션 내에서 동일 엔티티를 조회할 경우, 메모리에서 데이터를 가져옴

```
Long id = 100L;  
Member member1 = member.find(id); // DB 조회  
Member member2 = member.find(id); // 이미 조회 됐으므로 메모리에  
  
member1 == member2 // true
```

6. 배치 Insert
 - a. 동일 트랜잭션 내에 Insert가 다수 발생 시 트랜잭션 종료 때, 한 네트워크로 한번에 저장
7. JPQL
 - a. 객체를 대상으로 SQL문을 만듦
 - b. DB에 SQL문과 매우 유사하지만 객체를 대상으로 한다는 것과 DB에 맞는 방언으로 SQL문을 변경해준다는 이점이 존재

8. 쓰기 지연 저장소와 영속성 컨텍스트가 존재

- a. 쓰기 지연 저장소는 Entity의 변화 또는 persist로 엔티티를 저장 시 commit 시점에 DB에 flush 해준다
- b. 영속성 컨텍스트는 @Id를 키로 같은 Map이 있으며, 값으로는 Entity를 갖는다. 또한 스냅샷도 존재하여, 변경된 데이터가 있으면 Update를 하지 않아도 자동적으로 flush 할 때 DB에 업데이트 해준다

9. Flush

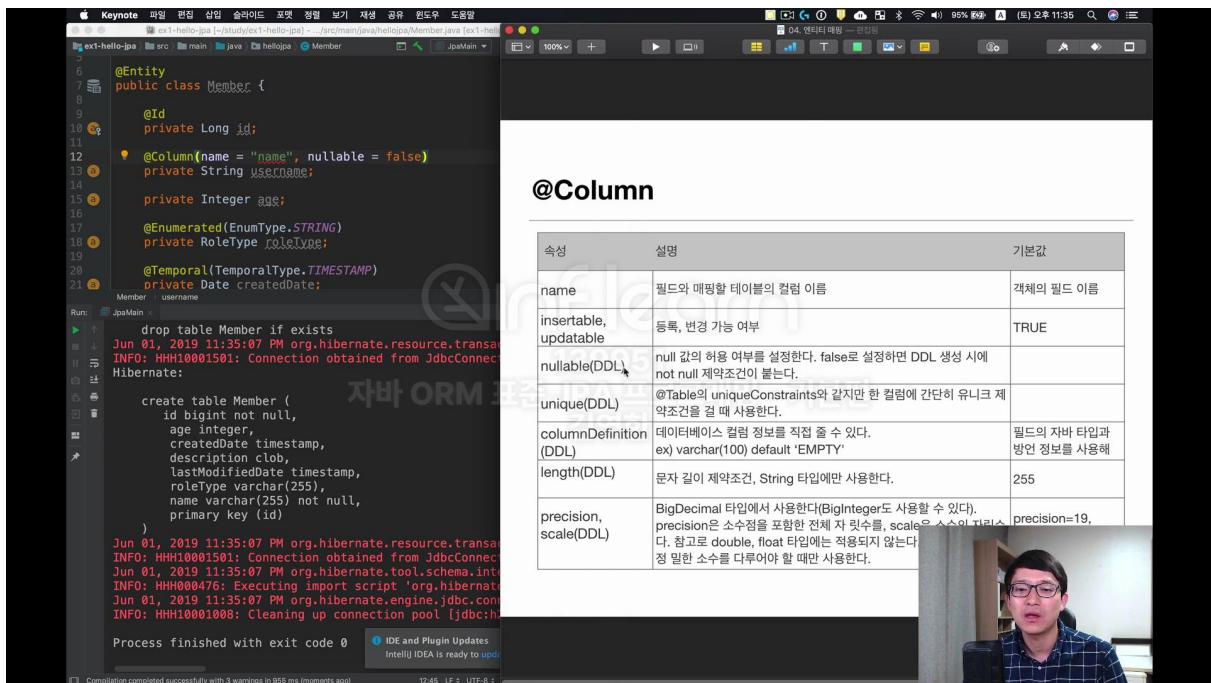
- a. Commit시에 DB에 엔티티들을 반영하는데, em.Flush()를 사용하면 내가 원할 때 DB에 값을 반영할 수 있다.
- b. 영속성 컨텍스트를 비우지 않음

10. 준영속성 컨텍스트

- a. 영속성에 있던 것을 detach() 혹은 clear() 하여 영속성 엔티티가 비영속성으로 변경된 Entity를 말함

11. @Temporal, @Lob, @Transient

- a. @Temporal : 날짜 타입 매핑
- b. @Lob : Varchar를 넘어서는 TEXT 타입과 같이 문자열이 긴 타입
- c. @Transient : 매핑하고 싶지 않은 필드에 적용



- 유니크 제약 조건은 자바에서 설정하기 보다는 DB에서 설정하는 것이 좋다. 왜냐하면 유니크 키의 이름이 랜덤으로 생성되기 때문에 에러 발생 시 어떤 컬럼에서 오류가 나는지 찾기 어렵기 때문이다.

