- **Java Persistence 쿼리 언어**
  - **기본 문법과 쿼리 API**
    - - SELECT 문
    - - TypeQuery, Query
    - - 결과 조회
  - **파라미터 바인딩**
    - - 이름 기준 파라미터 (Named parameters)
    - - 위치 기준 파라미터 (Positional parameters)
  - **프로젝션**
    - - 엔티티 프로젝션
    - - 임베디드 타입 프로젝션
    - - 스칼라 타입 프로젝션
    - - 여러 값 조회
    - - NEW 명령어
  - **페이징 API**
  - **집합과 정렬**
    - - 집합 함수
    - - GROUP BY, HAVING
    - - 정렬 (ORDER BY)
  - **JPQL 조인**
    - - 내부 조인
    - - 외부 조인

        - - 컬렉션 조인
        - - 세타 조인
        - - JOIN ON 절 (JPA 2.1)
    - **페치 조인**
        - - 엔티티 페치 조인
        - - 컬렉션 페치 조인
        - - 페치 조인과 DISTINCT
        - - 페치 조인과 일반 조인의 차이
        - - 페치 조인의 특징과 한계
    - **경로 표현식(Path Expression)**
        - - 경로 표현식의 용어정리
        - - 경로 탐색
            - – 상태 필드 경로 탐색
            - – 단일 값 연관 경로 탐색
            - – 컬렉션 값 연관 경로 탐색
            - – 경로 탐색을 사용한 묵시적 조인시 주의사항
    - **서브 쿼리**
        - - 서브 쿼리 함수
            - – EXISTS
            - – {ALL | ANY | SOME}
            - – IN
    - **조건식(Conditional Expression)**

        - - 타입 표현
        - - 논리 연산(조건식 합성)
        - - 비교식
        - - 연산자 우선 순위
        - - Between 식
        - - IN 식
        - - Like 식
        - - NULL 비교식
        - - 컬렉션 식
        - - 스칼라 식
            - – 수학 식
            - – 문자함수
            - – 수학함수
            - – 날짜함수
        - - CASE 식(Case Expression)
    - **다형성 쿼리**
        - - TYPE
        - - TREAT (JPA 2.1)
    - **사용자 정의 함수 호출 (JPA 2.1)**
    - **기타 정리**
        - - EMPTY STRING
        - - NULL 정의

- **엔티티를 직접 사용하기**
  - - 기본 키 값
  - - 외래 키 값
- **Named 쿼리 - 정적 쿼리**
  - - Named 쿼리를 어노테이션에 정의하기
  - - Named 쿼리를 XML에 정의하기
  - - 환경에 따른 설정

Java Persistence 쿼리 언어

객체 지향 쿼리 소개 장에서 엔티티를 쿼리하는 다양한 방법을 소개했다. 어떤 방법을 사용하든 Java Persistence Query Language 줄여서 JPQL에서 모든 것이 시작한다. 이론은 소개 장에서 이미 설명했으니 여기서는 JPQL의 사용방법 위주로 설명하겠다.

시작하기 전에 JPQL의 특징을 다시 한번 정리해보자.

- JPQL은 객체 지향 쿼리 언어다. 따라서 테이블을 대상으로 쿼리하는 것이 아니라 엔티티 객체를 대상으로 쿼리한다.
- JPQL은 SQL을 추상화해서 특정 데이터베이스 SQL에 의존하지 않는다.
- JPQL은 결국 SQL로 변환된다.

샘플 도메인 모델

시작하기 전에 이번 장에서 예제로 사용할 도메인 모델을 살펴보자.

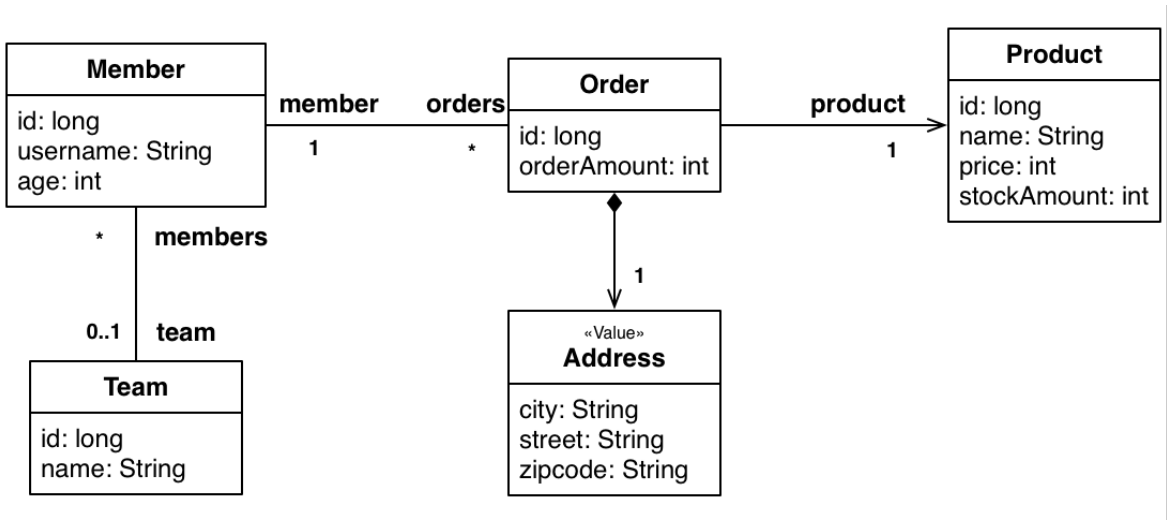

그림 9 I 샘플 모델 UML

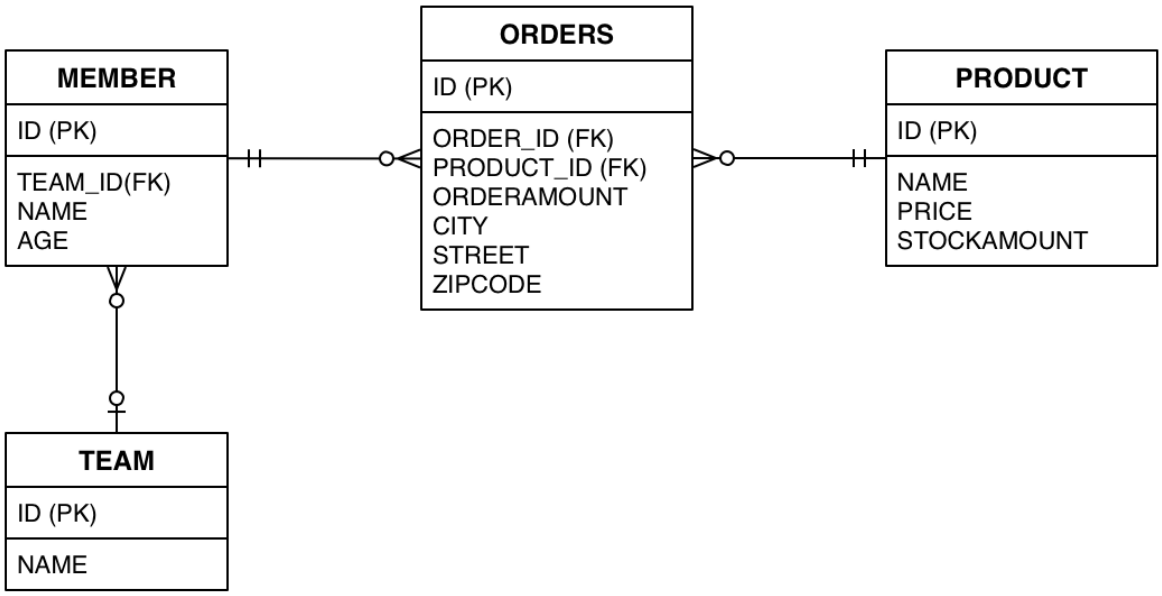

그림 9 I 샘플 모델 ERD

[그림 - 샘플 모델 UML]과 [그림 - 샘플 모델 ERD]를 보자. 실무에서 사용하는 주문 모델링은 더 복잡하지만 JPQL의 이해가 목적이므로 단순화했다. 여기서는 회원이 상품을 주문하는 다대다 관계라는 것을 특히 주의해서 보자. 그리고 `Address`는 임베디드 타입인데 이것은 값 타입이므로 UML에서 스테레오 타입을 사용해 `<<Value>>`로 정의했다. 이것은 ERD를 보면 `ORDERS` 테이블에 포함되어 있다.

