

11. JPQL(Java Persistence Query Language, 객체지향쿼리언어)

JPQL = 표준 문법

JPA Criteria + QueryDSL = JPQL을 기반으로 그것을 자바 코드로 짜서 빌드해주는 Generator class의 모음

Native SQL : 표준 SQL 문법을 벗어나는 생쿼리를 날릴 때 사용.

테이블이 아닌 객체를 대상으로 검색함

SQL을 추상화하여 특정 db에 의존하지않음

객체 지향 쿼리 언어

JPQL 문법만 알면 QueryDSL은 거저먹기!

em.flush()는

commit과 createQuery로 쿼리가 날아갈 때 호출된다!!!!

JPQL 문법

- select m from **Member** as m where **m.age** > 18
- 엔티티와 속성은 대소문자 구분O (Member, age)
- JPQL 키워드는 대소문자 구분X (SELECT, FROM, where)
- 엔티티 이름 사용, 테이블 이름이 아님(Member)
- **별칭은 필수(m)** (as는 생략가능)

- TypeQuery : 반환 타입이 명확
- Query : 반환 타입이 불명

결과 조회

결과가 컬렉션이라면 `.getResultList()`

결과가 하나라면 `.getSingleResult()`

결과 조회 API

- `query.getResultList()`: **결과가 하나 이상일 때**, 리스트 반환
 - 결과가 없으면 빈 리스트 반환
- `query.getSingleResult()`: **결과가 정확히 하나**, 단일 객체 반환
 - 결과가 없으면: `javax.persistence.NoResultException`
 - 둘 이상이면: `javax.persistence.NonUniqueResultException`

프로젝션

select 절에 조회할 대상을 정하는 것.

엔티티, 임베디드 타입, 스칼라 타입(기본 데이터 타입)

모두 적용 가능.

프로젝션

- SELECT 절에 조회할 대상을 지정하는 것
- 프로젝트 대상: 엔티티, 임베디드 타입, 스칼라 타입(숫자, 문자 등 기본 데이터 타입)
- SELECT **m** FROM Member m -> 엔티티 프로젝트
- SELECT **m.team** FROM Member m -> 엔티티 프로젝트
- SELECT **m.address** FROM Member m -> 임베디드 타입 프로젝트
- SELECT **m.username, m.age** FROM Member m -> 스칼라 타입 프로젝트
- DISTINCT로 중복 제거

페이징

페이징 API

- JPA는 페이징을 다음 두 API로 추상화
- **setFirstResult**(int startPosition) : 조회 시작 위치 (0부터 시작)
- **setMaxResults**(int maxResult) : 조회할 데이터 수

조인

조인

- 내부 조인:
`SELECT m FROM Member m [INNER] JOIN m.team t`
- 외부 조인:
`SELECT m FROM Member m LEFT [OUTER] JOIN m.team t`
- 세타 조인:
`select count(m) from Member m, Team t where m.username = t.name`

내부 조인 : team이 없고 member만 있으면 한줄이 빠져버리고 멤버만 나온다

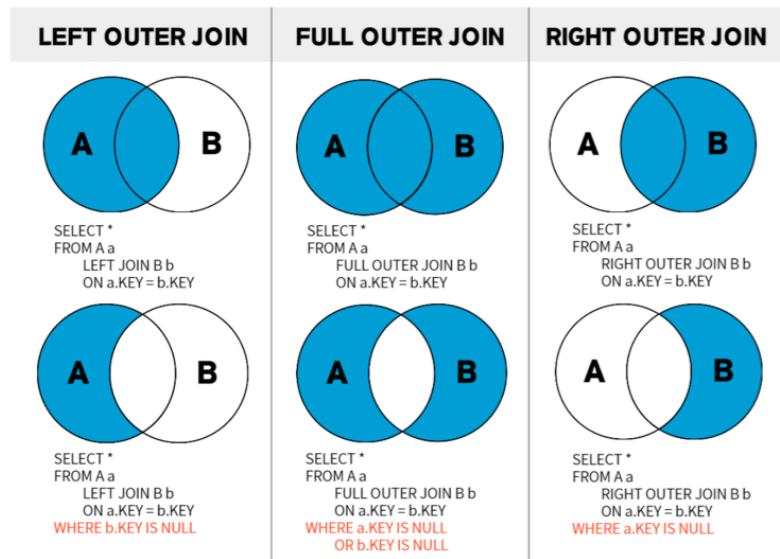
외부 조인 : 위와 똑같은 조건이어도 team은 null로 나오고 member가 나온다