12. JPA에서 다대다는 실무에서 쓰지 않도록 함

- 각각 @ManyToMany로 중간 테이블 없이 매핑할 수 있지만, 만약 중간 테이블에 다른 컬럼들이 존재한다면, JPA에서 표현할 방법이 없음
- 만약 사용한다면 @JoinTable 사용

```
@JoinTable(name = "CATEGORY_ITEM",
           joinColumns = @JoinColumn(name = "category_id"),
           inverseJoinColumns = @JoinColumn(n...
```

13. 1:1 관계에서는 주 테이블에 FK를 넣는 것을 선호

- Order와 Delivery 테이블이 있다면, Order테이블이 조회될 가능성이 높기에 Order 조회 시 Delivery가 있는지 없는지에 대해 한번에 알 수 있음(유무 확인 시 사용, 만약 Delivery에 다른 컬럼을 조회한다면 Delivery에 FK가 있는 것과 별반 다를 거 없음)

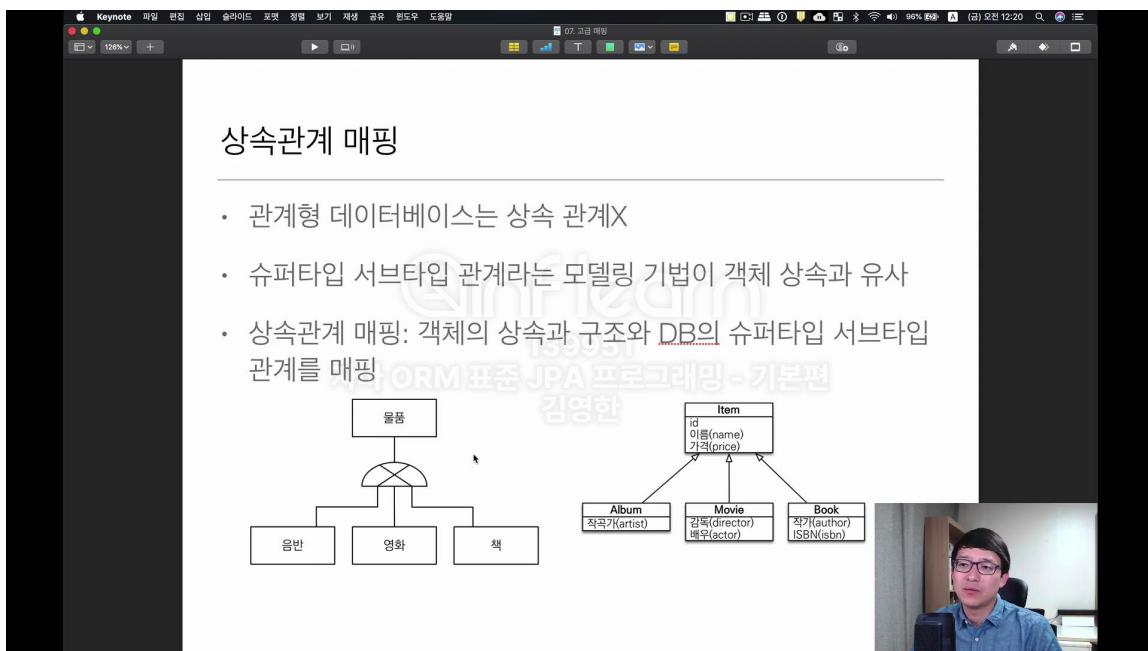
14. 카테고리

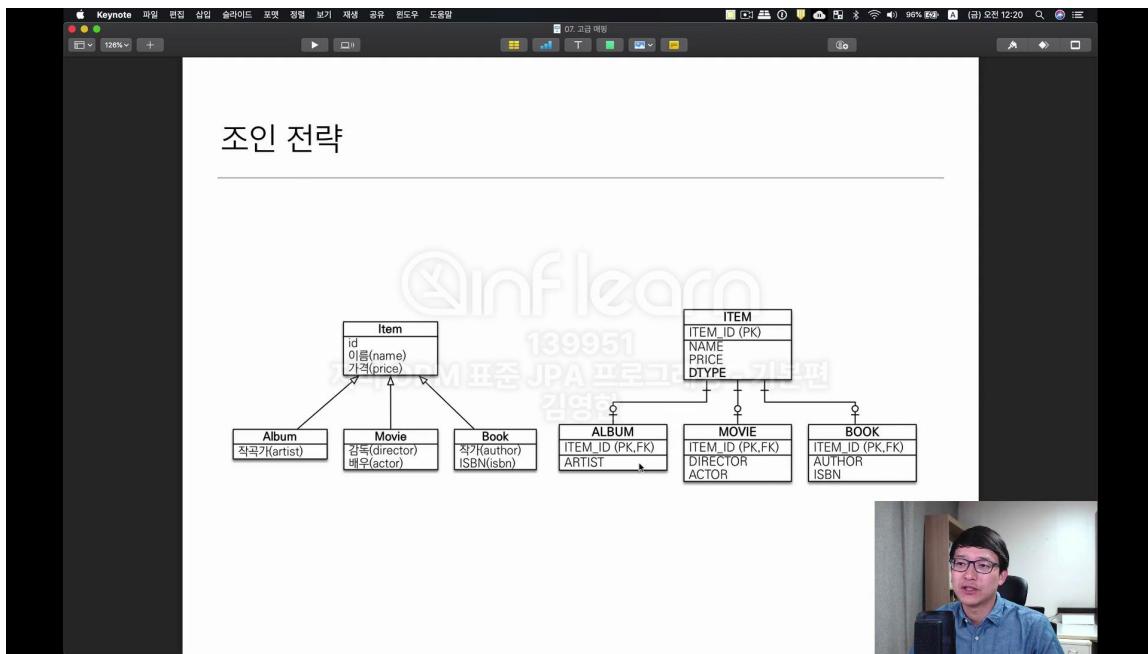
The screenshot shows the IntelliJ IDEA interface with the following details:

- Java Code:** The code is for a `Category` entity in the `jpaMain` package. It includes annotations like `@Entity`, `@Id`, `@GeneratedValue`, and `@ManyToMany`. The code defines fields for `name`, `parent`, and a list of `Category` objects named `child`.