기본 문법과 쿼리 API

JPQL도 SQL과 비슷하게 SELECT, UPDATE, DELETE 문을 사용할 수 있다. 참고로 엔티티를 저장할 때는 `EntityManager.persist()` 메서드를 사용하면 되므로 INSERT 문은 없다.

====== JPQL 문법 ======

```
select_문 :: =
    select_절
    from_절
    [where_절]
    [groupby_절]
    [having_절]
    [orderby_절]

update_문 :: = update_절 [where_절]
delete_문 :: = delete_절 [where_절]
```

JPQL 문법을 보면 전체 구조는 SQL과 비슷한 것을 알 수 있다. JPQL에서 UPDATE, DELETE 문은 벌크 연산이라 하는데 객체지향 쿼리 - 고급주제 장에서 설명하겠다. 지금부터 SELECT 문을 자세히 알아보자.

- SELECT 문

SELECT 문은 다음과 같이 사용한다.

```
SELECT m FROM Member AS m where m.username = 'Hello'
```

대소문자 구분

엔티티와 속성은 대소문자를 구분한다. 예를 들어 `Member`, `username`은 대소문자를 구분한다. 반면에 `SELECT`, `FROM`, `AS` 같은 JPQL 키워드는 대소문자를 구분하지 않는다.

엔티티 이름

JPQL에서 사용한 `Member`는 클래스 명이 아니라 엔티티 명이다. 엔티티 명은 `@Entity(name="XXX")`로 지정할 수 있다. 엔티티 명을 지정하지 않으면 클래스 명을 기본값으로 사용한다. 기본값인 클래스 명을 엔티티 명으로 사용하는 것을 추천한다.

별칭은 필수

`Member AS m`을 보면 `Member`에 `m`이라는 별칭을 주었다. JPQL은 별칭을 필수로 사용해야 한다. 따라서 다음 코드처럼 별칭 없이 작성하면 잘못된 문법이라는 오류가 발생한다.

```
SELECT username FROM Member m //잘못된 문법, username을 m.username으로 고쳐야 한다.
```

`AS`는 생략할 수 있다. 따라서 `Member m`처럼 사용해도 된다.

> **참고:** 하이버네이트는 JPQL 표준도 지원하지만 더 많은 기능을 가진 HQL(Hibernate Query Language)을 제공한다. JPA 구현체로 하이버네이트를 사용하면 HQL도 사용할 수 있다. HQL은 `SELECT username FROM Member m`의 `username`처럼 별칭 없이 사용할 수 있다.

> **참고:** JPA 표준 명세는 별칭을 식별 변수(Identification variable)라는 용어로 정의했다. 하지만 보통 별칭(alias)이라는 단어가 익숙하므로 별칭으로 부르겠다.

- TypeQuery, Query

작성한 JPQL을 실행하려면 쿼리 객체를 만들어야 한다. 쿼리 객체는 `TypeQuery`와 `Query`가 있는데 반환할 타입을 명확하게 지정할 수 있으면 `TypeQuery` 객체를 사용하고, 반환 타입을 명확하게 지정할 수 없으면 `Query` 객체를 사용하면 된다. 다음 예제로 알아보자.

====== `TypeQuery` 사용 ======

```
TypedQuery<Member> query = em.createQuery("SELECT m FROM Member m", Member.class
List<Member> resultList = query.getResultList();
for (Member member : resultList) {
    System.out.println("member = " + member);
}
```

`em.createQuery()`의 두 번째 파라미터에 반환할 타입을 지정하면 `TypeQuery`를 반환하고 지정하지 않으면 `Query`를 반환한다. 조회 대상이 `Member` 엔티티이므로 조회 대상 타입이 명확하다. 이때는 `TypeQuery`를 사용할 수 있다.

====== `Query` 사용 ======

```
Query query = em.createQuery("SELECT m.username, m.age from Member m"); //**
List resultList = query.getResultList();

for (Object o : resultList) {
    Object[] result = (Object[]) o; //결과가 둘 이상이면 Object[] 반환
    System.out.println("username = " + result[0]);
    System.out.println("age = " + result[1]);
```

```
}
```

조회 대상이 회원 이름과 나이 이므로 조회 대상 타입이 명확하지 않다. 이때는 `Query`를 사용해야 한다. 이처럼 SELECT 절에서 여러 엔티티나 컬럼을 선택할 때는 반환할 타입이 명확하지 않으므로 `Query` 객체를 사용해야 한다.

`Query` 객체는 SELECT 절의 조회 대상이 예제처럼 둘 이상 이면 `Object[]`을 반환하고 SELECT 절의 조회 대상이 하나면 `Object`를 반환한다. 예를 들어 `SELECT m.username from Member m`이면 결과를 `Object`로 반환하고 `SELECT m.username, m.age from Member m`이면 `Object[]`을 반환한다.

두 코드를 비교해보면 타입을 변환할 필요가 없는 `TypeQuery`를 사용하는 것이 더 편리한 것을 알 수 있다.

- 결과 조회

다음 메서드들을 호출하면 실제 쿼리를 실행해서 데이터베이스를 조회한다.

- `query.getResultList()` : 결과를 리스트로 반환한다. 만약 결과가 없으면 빈 컬렉션을 반환한다.
- `query.getSingleResult()` : 결과가 정확히 하나일 때 사용한다.
  - 결과가 없으면 : `javax.persistence.NoResultException` 예외가 발생한다.
  - 결과과 1개보다 많으면 : `javax.persistence.NonUniqueResultException` 예외가 발생한다.

`getSingleResult()`는 결과가 정확히 1개가 아니면 예외가 발생한다는 점에 주의해야 한다.

```
Member member = query.getSingleResult();
```

파라미터 바인딩

JDBC는 위치 기준 파라미터 바인딩만 지원하지만 JPQL은 이름 기준 파라미터 바인딩도 지원한다.

- 이름 기준 파라미터 (Named parameters)

이름 기준 파라미터는 파라미터를 이름으로 구분하는 방법이다. 이름 기준 파라미터는 앞에 `:`를 사용한다.

```
String usernameParam = "User1";

TypedQuery<Member> query =
  em.createQuery("SELECT m FROM Member m where m.username=:username", Member.cla

query.setParameter("username", usernameParam);
List<Member> resultList = query.getResultList();
```

예제 코드의 JPQL을 보면 `:username`이라는 이름 기준 파라미터를 정의하고 `query.setParameter()`에서 `username`이라는 이름으로 파라미터를 바인딩 한다.

참고로 JPQL API는 대부분 메서드 체인 방식으로 설계되어 있어서 다음과 같이 연속해서 작성할 수 있다.

```
List<Member> members =
  em.createQuery("SELECT m FROM Member m where m.username = :username", Member.c
    .setParameter("username", usernameParam)
    .getResultList();
```

- 위치 기준 파라미터 (Positional parameters)

위치 기준 파라미터를 사용하려면 `?` 다음에 위치 값을 주면 된다. 위치 값은 1부터 시작한다.

```
List<Member> members =
    em.createQuery("SELECT m FROM Member m where m.username = ?1", Member.class)
        .setParameter(1, usernameParam)
        .getResultList();
```

위치 기준 파라미터 방식보다는 **이름 기준 파라미터 바인딩 방식을 사용하는 것이 더 명확하다.**

> **참고:** JPQL을 수정해서 다음 코드처럼 파라미터 바인딩 방식을 사용하지 않고 직접 문자를 더해 만들어 넣으면 악의적인 사용자에 의해 SQL 인젝션 공격을 당할 수 있다.