세타 조인 : 연관관계가 아무것도 없는 엔티티 두 개를 조인한다

외부조인

>>

OUTER JOIN



1. 조인 대상 필터링

- 예) 회원과 팀을 조인하면서, 팀 이름이 A인 팀만 조인

JPQL:

```
SELECT m, t FROM Member m LEFT JOIN m.team t on t.name = 'A'
```

SQL:

```
SELECT m.*, t.* FROM  
Member m LEFT JOIN Team t ON m.TEAM_ID=t.id and t.name='A'
```

2. 연관관계 없는 엔티티 외부 조인

- 예) 회원의 이름과 팀의 이름이 같은 대상 외부 조인

JPQL:

```
SELECT m, t FROM  
Member m LEFT JOIN Team t on m.username = t.name
```

SQL:

```
SELECT m.*, t.* FROM  
Member m LEFT JOIN Team t ON m.username = t.name
```

서브 쿼리

서브 쿼리

- 나이가 평균보다 많은 회원
select m from Member m
where m.age > (**select avg(m2.age) from Member m2**)
- 한 건이라도 주문한 고객
select m from Member m
where (**select count(o) from Order o where m = o.member**) > 0

서브 쿼리 지원 함수

- [NOT] EXISTS (subquery): 서브쿼리에 결과가 존재하면 참
 - {ALL | ANY | SOME} (subquery)
 - ALL 모두 만족하면 참
 - ANY, SOME: 같은 의미, 조건을 하나라도 만족하면 참
- [NOT] IN (subquery): 서브쿼리의 결과 중 하나라도 같은 것이 있으면 참

서브 쿼리 - 예제

- 팀A 소속인 회원
select m from Member m
where **exists** (select t from m.team t where t.name = '팀A')
- 전체 상품 각각의 재고보다 주문량이 많은 주문들
select o from Order o
where o.orderAmount > **ALL** (select p.stockAmount from Product p)
- 어떤 팀이든 팀에 소속된 회원
select m from Member m
where m.team = **ANY** (select t from Team t)

JPA 서브 쿼리 한계

- JPA는 WHERE, HAVING 절에서만 서브 쿼리 사용 가능
- SELECT 절도 가능(하이버네이트에서 지원)
- **FROM 절의 서브 쿼리는 현재 JPQL에서 불가능**
 - 조인으로 풀 수 있으면 풀어서 해결

→ 하이버네이트 6에서는 가능!!

```
String query = "select mm.age, mm.uesrname " +  
              "      from (select m.age, m.username from Member m) as mm";  
List<Member> result = em.createQuery(query, Member.class)
```

JPQL 타입 표현

JPQL 타입 표현

- 문자: 'HELLO', 'She"s'
- 숫자: 10L(Long), 10D(Double), 10F(Float)
- Boolean: TRUE, FALSE
- ENUM: jpabook.MemberType.Admin (패키지명 포함)
- 엔티티 타입: TYPE(m) = Member (상속 관계에서 사용)

엔티티 타입 : 상속할 때 DTYPE을 @DiscriminatorValue()에서 설정한 것을 사용할 수 있다.

조건식 - CASE

조건식 - CASE 식

기본 CASE 식

```
select
    case when m.age <= 10 then '학생요금'
         when m.age >= 60 then '경로요금'
         else '일반요금'
    end
from Member m
```

단순 CASE 식

```
select
    case t.name
        when '팀A' then '인센티브110%'
        when '팀B' then '인센티브120%'
        else '인센티브105%'
    end
from Team t
```

조건식 - CASE 식

- COALESCE: 하나씩 조회해서 null이 아니면 반환
- NULLIF: 두 값이 같으면 null 반환, 다르면 첫번째 값 반환

사용자 이름이 없으면 이름 없는 회원을 반환

```
select coalesce(m.username, '이름 없는 회원') from Member m
```

사용자 이름이 '관리자'면 null을 반환하고 나머지는 본인의 이름을 반환

```
select NULLIF(m.username, '관리자') from Member m
```