- Database Schema:** A diagram titled "배송, 카테고리 추가 - 엔티티 상세" (Delivery, Category Addition - Entity Detail) shows the database schema. It includes entities for `Category`, `Member`, `Order`, `OrderItem`, `Delivery`, and `Album`. Relationships are shown between `Category` and `Member`, `Category` and `Order`, `Category` and `OrderItem`, `Category` and `Delivery`, and `Category` and `Album`.
- Video Overlay:** A video overlay of a person speaking is visible in the bottom right corner.

- 상위 카테고리, 하위 카테고리와 같이 카테고리의 등급이 존재한다면 엔티티 하나로 관계를 표현할 수 있는 방법이 존재

15. 슈퍼타입과 서브타입





1. 상속관계를 표현한 것으로 “아채”, “고기”, “장”들은 “물품”이라는 큰 카테고리로 묶을 수 있다. 이것을 슈퍼타입, 서브타입 관계라고 한다. 여기서 서브타입은 각각 “가격”, “이름”이라는 컬럼을 가질 수 있고, 이것을 “물품”에 컬럼으로 이관하여 FK를 건다.
2. @Inheritance, @DiscriminatorColumn
 - a. @Inheritance : JPA는 슈퍼타입과 서브타입을 생성할 때, 한 테이블에 몰아 넣는게 기본이다. 이 어노테이션을 통해 하나의 테이블에 컬럼들을 넣을지 두개로 나눌 지 전략을 결정할 수 있다

```
# 조인 전략
@Inheritance(stratgy = InheritanceType.JOINED)
```

====>

Item 테이블(price 컬럼, name 컬럼)
meet 테이블(type 컬럼)

```
# 싱글 테이블 전략
@Inheritance(stratgy = InheritanceType.SINGLE_TABLE)
```

====>

Item 테이블(price 컬럼, name 컬럼, type 컬럼)