> 또한 성능이슈도 있는데 파라미터 바인딩 방식을 사용하면 파라미터의 값이 달라도 같은 쿼리로 인식해서 JPA는 JPQL을 SQL로 파싱한 결과를 재사용할 수 있다. 그리고 데이터베이스도 내부에서 실행한 SQL을 파싱해서 사용하는데 같은 쿼리는 파싱한 결과를 재사용할 수 있다. 결과적으로 애플리케이션과 데이터베이스 모두 해당 쿼리의 파싱 결과를 재사용할 수 있어서 전체 성능이 향상된다. 따라서 **파라미터 바인딩 방식은 선택이 아닌 필수다.**

```
//파라미터 바인딩 방식을 사용하지 않고 직접 JPQL을 만들면 위험하다.
"select m from Member m where m.username = '" + usernameParam + "'"
```

프로젝션

SELECT 절에 조회할 대상을 지정하는 것을 프로젝션이라 하고 `[SELECT {프로젝션 대상} FROM]`으로 대상을 선택한다. 프로젝션 대상은 엔티티, 엠비디드 타입, 스칼라 타입이 있다. 스칼라 타입은 숫자, 문자 등 기본 데이터 타입을 뜻한다.

- 엔티티 프로젝션

```
SELECT m FROM Member m       //회원
SELECT m.team FROM Member m //팀
```

처음은 회원을 조회했고 두 번째는 회원과 연관된 팀을 조회했는데 둘 다 엔티티를 프로젝션 대상으로 사용했다. 쉽게 생각하면 원하는 객체를 바로 조회한 것인데 컬럼을 하나하나 나열해서 조회해야 하는 SQL과는 차이가 있다.

참고로 이렇게 **조회한 엔티티는 영속성 컨텍스트에서 관리된다.**

- 임베디드 타입 프로젝션

JPQL에서 임베디드 타입은 엔티티와 거의 비슷하게 사용된다. 임베디드 타입은 조회의 시작점이 될 수 없다는 제약이 있다.

다음은 임베디드 타입인 `Address`를 조회의 시적점으로 사용해서 잘못된 쿼리다.

```
String query = "SELECT a FROM Address a";
```

다음 코드에서 `Order` 엔티티가 시작점이다. 이렇게 엔티티를 통해서 임베디드 타입을 조회할 수 있다.

```
String query = "SELECT o.address FROM Order o";
List<Address> addresses = em.createQuery(query, Address.class)
                            .getResultList();
```

====== 실행된 SQL ======

```
select
    order.city,
    order.street,
    order.zipcode
from
    Orders order
```

임베디드 타입은 엔티티 타입이 아닌 값 타입이다. 따라서 이렇게 직접 조회한 임베디드 타입은 영속성 컨텍스트에서 관리되지 않는다.

- 스칼라 타입 프로젝션

숫자, 문자, 날짜와 같은 기본 데이터 타입들을 스칼라 타입이라 한다.

예를 들어 전체 회원의 이름을 조회하려면 다음처럼 쿼리하면 된다.

```
List<String> usernames =
  em.createQuery("SELECT username FROM Member m", String.class)
    .getResultList();
```

중복 데이터를 제거하려면 `DISTINCT`를 사용한다.

```
SELECT DISTINCT username FROM Member m
```

다음과 같은 통계 쿼리도 주로 스칼라 타입으로 조회한다. 통계 쿼리용 함수들은 뒤에서 설명하겠다.

```
Double orderAmountAvg =
  em.createQuery("SELECT AVG(o.orderAmount) FROM Order o", Double.class)
```

```
    .getSingleResult();
```

- 여러 값 조회

엔티티를 대상으로 조회하면 편리하겠지만, 꼭 필요한 데이터들만 선택해서 조회해야 할 때도 있다.

프로젝션에 여러 값을 선택하면 `TypeQuery`를 사용할 수 없고 대신에 `Query`를 사용해야 한다.

```
Query query = em.createQuery("SELECT m.username, m.age FROM Member m");
List resultList = query.getResultList();

Iterator iterator = resultList.iterator();
while (iterator.hasNext()) {
    Object[] row = (Object[]) iterator.next();
    String username = (String) row[0];
    Integer age = (Integer) row[1];
}
```

제네릭에 `Object[]`을 사용하면 다음 코드처럼 조금 더 간결하게 개발할 수 있다.

```
List<Object[]> resultList =
  em.createQuery("SELECT m.username, m.age FROM Member m")
    .getResultList();

for (Object[] row : resultList) {
    String username = (String) row[0];
    Integer age = (Integer) row[1];
}
```

스칼라 타입뿐만 아니라 엔티티 타입도 여러 값을 함께 조회할 수 있다.

```
List<Object[]> resultList =
    em.createQuery("SELECT o.member, o.product, o.orderAmount FROM Order o")
      .getResultList();

for (Object[] row : resultList) {
    Member member = (Member) row[0];     //엔티티
    Product product = (Product) row[1]; //엔티티
    int orderAmount = (Integer) row[2]; //스칼라
}
```

물론 이때도 조회한 엔티티는 영속성 컨텍스트에서 관리된다.

- NEW 명령어

다음은 `username` , `age` 두 필드를 프로젝션해서 타입을 지정할 수 없으므로 `TypeQuery` 를 사용할 수 없다. 따라서 `Object[]` 을 반환받았다. 실제 애플리케이션 개발시에는 `Object[]` 을 직접 사용하지 않고 다음 코드의 `UserDTO` 처럼 의미있는 객체로 변환해서 사용할 것이다.

```
List<Object[]> resultList =
  em.createQuery("SELECT m.username, m.age FROM Member m")
    .getResultList();

//객체 변환 작업
List<UserDTO> userDTOs = new ArrayList<UserDTO>();
for (Object[] row : resultList) {
    UserDTO userDTO = new UserDTO((String)row[0], (Integer)row[1]);
    userDTOs.add(userDTO);
}
return userDTOs;
```

====== UserDTO ======

```
public class UserDTO {

    private String username;
    private int age;

    public UserDTO(String username, int age) {
        this.username = username;
        this.age = age;
    }
    //...
}
```

이런 객체 변환 작업은 지루하다. 이번에는 NEW 명령어를 사용해보자.

====== NEW 명령어를 사용 ======

```
TypedQuery<UserDTO> query =
  em.createQuery("SELECT new jpabook.jpql.UserDTO(m.username, m.age) FROM Member
                UserDTO.class);
```

```
List<UserDTO> resultList = query.getResultList();
```

`SELECT` 다음에 `NEW` 명령어를 사용하면 반환 받을 클래스 지정할 수 있는데 이 클래스의 생성자에 JPQL 조회 결과를 넘겨줄 수 있다. 그리고 `NEW` 명령어를 사용한 클래스로 `TypeQuery` 사용할 수 있어서 지루한 객체 변환 작업을 줄일 수 있다.

NEW 명령어를 사용할 때는 다음 2가지를 주의해야 한다.

1. 패키지 명을 포함한 전체 클래스 명을 입력해야 한다.
2. 순서와 타입이 일치하는 생성자가 필요하다.

페이징 API

페이징 처리용 SQL을 작성하는 일은 지루하고 반복적이다. 더 큰 문제는 데이터베이스마다 페이징을 처리하는 SQL 문법이 다르다는 점이다.

JPA는 페이징을 다음 두 API로 추상화했다.

- `setFirstResult(int startPosition)` : 조회 시작 위치 (0부터 시작한다.)
- `setMaxResults(int maxResult)` : 조회할 데이터 수

====== 페이징 사용 ======

```
TypedQuery<Member> query =
  em.createQuery("SELECT m FROM Member m ORDER BY m.username DESC", Member.class

query.setFirstResult(10);
query.setMaxResults(20);
query.getResultList();
```

코드를 분석하면 `FirstResult`의 시작은 0이므로 11번째부터 시작해서 총 20건의 데이터를 조회한다. 따라서 11 ~ 30번 데이터를 조회한다.

데이터베이스마다 다른 페이징 처리를 같은 API로 처리할 수 있는 것은 데이터베이스 방언(Dialect) 덕분이다. 이 JPQL이 방언에 따라 어떤 SQL로 변환되는지 확인해보자. 참고로 페이징 쿼리는 정렬조건이 중요하므로 예제에 포함했다.