JPQL 표준 함수

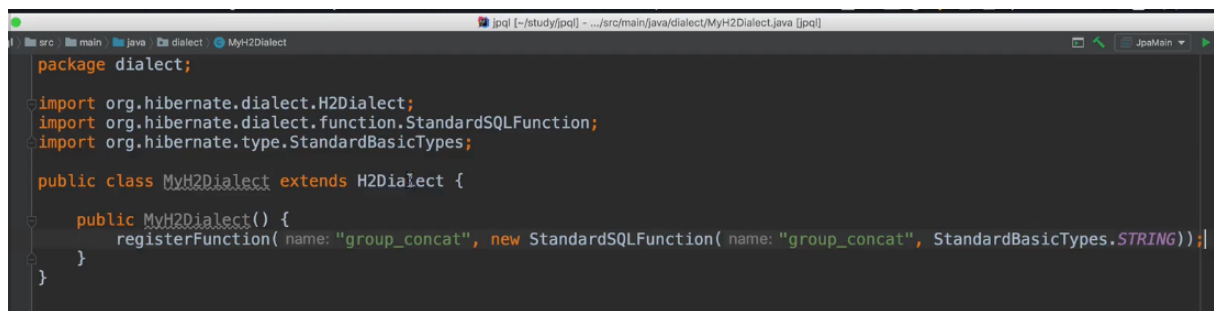
JPQL 기본 함수

- CONCAT
- SUBSTRING
- TRIM
- LOWER, UPPER
- LENGTH
- LOCATE
- ABS, SQRT, MOD
- SIZE, INDEX(JPA 용도)

다행히도 하이버네이트에 방언 별로 DB에서 제공하는 함수가 있음!

- concat : 문자 두개 더하기
- substring : 문자열 잘라내기
- trim : 띄어쓰기 없애기
- lower, upper : 대소문자 바꾸기
- length : 길이
- locate : 위치 리턴. Integer!
- abs, sort, mod : 수학 함수. sql과 동일함
- size, index : size는 컬렉션의 개수, index는 컬렉션에서의 위치

사용자 등록 함수 → 실제 소스코드 열어보면 나옴. 방언 소스코드 보기에서
→ 쓰려면 방언 등록까지 해야



```
package dialect;

import org.hibernate.dialect.H2Dialect;
import org.hibernate.dialect.function.StandardSQLFunction;
import org.hibernate.type.StandardBasicTypes;

public class MyH2Dialect extends H2Dialect {

    public MyH2Dialect() {
        registerFunction( name: "group_concat", new StandardSQLFunction( name: "group_concat", StandardBasicTypes.STRING));
    }
}
```

경로 표현식

- 상태 필드 : 더 이상 . 을 사용해서 더 들어갈 수 없음.
- 단일 값 연관 경로 : 쿼리 튜닝 시 어렵다. 묵시적인 내부 조인이 발생하기 때문에.
- 컬렉션 값 연관 경로 : 묵시적 내부 조인 발생.
 - 컬렉션이기 때문에 내부 메소드가 없음. 하지만 컬렉션의 메소드인 size등은 사용이 가능함.
 - From 절에서 명시적 조인을 통해 별칭을 얻으면 별칭으로 탐색이 가능함.

```
String query = "select m.username from Member m"; // 경로 표현식 사용 -> 상태 필드 사용
```

```
String query = "select m.team From Member m"; // 경로 표현식 사용 시 주의 -> 단일 값 연관 경로 조회 시 묵시적 내부 조인 발생
```

```
String query = "select t.members.username From Team t"; // 경로 표현식 사용 시 주의
```

-> 컬렉션 값 연관 경로 조회 시 묵시적 내부 조인 발생. 실행 시 오류

```
Collection result = em.createQuery(query, Collection.class).getResultList();
```

```
String query = "select t.members from Team t join t.members m"; // 명시적 조인을 통한 조회
```

연관 경로들은 항상 묵시적 내부 조인 발생함!!

=> 실무에서는 묵시적 내부 조인 쓰지말자!

Fetch 조인

정말정말 중요!!!!!!!!!!!!!!