- ii. `@DiscriminatorColumn` : Item 테이블에 어떤 컬럼과 매팅이 되는지 확인 가능한 컬럼을 넣어줌

`@DiscriminatorColumns`

====>

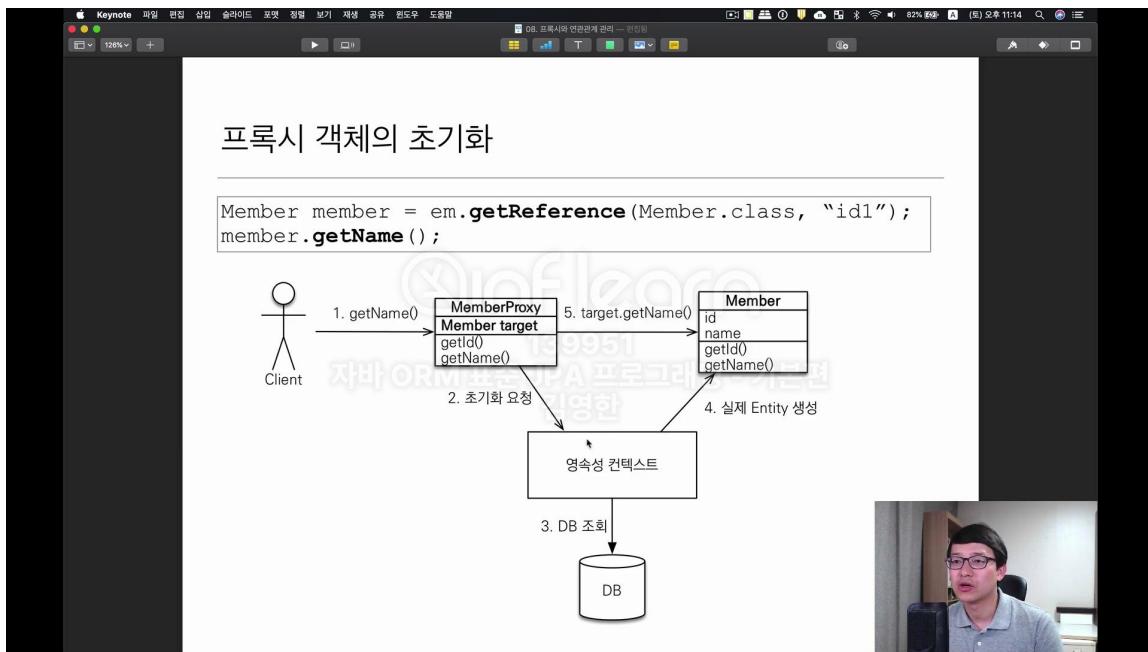
Item 테이블(price 컬럼, name 컬럼, DType 컬럼) -> DType에
meet 테이블(type 컬럼)

16. `@MappedSuperclass`

- a. BaseEntity 만들 때 사용

17. 프록시 객체

- a. Entity를 상속한 껍데기만 있는 Proxy 객체 생성
- b. 이 객체는 Entity 객체를 참조함
- c. 하지만 처음엔 Entity 객체가 없으므로 Member를 조회하면 영속성 컨텍스트에게 DB에서 Entity 값을 가져오라 요청
- d. 영속성 컨텍스트는 DB를 조회한 후 객체 저장
- e. 프록시 객체는 Entity 연결
- f. 만약 `getId()` 요청 시 Entity의 name을 조회해서 반환



- 엔티티 타입 비교 시에는 Entity를 상속받은 프록시 객체가 넘어올 가능성이 다분하기에 == 비교 연산자를 사용하지 말고 instanceof 사용을 권장함

18. 실무에서는 자연 로딩만 사용하라!