====== HSQLDB (`org.hibernate.dialect.HSQLDialect`) ======

```
SELECT
    M.ID AS ID,
    M.AGE AS AGE,
    M.TEAM_ID AS TEAM_ID,
    M.NAME AS NAME
FROM
    MEMBER M
ORDER BY
    M.NAME DESC OFFSET ? LIMIT ?
```

====== MySQL (`org.hibernate.dialect.MySQL5InnoDBDialect`) =======

```
SELECT
    M.ID AS ID,
    M.AGE AS AGE,
    M.TEAM_ID AS TEAM_ID,
    M.NAME AS NAME
FROM
    MEMBER M
ORDER BY
    M.NAME DESC LIMIT ?, ?
```

====== PostgreSQL (`org.hibernate.dialect.PostgreSQL82Dialect`) ======

```
SELECT
    M.ID AS ID,
    M.AGE AS AGE,
    M.TEAM_ID AS TEAM_ID,
    M.NAME AS NAME
FROM
    MEMBER M
ORDER BY
    M.NAME DESC LIMIT ? OFFSET ?
```

====== Oracle (`org.hibernate.dialect.Oracle10gDialect`) =======

```
SELECT *
FROM
    ( SELECT ROW_.*, ROWNUM ROWNUM_
    FROM
        ( SELECT
            M.ID AS ID,
            M.AGE AS AGE,
```

```
            M.TEAM_ID AS TEAM_ID,
            M.NAME AS NAME
        FROM MEMBER M
        ORDER BY M.NAME
        ) ROW_
    WHERE ROWNUM <= ?
    )
WHERE ROWNUM_ > ?
```

====== SQLServer (`org.hibernate.dialect.SQLServer2008Dialect`) ======

```
WITH query AS (
    SELECT
    inner_query.*,
    ROW_NUMBER() OVER (ORDER BY CURRENT_TIMESTAMP) as __hibernate_row_nr__
    FROM
        ( select
            TOP(?) m.id as id,
            m.age as age,
            m.team_id as team_id,
            m.name as name
        from Member m
        order by m.name DESC
        ) inner_query
)
SELECT id, age, team_id, name
FROM query
WHERE __hibernate_row_nr__ >= ? AND __hibernate_row_nr__ < ?
```

데이터베이스마다 SQL이 다른 것은 물론이고 Oracle과 SQLServer는 페이징 쿼리를 따로 공부해야 SQL을 작성할 수 있을 정도로 복잡하다. 참고로 ? 에 바인딩하는 값도 데이터베이스마다 다른데 이 값도 적절한 값을 입력한다.

실행된 페이징 SQL을 보면 실무에서 작성한 것과 크게 다르지 않을 것이다. 페이징 SQL을 더 최적화하고 싶다면 JPA가 제공하는 페이징 API가 아닌 네이티브SQL을 직접 사용해야 한다.

집합과 정렬

집합은 집합함수와 함께 통계 정보를 구할 때 사용한다.

예를 들어 다음코드는 순서대로 회원수, 나이 합, 평균 나이, 최대 나이, 최소 나이를 조회한다.

```
select
    COUNT(m),   //회원수
    SUM(m.age), //나이 합
    AVG(m.age), //평균 나이
    MAX(m.age), //최대 나이
    MIN(m.age)  //최소 나이
from Member m
```

먼저 집합 함수부터 알아보자.

- 집합 함수

함수	설명
COUNT	결과 수를 구한다. 반환 타입: `Long`
MAX, MIN	최대, 최소 값을 구한다. 문자, 숫자, 날짜 등에 사용한다.
AVG	평균값을 구한다. 숫자타입만 사용할 수 있다. 반환 타입: `Double`
SUM	합을 구한다. 숫자타입만 사용할 수 있다. 반환 타입: 정수합 `Long`, 소수합: `Double`, `BigInteger` 합: `BigInteger`, `BigDecimal` 합: `BigDecimal`

집합 함수 사용시 참고사항

- `NULL` 값은 무시하므로 통계에 잡히지 않는다. (`DISTINCT`가 정의되어 있어도 무시된다.)
- 만약 값이 없는데 `SUM`, `AVG`, `MAX`, `MIN` 함수를 사용하면 `NULL` 값이 된다. 단 `COUNT`는 0이 된다.
- `DISTINCT`를 집함 함수 안에 사용해서 중복된 값을 제거하고 나서 집합을 구할 수 있다.
  - 예) `select COUNT( DISTINCT m.age ) from Member m`
- `DISTINCT`를 `COUNT`에서 사용할 때 임베디드 타입은 지원하지 않는다.

- GROUP BY, HAVING

`GROUP BY`는 통계데이터를 구할 때 특정 그룹끼리 묶어준다. 다음은 팀 이름을 기준으로 그룹별로 묶어서 통계 데이터를 구한다.

```
select t.name, COUNT(m.age), SUM(m.age),  AVG(m.age), MAX(m.age), MIN(m.age)
from Member m LEFT JOIN m.team t
GROUP BY t.name
```

`HAVING`은 `GROUP BY`와 함께 사용하는데 `GROUP BY`로 그룹화한 통계 데이터를 기준으로 필터링한다.

다음 코드는 방금 구한 그룹별 통계 데이터 중에서 평균나이가 10살 이상인 그룹을 조회한다.

```
select t.name, COUNT(m.age), SUM(m.age), AVG(m.age), MAX(m.age), MIN(m.age)
from Member m LEFT JOIN m.team t
GROUP BY t.name
HAVING AVG(m.age) >= 10
```

문법은 다음과 같다.

```
groupby_절 ::= GROUP BY {단일값 경로 | 별칭}+
having_절 ::= HAVING 조건식
```

이런 쿼리들을 보통 리포팅 쿼리나 통계 쿼리라 한다. 이러한 통계 쿼리를 잘 활용하면 애플리케이션으로 수십 라인을 작성할 코드도 단 몇 줄이면 처리할 수 있다. 하지만 통계 쿼리는 보통 전체 데이터를 기준으로 처리하므로 실시간으로 사용하기엔 부담이 많다. 결과가 아주 많다면 통계 결과만 저장하는 테이블을 별도로 만들어 두고 사용자가 적은 새벽에 통계 쿼리를 실행해서 그 결과를 보관하는 것이 좋다.

- 정렬 (ORDER BY)

`ORDER BY`는 결과를 정렬할 때 사용한다. 다음은 나이를 기준으로 내림차순으로 정렬하고 나이가 같으면 이름을 기준으로 오름차순으로 정렬한다.

```
select m from Member m order by m.age DESC, m.username ASC
```

문법은 다음과 같다.

```
orderby_절 ::= ORDER BY {상태필드 경로 | 결과 변수 [ASC | DESC]}+
```

- ASC: 오름차순 (기본값)
- DESC: 내림차순

문법에서 이야기하는 상태필드는 `t.name` 같이 객체의 상태를 나타내는 필드를 말한다. 그리고 결과 변수는 SELECT 절에 나타나는 값을 말한다. 다음 예에서 `cnt`가 결과 변수다.

```
select t.name, COUNT(m.age) as cnt
from Member m LEFT JOIN m.team t
GROUP BY t.name
ORDER BY cnt
```

JPQL 조인

JPQL도 조인을 지원하는데 SQL 조인과 기능은 같고 문법만 약간 다르다.

- 내부 조인

내부 조인은 `INNER JOIN`을 사용한다. 참고로 `INNER`는 생략할 수 있다.

====== 내부 조인 사용 예 ======

```
String teamName = "팀A";
String query = "SELECT m FROM Member m INNER JOIN m.team t "
             + "WHERE t.name = :teamName";

List<Member> members = em.createQuery(query, Member.class)
        .setParameter("teamName", teamName)
        .getResultList();
```

회원과 팀을 내부 조인해서 “팀A”에 소속된 회원을 조회하는 JPQL을 보자.

====== JPQL 내부 조인 ======

```
SELECT m
FROM Member m INNER JOIN m.team t    //**
where t.name = '팀A'
```

====== 생성된 내부 조인 SQL ======

```
SELECT
    M.ID AS ID,
    M.AGE AS AGE,
    M.TEAM_ID AS TEAM_ID,
    M.NAME AS NAME
FROM
    MEMBER M INNER JOIN TEAM T ON M.TEAM_ID=T.ID
WHERE
    T.NAME=?
```

JPQL 내부 조인 구문을 보면 SQL의 조인과 약간 다른 것을 확인할 수 있다. JPQL 조인의 가장 큰 특징은 **연관 필드**를 사용한다는 것이다. 여기서 `m.team`이 연관 필드인데 연관 필드는 다른 엔티티와 연관관계를 가지기 위해 사용하는 필드를 말한다.