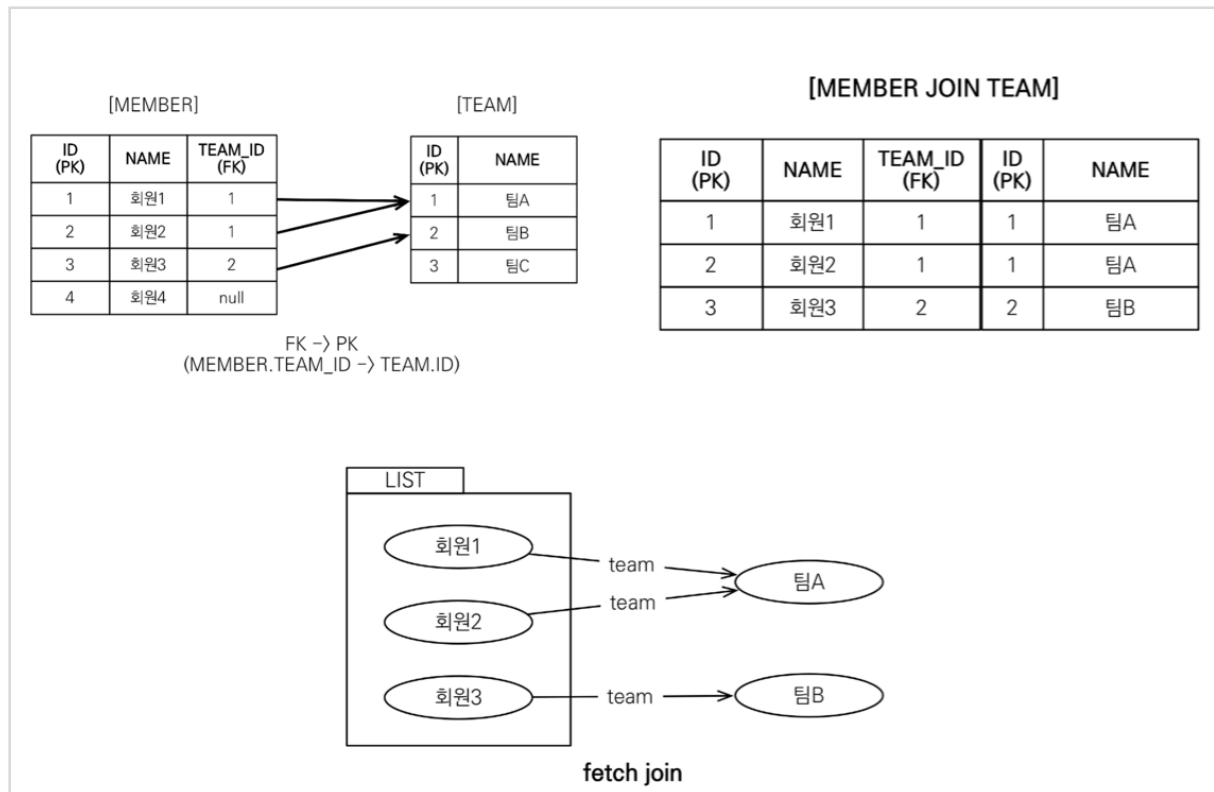
연관된 엔티티나 컬렉션을 sql 한번에 함께 조회.

left fetch join 방식으로 사용

엔티티 페치 조인

- 회원을 조회하면서 연관된 팀도 함께 조회(SQL 한 번에)
- SQL을 보면 회원 뿐만 아니라 팀(T.*)도 함께 **SELECT**
- **[JPQL]**
select m from Member m **join fetch** m.team
- **[SQL]**
SELECT M.*, **T.*** FROM MEMBER M
INNER JOIN TEAM T ON M.TEAM_ID=T.ID

=> 이거 즉시 로딩과 똑같다. 다른 점은 쿼리로 내가 원하는 것을 명확하게 명시 가능.



member와 team을 inner join하면
영속성 컨텍스트에는 회원13, 팀AB가 존재.

```
String query = "select m from Member m join fetch m.team";
```

=> Team을 프록시가 아닌 실제 엔티티로 가져옴.

=> LAZY를 유지하되, N + 1 문제로 쿼리가 많이 찍히는 것을 방지하기 위함.