- 즉시 로딩은 상상치 못한 쿼리가 나간다
- 만약 즉시 로딩이 필요하다면 fetch 조인이나, 엔티티 그래프를 사용해라

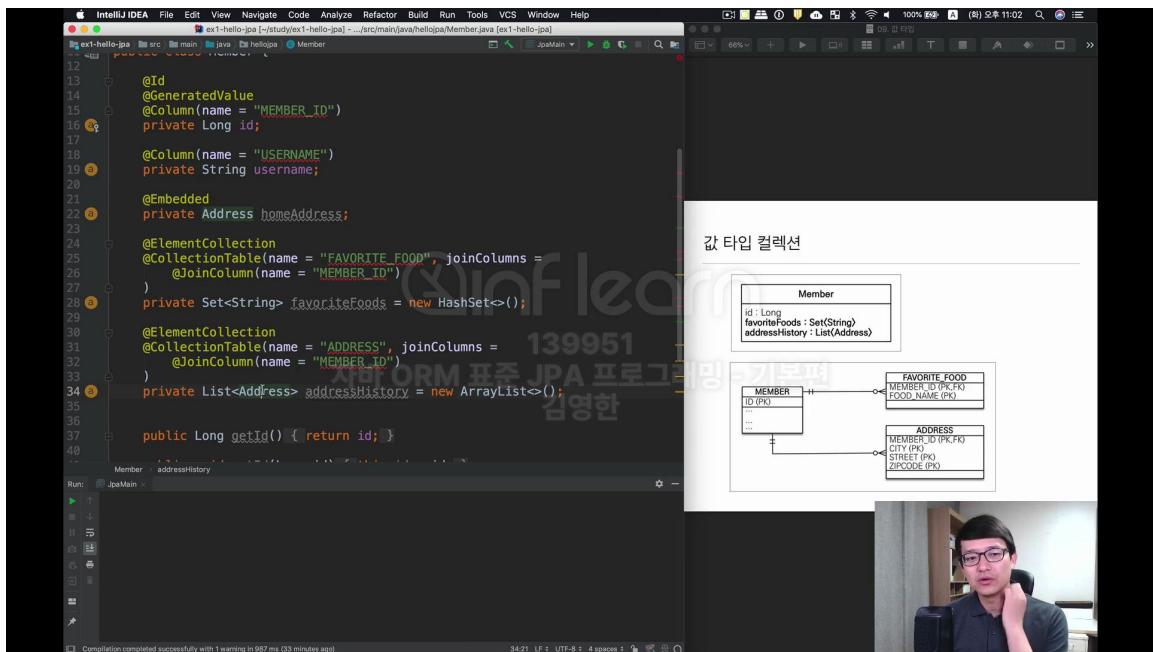
19. Cascade

```
@OneToOne(mappedBy = "parent", cascade = CascadeType.ALL)
private List<Child> childList = new ArrayList<>();
```

- 케스케이드를 지정해주지 않는다면 영속화 하는 경우에서 부모 엔티티 영속화, 자식 엔티티 영속화를 해줘야 한다. 하지만 케스케이드를 지정한 경우 부모만 저장을 해줘도 나머지 속해있는 자식 엔티티들이 자동으로 영속화가 된다.
- 영속성 전이는 연관 관계를 매팅하는 것과 아무 관련이 없음
- 주의사항
 - 자식 엔티티가 부모 엔티티에 귀속돼 있는 경우에만 사용 권장
 - 소유자가 하나 일 경우
 - 만약 자식 엔티티가 여러개의 부모 엔티티에 속해있다고 하면 사용하지 말 것

20. 임베디드

- a. 임베디드 객체는 값 타입으로 활용이 되지만 객체이기 때문에 참조 복사를 할 수 있다. 그렇기에 불변 객체로 선언하는 것을 권장한다.
 - i. 불변이기에 값을 교체해야한다면 new로 새로운 객체를 만들어서 갈아줘야한다.
 - ii. Equals() 구현을 무조건 무조건 무조건 해야함



```
@ElementCollection  
@CollectionTable(name = "MEMBER_ADDRESS_HISTORY", joinColumn =  
private List<Address> addressHistories;
```

- Embedded를 컬렉션으로 지정 시 DB는 컬렉션 개념이 없기 때문에 새로운 값을 담는 테이블을 만들어야 한다. 이때 사용하는 어노테이션들이다.
 - 값 타입 컬렉션과 값 타입은 모두 라이프사이클을 부모에 따른다. 즉 persist에 하지 않아도 자동으로 insert가 발생한다.
 - 영속성을 clear() 하고 조회를 해보면 값 타입 컬렉션 테이블은 자연 로딩이 됨

* 임베디드 컬렉션은 절대 사용하지 말 것

- 식별자가 존재하지 않기 때문에 내가 원하는 쿼리를 만들 수 없음
 - 부모 엔티티가 갖고 있는 임베디드 리스트에 한개 만 값이 변경되더라도, 전체가 삭제되고 인서트 되는 문제 발생
 - 일대다 엔티티로 푸는 게 나음

21. Merge

- a. 병합은 준영속 상태의 객체를 영속 상태로 만들어 줌
- b. 준영속 객체를 DB에 업데이트 해주기 위해 사용하는데, 전체 필드 데이터를 교체함
 - i. 이로 인해 null로 변경될 수 있으니, 사용 금지