- `FROM Member m` : 회원을 선택하고 `m` 이라는 별칭을 주었다.
- `Member m JOIN m.team t` : 회원이 가지고 있는 연관 필드로 팀과 조인한다. 조인한 팀에는 `t` 라는 별칭을 주었다.

혹시라도 JPQL 조인을 SQL 조인처럼 사용하면 문법 오류가 발생한다. JPQL은 `JOIN` 명령어 다음에 조인할 객체의 **연관 필드**를 사용한다. 다음은 잘못된 예이다.

```
FROM Member m JOIN Team t //잘못된 JPQL 조인, 오류!
```

조인 결과를 활용해보자.

```
SELECT m.username, t.name
FROM Member m JOIN m.team t
WHERE t.name  = '팀A'
ORDER BY m.age DESC
```

쿼리는 “팀A” 소속인 회원을 나이 내림차순으로 정렬하고 회원명과 팀명을 조회한다.

만약 조인한 두 개의 엔티티를 조회하려면 다음과 같이 JPQL을 작성하면 된다.

```
SELECT m, t
FROM Member m JOIN m.team t
```

서로 다른 타입의 두 엔티티를 조회했으므로 `TypeQuery`를 사용할 수 없다. 따라서 다음처럼 조회해야 한다.

```
List<Object[]> result = em.createQuery(query).getResultList();

for (Object[] row : result) {
    Member member = (Member) row[0];
    Team team = (Team) row[1];
}
```

- 외부 조인

```
SELECT m
FROM Member m LEFT [OUTER] JOIN m.team t
```

외부 조인은 기능상 SQL의 외부 조인과 같다. `OUTER`는 생략 가능해서 보통 `LEFT JOIN`으로 사용한다.

====== 생성된 외부 조인 SQL ======

```
SELECT
    M.ID AS ID,
    M.AGE AS AGE,
    M.TEAM_ID AS TEAM_ID,
    M.NAME AS NAME
FROM
    MEMBER M LEFT OUTER JOIN TEAM T ON M.TEAM_ID=T.ID
WHERE
    T.NAME=?
```

- 컬렉션 조인

일대다 관계나 다대다 관계처럼 컬렉션을 사용하는 곳에 조인하는 것을 컬렉션 조인이라 한다.

- [회원 -> 팀]으로의 조인은 다대일 조인이면서 **단일 값 연관 필드**(`m.team`)를 사용한다.
- [팀 -> 회원]은 반대로 일대다 조인이면서 **컬렉션 값 연관 필드(`m.members`)**를 사용한다.

다음 코드를 보자.

```
SELECT t, m FROM Team t LEFT JOIN t.members m
```

여기서 `t JOIN t.members` 는 팀과 팀이 보유한 회원목록을 **컬렉션 값 연관 필드로** 외부 조인했다.

> **참고:** 컬렉션 조인시에 `JOIN` 대신에 `IN` 을 사용할 수 있는데 기능상 `JOIN` 과 같지만 컬렉션일 때만 사용할 수 있다. 과거 EJB 시절의 유물이고 특별한 장점도 없으므로 그냥 `JOIN` 명령어를 사용하자.
>
> ```
> SELECT t, m FROM Team t, IN(t.members) m
> ```

- 세타 조인

WHERE 절을 사용해서 세타 조인을 할 수 있다. 참고로 세타 조인은 내부 조인만 지원한다. 세타 조인을 사용하면 다음 예처럼 전혀 관계 없는 엔티티도 조인할 수 있다.

예) 회원 이름이 팀 이름과 똑같은 사람 수를 구해라.

```
//JPQL
select count(m) from Member m, Team t
where m.username = t.name

//SQL
SELECT COUNT(M.ID)
FROM
    MEMBER M CROSS JOIN TEAM T
WHERE
    M.USERNAME=T.NAME
```

- JOIN ON 절 (JPA 2.1)

JPA 2.1부터 조인할 때 ON 절을 지원한다. ON 절을 사용하면 조인 대상을 필터링하고 조인할 수 있다. 참고로 내부 조인의 ON 절은 WHERE 절을 사용할 때와 결과가 같으므로 보통 `ON` 절은 외부 조인에서만 사용한다.

예를 들어 모든 회원을 조회하면서 회원과 연관된 팀도 조회하자. 이때 팀은 이름이 A인 팀만 조회하자.

====== JOIN ON 사용 예 ======

```
//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'
```

SQL 결과를 보면 `and t.name='A'` 로 조인 시점에 조인 대상을 필터링한다.

페치 조인

페치(fetch) 조인은 SQL에서 이야기하는 조인의 종류는 아니고 JPQL에서 성능 최적화를 위해 제공하는 기능이다. 이것은 연관된 엔티티나 컬렉션을 한 번에 같이 조회하는 기능인데 `join fetch` 명령어로 사용할 수 있다.

JPA 표준 명세에 정의된 페치 조인 문법은 다음과 같다.

```
페치 조인 ::= [ LEFT [OUTER] | INNER ] JOIN FETCH 조인경로
```

페치 조인에 대해 자세히 알아보자.

- 엔티티 페치 조인

페치 조인을 사용해서 회원 엔티티를 조회하면서 연관된 팀 엔티티도 함께 조회해보자.

====== 엔티티 페치 조인 JPQL ======

```
select m
from Member m join fetch m.team
```

예제를 보면 `join` 다음에 `fetch` 라 적었다. 이렇게 하면 연관된 엔티티나 컬렉션을 함께 조회하는데 여기서는 회원(`m`)과 팀(`m.team`)을 함께 조회한다. 참고로 일반적인 JPQL 조인과는 다르게 `m.team` 다음에 별칭이 없는데 페치 조인은 별칭을 사용할 수 없다.

> **참고:** 하이버네이트는 페치 조인에도 별칭을 허용한다.

====== 실행된 SQL======

```
SELECT
    M.*, T.*
FROM MEMBER T
INNER JOIN TEAM T ON M.TEAM_ID=T.ID
```

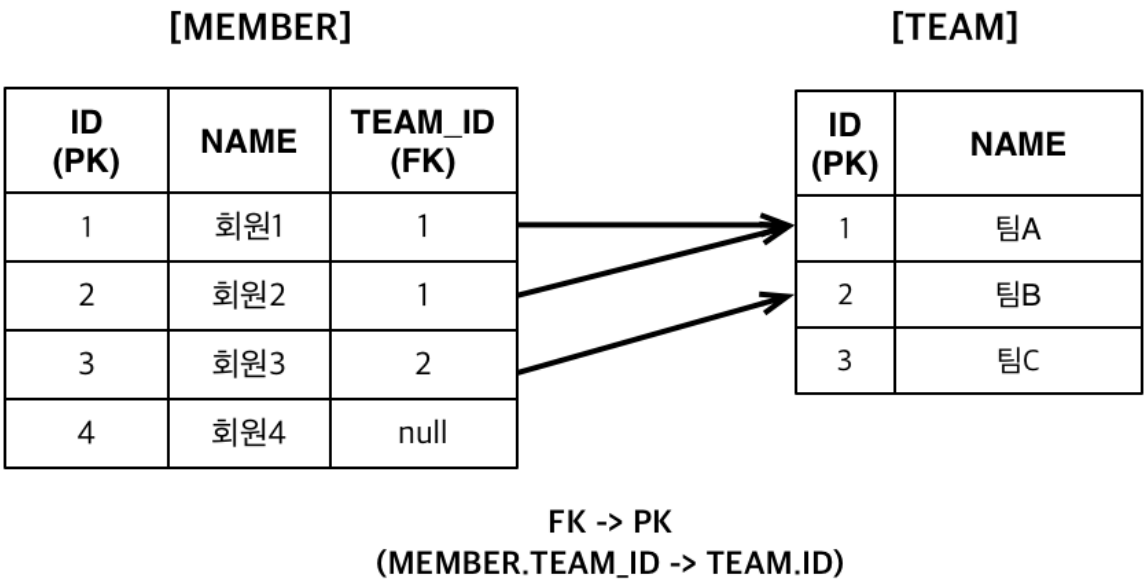

그림 9 I 엔티티 페치 조인 시도

[MEMBER JOIN TEAM]