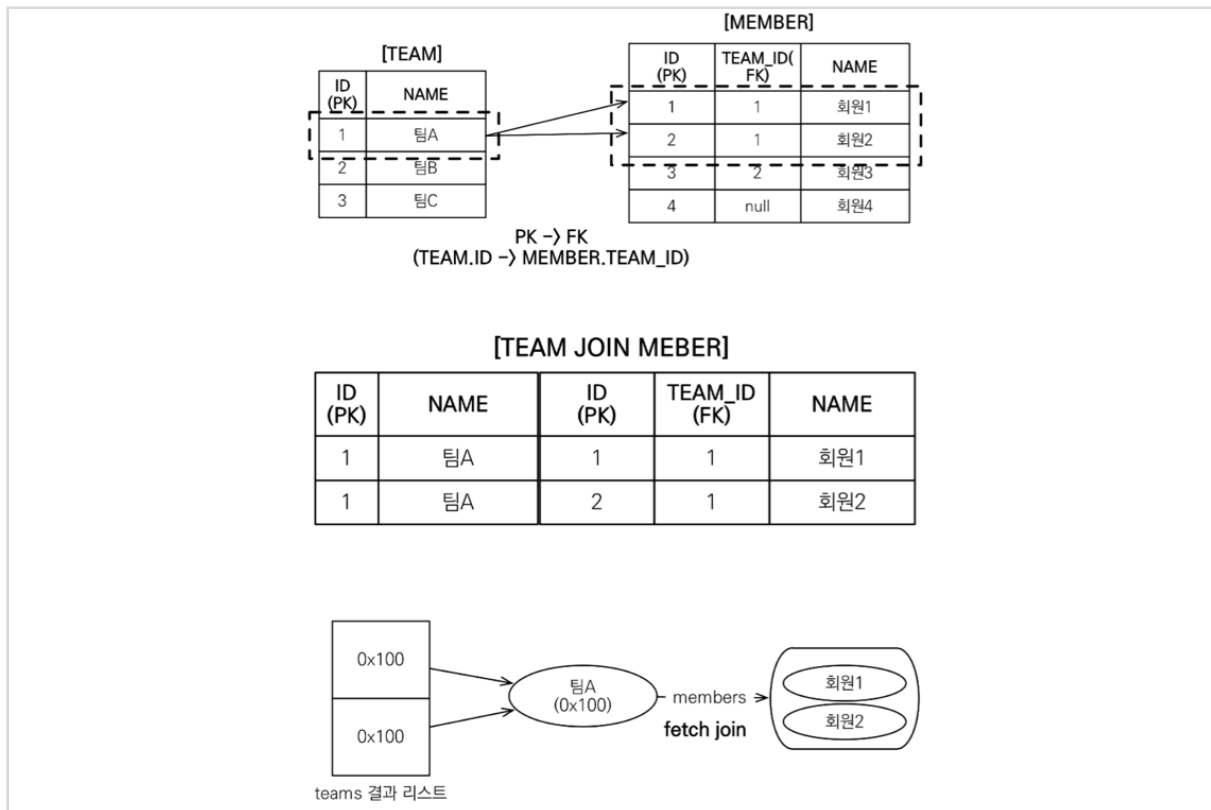
1 + N 문제 :

- 처음에 가져온 member 객체에 대한 쿼리
- member들에 대한 team을 가져오는데 이것에 대한 쿼리가 N번 나갈 것임.

=> 1 + N

=> 지연 로딩을 했어도 페치 조인이 더 우선순위가 높음

컬렉션 페치 조인



일대다 조인 시 데이터 뺏튀기는 불가피.

팀A 입장에서는 한줄인데 멤버가 두 명이기 때문에 행이 두개가 나와야함.

일대다 조인 시 jpa도 줄이 몇개가 나갈지 모름..

페치 조인과 distinct

- sql에 distinct 추가
- 애플리케이션에서 중복 제거

distinct로 중복 제거 안함

```
team = teamA members = 2
-> member = Member{id=3, username='member1', age=0}
-> member = Member{id=4, username='member2', age=0}
team = teamA members = 2
-> member = Member{id=3, username='member1', age=0}
-> member = Member{id=4, username='member2', age=0}
team = teamB members = 1
-> member = Member{id=5, username='member3', age=0}
```

distinct로 중복 제거

```
team = teamA members = 2
-> member = Member{id=3, username='member1', age=0}
-> member = Member{id=4, username='member2', age=0}
team = teamB members = 1
-> member = Member{id=5, username='member3', age=0}
```

다대일은 그냥 조인해도 뺄필요 없으므로 그냥 사용해도 됨!!!