ID (PK)	NAME	TEAM_ID (FK)	ID (PK)	NAME
1	회원1	1	1	팀A
2	회원2	1	1	팀A
3	회원3	2	2	팀B

그림 9 I 엔티티 페치 조인 결과 테이블

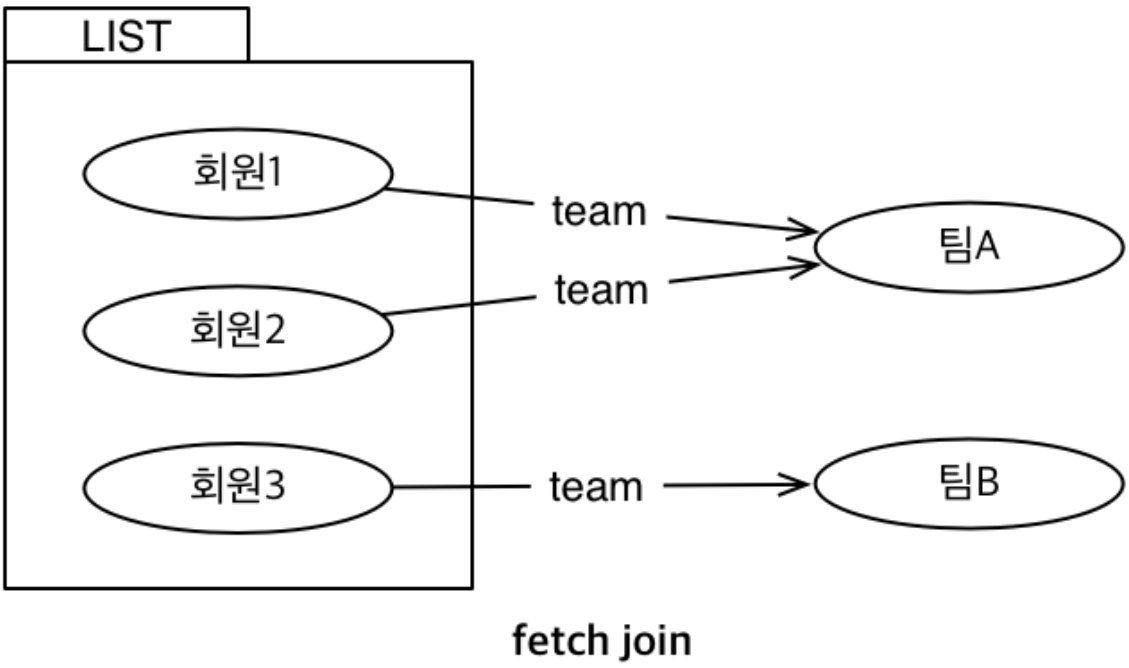

그림 10 I 엔티티 페치 조인 결과 객체

엔티티 페치 조인 JPQL에서 `select m`으로 회원 엔티티만 선택했는데 실행된 SQL을 보면 `SELECT M.*, T.*`로 회원과 연관된 팀도 함께 조회된 것을 확인할 수 있다.

====== 페치 조인 사용 ======

```
List<Member> members = em.createQuery(jpql, Member.class)
            .getResultList();

for (Member member : members) {
    //페치 조인으로 회원과 팀을 함께 조회해서 지연 로딩 발생 안함
    System.out.println("username = " + member.getUsername() + ", " +
                       "teamname = " + member.getTeam().name());
}
```

====== 출력 결과 ======

```
username = 회원1, teamname = 팀A
username = 회원2, teamname = 팀A
username = 회원3, teamname = 팀B
```

회원과 팀을 지연 로딩으로 설정했다고 가정해보자. 회원을 조회할 때 페치 조인을 사용해서 팀도 함께 조회했으므로 연관된 팀 엔티티는 프록시가 아닌 실제 엔티티다. 따라서 연관된 팀을 사용해도 지연 로딩이 일어나지 않는다. 그리고 프록시가 아닌 실제 엔티티이므로 회원 엔티티가 영속성 컨텍스트에서 분리되어 준영속 상태가 되어도 연관된 팀을 조회할 수 있다.

- 컬렉션 페치 조인

이번에는 일대다 관계인 컬렉션을 페치 조인해보자.

====== 컬렉션 페치 조인 JPQL ======

```
select t
from Team t join fetch t.members
where t.name = '팀A'
```

여기서는 팀(`t`)을 조회하면서 페치 조인을 사용해서 연관된 회원 컬렉션(`t.members`)도 함께 조회한다.

====== 실행된 SQL======

```
SELECT
    T.*, M.*
FROM TEAM T
INNER JOIN MEMBER M ON T.ID=M.TEAM_ID
WHERE T.NAME = '팀A'
```

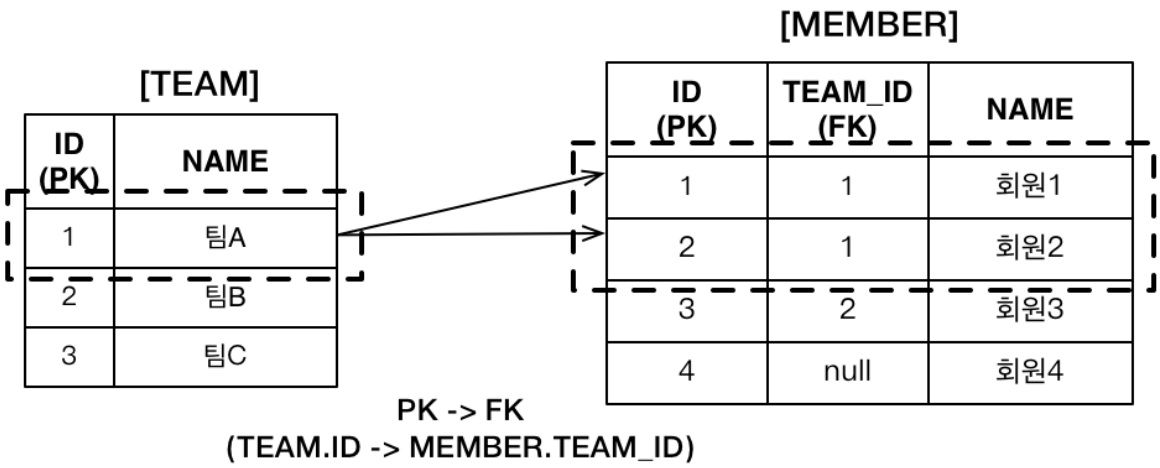

그림 9 I 컬렉션 페치 조인 시도

[TEAM JOIN MEBER]

ID (PK)	NAME	ID (PK)	TEAM_ID (FK)	NAME
1	팀A	1	1	회원1
1	팀A	2	1	회원2

그림 9 I 컬렉션 페치 조인 결과 테이블

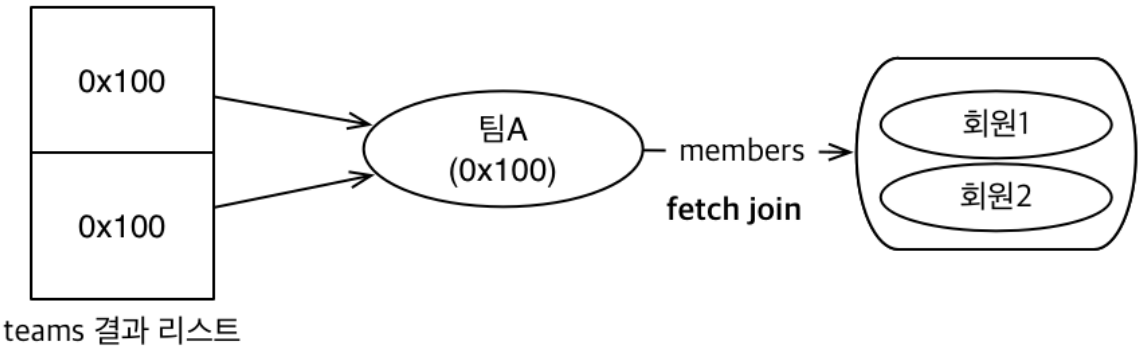

그림 10 I 컬렉션 페치 조인 결과 객체

컬렉션 페치 조인 JPQL에서 `select t`로 팀만 선택했는데 실행된 SQL을 보면 `T.*, M.*`로 팀과 연관된 회원도 함께 조회한 것을 확인할 수 있다. 그리고 `TEAM` 테이블에서 "팀A"는 하나지만 `MEMBER` 테이블과 조인하면서 결과가 증가해서 [그림 9 I 컬렉션 페치 조인 결과 테이블]을 보면 같은 "팀A"가 2건 조회되었다. 따라서 [그림 10 I 컬렉션 페치 조인 결과 객체]에서 `teams` 결과 리스트를 보면 주소가 `0x100`인 같은 "팀A"를 2건 가지게 된다.

컬렉션 페치 조인을 사용하는 코드를 보자.

> **참고:** 일대다 조인은 결과가 증가할 수 있지만 일대일, 다대일 조인은 결과가 증가하지 않는다.

====== 컬렉션 페치 조인 사용 ======

```
String jpql = "select t from Team t join fetch t.members where t.name = '팀A'"
List<Team> teams = em.createQuery(jpql, Team.class).getResultList();

for(Team team : teams) {
```

```
    System.out.println("teamname = " + team.getName() + ", team = " + team);

    for (Member member : team.getMembers()) {

      //페치 조인으로 팀과 회원을 함께 조회해서 지연 로딩 발생 안함
      System.out.println(
        "->username = " + member.getUsername()+ ", member = " + member);
    }
  }
```

====== 출력 결과 ======

```
teamname = 팀A, team = Team@0x100
->username = 회원1, member = Member@0x200
->username = 회원2, member = Member@0x300
teamname = 팀A, team = Team@0x100
->username = 회원1, member = Member@0x200
->username = 회원2, member = Member@0x300
```

출력 결과를 보면 같은 “팀A”가 2건 조회된 것을 확인할 수 있다.

- 페치 조인과 DISTINCT

SQL의 `DISTINCT` 는 중복된 결과를 제거하는 명령어다. JPQL의 `DISTINCT` 명령어는 SQL에 `DISTINCT` 를 추가하는 것은 물론이고 애플리케이션에서 한 번 더 중복을 제거한다.

바로 직전에 컬렉션 페치 조인은 팀A가 중복으로 조회된다. 다음처럼 `DISTINCT` 를 추가해보자.

```
select distinct t  //**
from Team t join fetch t.members
where t.name = '팀A'
```

먼저 `DISTINCT` 를 사용하면 SQL에 `SELECT DISTINCT` 가 추가된다. 하지만 지금은 각 로우의 데이터가 다르므로 SQL의 `DISTINCT` 는 효과가 없다.

====== 데이터가 달라 SQL의 DISTINCT 효과가 없음 ======

로우 번호	팀	회원
1	팀A	회원1
2	팀A	회원2

다음으로 애플리케이션에서 `distinct` 명령어를 보고 중복된 데이터를 걸러낸다. `select distinct t`의 의미는 팀 엔티티의 중복을 제거하라는 것이다. 따라서 중복인 팀A는 하나만 조회된다.

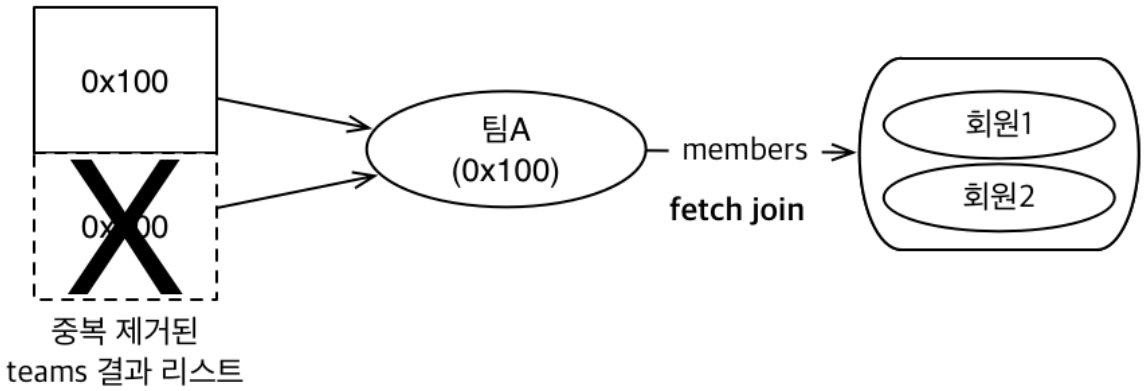

그림 9 I 페치 조인 DISTINCT 결과

컬렉션 페치 조인 사용 예제에 `distinct`를 추가하면 출력 결과는 다음과 같다.

====== `distinct` 추가 출력 결과 ======

```
teamname = 팀A, team = Team@0x100
->username = 회원1, member = Member@0x200
->username = 회원2, member = Member@0x300
```

- 페치 조인과 일반 조인의 차이

페치 조인을 사용하지 않고 조인만 사용하면 어떻게 될까?

====== 내부 조인 JPQL ======

```
select t
from Team t join t.members m
where t.name = '팀A'
```

====== 실행된 SQL ======

```
SELECT
    T.*
FROM TEAM T
INNER JOIN MEMBER M ON T.ID=M.TEAM_ID
WHERE T.NAME = '팀A'
```

JPQL에서 팀과 회원 컬렉션을 조인했으므로 회원 컬렉션도 함께 조회할 것으로 기대해선 안 된다. 실행된 SQL의 SELECT 절을 보면 팀만 조회하고 조인했던 회원은 전혀 조회하지 않는다.

JPQL은 결과를 반환할 때 연관관계까지 고려하지 않는다. 단지 SELECT 절에 지정한 엔티티만 조회할 뿐이다. 따라서 팀 엔티티만 조회하고 연관된 회원 컬렉션은 조회하지 않는다. 만약 회원 컬렉션을 지연 로딩으로 설정하면 다음 그림처럼 프록시나 컬렉션 래퍼를 반환하고, 즉시 로딩으로 설정하면 회원 컬렉션을 즉시 로딩하기 위해 쿼리를 한 번 더 실행한다.

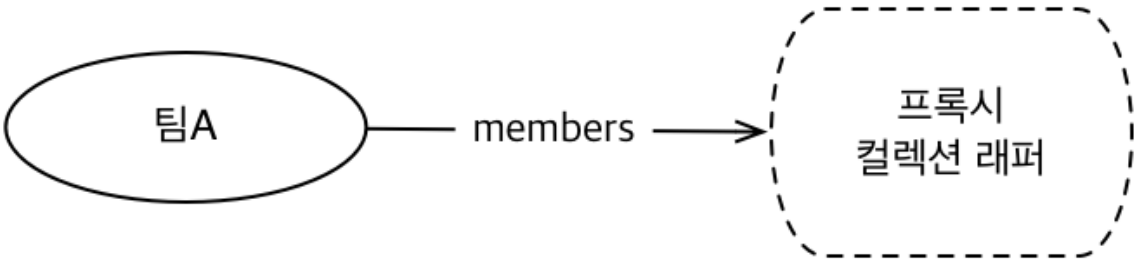

그림 9 I 페치 조인을 사용하지 않음, 조회 직후

반면에 페치 조인을 사용하면 연관된 엔티티도 함께 조회한다.

====== 컬렉션 페치 조인 JPQL ======

```
select t
from Team t join fetch t.members
where t.name = '팀A'
```

======실행된 SQL======

```
SELECT
    T.*, M.*
FROM TEAM T
INNER JOIN MEMBER M ON T.ID=M.TEAM_ID
WHERE T.NAME = '팀A'
```

실행된 SQL을 보면 `SELECT T.*, M.*` 로 팀과 회원을 함께 조회한 것을 알 수 있다.

- 페치 조인의 특징과 한계

페치 조인을 사용하면 SQL 한 번으로 연관된 엔티티들을 함께 조회할 수 있어서 SQL 호출 횟수를 줄여 성능을 최적화할 수 있다.