Fetch join VS join

- 페치 조인은 실제로 데이터를 퍼올림.
=> EAGER 형식이라고 보면 편하다! 하지만 얻어오고 싶은 데이터에 대해 더 명확한 표시가 가능하다.
(팀 엔티티 + 멤버 엔티티 둘 다 끌어오기 => 팀, 멤버 둘 다 조회 가능)
- 조인은 실제로 데이터를 퍼올리지는 않고 그냥 조인만 실행.
=> LAZY 형식으로 설정했다면 엔티티를 끌어오지 않고 조인만 실행해서 하나의 엔티티만 조회할 수 있다.
(팀 엔티티 join 멤버 엔티티 했을 때 멤버만 끌어와서 멤버만 조회 가능)

fetch join의 한계

- 페치 조인 대상에는 별칭을 줄 수 없다.
 - jpa 객체 그래프 특성 상 Team에서 조건 만족하는 Member 3명 뽑으려면 Team에 있는 멤버를 100명 다 조회한 다음 거기서 조건 만족하는 3명 뽑음.
 - 너무 별로인 방법이므로 처음부터 그냥 member에서 조건문 뽑는게 나음.
 - 따라서, 두 쿼리를 비교할 때
 - 하나는 Team에서 조건 만족하는 Member 뽑음
 - 하나는 Member에서 뽑음
 - 옵션이 매우 많이 발라져있다면 데이터가 이상하게 됨.
 - 영속성 컨텍스트 입장에서도 매우 애매해짐
 - 어떤 것은 5개만 뽑아왔고 어떤 것은 100개 다 뽑아옴

fetch join을 연쇄적으로 쓸 때 아니면 별칭은 쓰지말자. (정합성 이슈!)

- 둘 이상의 컬렉션은 페치 조인 할 수 없다.
 - 엔티티 : 컬렉션 페치 조인도 일대다인데 엔티티 : 컬렉션 : 컬렉션 페치 조인은 일대다대다라서 너무 어렵다!
- 컬렉션을 페치 조인하면 페이징을 할 수 없다.
 - `setFirstResult, setMaxResult` 는 사용 불가.
 - 일대다 페치 조인 시 데이터 뺏기기 현상 생각해보기. 만약 두개로 뺏기기됐는데 데이터 하나만 갖고오라고 하면 모든 데이터를 드러내지 않게 되므로 X.
 - 해결 방법 : query의 자리 바꾸기. 즉 일대다 → 다대일로 바꾸기! 다대일은 페이징이 문제가 없으므로... 왜냐면 다대일에 페치 조인을 해도 어차피 뺏기기 없을수밖에없음

Named 쿼리

어플리케이션 로딩 시점에 초기화 후 재사용 할 수 있음.

정적 쿼리가 변하지 않기 때문에 JPA가 로딩 시점에 아예 SQL을 캐시함.

=> 검증 가능하며 초기화 후 재사용.

SpringDataJpa를 쓸 때 더 잘 쓸 수 있음.

벌크 연산

일반적으로 아는 sql의 업데이트, 딜리트.

pk를 딱 찍어서 하는것 제외한 업데이트, 딜리트

벌크 연산

- 재고가 10개 미만인 모든 상품의 가격을 10% 상승하려면?
- JPA 변경 감지 기능으로 실행하려면 너무 많은 SQL 실행
 - 1. 재고가 10개 미만인 상품을 리스트로 조회한다.
 - 2. 상품 엔티티의 가격을 10% 증가한다.
 - 3. 트랜잭션 커밋 시점에 변경감지가 동작한다.
- 변경된 데이터가 100건이라면 100번의 UPDATE SQL 실행

벌크 연산 주의!

- 벌크 연산은 영속성 컨텍스트 무시.
 - => 벌크 연산을 먼저 실행하면 어차피 영속성 컨텍스트를 무시하므로 영속성 컨텍스트 만지지도 않고 가니까 상관없음
 - 벌크연산 수행 후 영속성 컨텍스트 초기화