다음처럼 엔티티에 직접 적용하는 로딩 전략은 애플리케이션 전체에 영향을 미치므로 글로벌 로딩 전략이라 부른다. 페치 조인은 글로벌 로딩 전략보다 우선한다. 예를 들어 글로벌 로딩 전략을 지연 로딩으로 설정해도 JPQL에서 페치 조인을 사용하면 페치 조인을 적용해서 함께 조회한다.

```
@OneToMany(fetch = FetchType.LAZY) //글로벌 로딩 전략
```

최적화를 위해 글로벌 로딩 전략을 즉시 로딩으로 설정하면 애플리케이션 전체에서 항상 즉시 로딩이 일어난다. 물론 일부는 빠를 수는 있지만 전체로 보면 사용하지 않는 엔티티를 자주 로딩하므로 오히려 성능에 악영향을 미칠 수 있다. 따라서 글로벌 로딩 전략은 될 수 있으면 지연 로딩을 사용하고 최적화가 필요하면 페치 조인을 적용하는 것이 효과적이다.

또한 페치 조인을 사용하면 연관된 엔티티를 쿼리 시점에 조회하므로 지연 로딩이 발생하지 않는다. 따라서 **준영속 상태에서도 객체 그래프를 탐색할 수 있다.**

페치 조인은 다음과 같은 한계가 있다.

- **페치 조인 대상에는 별칭을 줄 수 없다.**
  - 문법을 자세히 보면 페치 조인에 별칭을 정의하는 내용이 없다. 따라서 SELECT, WHERE 절, 서브 쿼리에 페치 조인 대상을 사용할 수 없다.
  - JPA 표준에서는 지원하지 않지만 하이버네이트를 포함한 몇몇 구현체들은 페치 조인에 별칭을 지원한다. 하지만 별칭을 잘못 사용하면 연관된 데이터 수가 달라져서 데이터 무결성이 깨질 수 있으므로 조심해서 사용해야 한다. 특히 2차 캐시와 함께 사용할 때 조심해야 하는데 연관된 데이터 수가 달라진 상태에서 2차 캐시에 저장되면 다른 곳에서 조회할 때도 연관된 데이터 수가 달라지는 문제가 발생할 수 있다.

- **둘 이상의 컬렉션을 페치할 수 없다.** 구현체에 따라 되기도 하는데 컬렉션 * 컬렉션의 카테시안 곱이 만들어지므로 주의해야 한다. 하이버네이트를 사용하면 “javax.persistence.PersistenceException: org.hibernate.loader.MultipleBagFetchException: cannot simultaneously fetch multiple

bags” 예외가 발생한다.

- **컬렉션을 페치 조인하면 페이징API(`setFirstResult` , `setMaxResults`)를 사용할 수 없다.**
  - 컬렉션(일대다)이 아닌 단일 값 연관 필드(일대일, 다대일)들은 페치 조인을 사용해도 페이징 API를 사용할 수 있다.
  - 하이버네이트에서 컬렉션을 페치 조인하고 페이징API를 사용하면 경고 로그를 남기면서 메모리에서 페이징 처리를 한다. 데이터가 적으면 상관없겠지만 데이터가 많으면 성능 이슈와 메모리 초과 예외가 발생할 수 있어서 위험하다.

경로 표현식(Path Expression)

지금까지 JPQL 조인을 알아보았다. 이번에는 JPQL에서 사용하는 경로 표현식을 알아보고 경로 표현식을 통한 묵시적 조인도 알아보자.

경로 표현식이라는 것은 쉽게 이야기해서 `.`(점)을 찍어 객체 그래프를 탐색하는 것이다.

```
select m.username
from Member m
    join m.team t
    join m.orders o
where t.name = '팀A'
```

여기서 `m.username` , `m.team` , `m.orders` , `t.name` 이 모두 경로 표현식을 사용한 예다.

- 경로 표현식의 용어정리

경로 표현식을 이해하려면 우선 다음 용어들을 알아야 한다.

- 상태 필드(state field): 단순히 값을 저장하기 위한 필드 (필드 or 프로퍼티)
- 연관 필드(association field): 연관관계를 위한 필드, 임베디드 타입 포함 (필드 or 프로퍼티)
  - **단일 값 연관 필드:** `@ManyToOne` , `@OneToOne` , 대상이 엔티티
  - **컬렉션 값 연관 필드:** `@OneToMany` , `@ManyToMany` , 대상이 컬렉션

상태 필드는 단순히 값을 저장하는 필드고 연관 필드는 객체 사이의 연관관계를 맺기 위해 사용하는 필드다.

예제 코드로 상태 필드와 연관 필드를 알아보자.

==== 상태 필드, 연관 필드 설명 예제 코드 ====

```
@Entity
public class Member {

    @Id @GeneratedValue
    private Long id;

    @Column(name = "name")
    private String username; //상태 필드
    private Integer age; //상태 필드

    @ManyToOne(..)
    private Team team; //연관 필드(단일 값 연관 필드)

    @OneToMany(..)
    private List<Order> orders; //연관 필드(컬렉션 값 연관 필드)
```

정리하면 다음 3가지 경로 표현식이 있다.

- 상태 필드 : 예) `t.username` , `t.name`
- 단일 값 연관 필드 : 예) `m.team`
- 컬렉션 값 연관 필드 : 예) `m.orders`

- 경로 탐색

JPQL에서 경로 표현식을 사용해서 경로 탐색을 하려면 다음 3가지 경로에 따라 어떤 특징이 있는지 이해해야 한다.

- 상태 필드 경로 : 경로 탐색의 끝이다. 더는 탐색할 수 없다.
- 단일 값 연관 경로 : **묵시적으로 내부 조인**이 일어난다. 단일 값 연관 경로는 계속 탐색할 수 있다.
- 컬렉션 값 연관 경로 : **묵시적으로 내부 조인**이 일어난다. 더는 탐색할 수 없다. 단 FROM 절에서 조인을 통해 별칭을 얻으면 별칭으로 탐색할 수 있다.

예제를 통해 경로 탐색을 하나하나 알아보자.

– 상태 필드 경로 탐색

다음 코드의 `m.username` , `m.age` 는 상태 필드 경로 탐색이다.

====== JPQL ======

```
select m.username, m.age from Member m
```

====== 실행된 SQL ======

```
select m.name, m.age
from Member m
```

상태 필드 경로 탐색은 이해하는데 어려움이 없을 것이다.

– 단일 값 연관 경로 탐색

====== JPQL ======

```
select o.member from Order o
```

====== 실행된 SQL ======

```
select m.*
from Orders o
    inner join Member m on o.member_id=m.id
```

JPQL을 보면 `o.member`를 통해 주문에서 회원으로 단일 값 연관 필드로 경로 탐색을 했다. **단일 값 연관 필드로 경로 탐색을 하면 SQL에서 내부 조인이 일어나는데 이것을 묵시적 조인이라 한다.** 참고로 **묵시적 조인은 모두 내부 조인이다.** 외부 조인은 명시적으로 JOIN 키워드를 사용해야 한다.

명시적 조인과 묵시적 조인

- **명시적 조인:** JOIN을 직접 적어주는 것
  - ex) `SELECT m FROM Member m JOIN m.team t`
- **묵시적 조인:** 경로 표현식에 의해 묵시적으로 조인이 일어나는 것, 내부 조인(INNER JOIN)만 할 수 있다.
  - ex) `SELECT m.team FROM Member m`

이번에는 복잡한 예제를 보자. 처음에 나오는 샘플 모델 UML 그림을 보면서 다음 JPQL을 분석해보자.

====== JPQL ======

```
select o.member.team
from Order o
where o.product.name = 'productA' and o.address.city = 'JINJU'
```

주문 중에서 상품명이 “productA”고 배송지가 “JINJU”인 회원이 소속된 팀을 조회한다. 실제 내부 조인이 몇 번 일어날지 생각해 보자.

====== 실행된 SQL ======

```
select t.*
from Orders o
inner join Member m on o.member_id=m.id
inner join Team t on m.team_id=t.id
inner join Product p on o.product_id=p.id
where p.name='productA' and o.city='JINJU'
```

실행된 SQL을 보면 총 3번의 조인이 발생했다. 참고로 `o.address`처럼 임베디드 타입에 접근하는 것도 단일 값 연관 경로 탐색이지만 주문 테이블에 이미 포함되어 있으므로 조인이 발생하지 않는다.

– 컬렉션 값 연관 경로 탐색

JPQL을 다루면서 많이 하는 실수 중 하나는 컬렉션 값에서 경로 탐색을 시도하는 것이다.

```
select t.members from Team t // 성공
select t.members.username from Team t // 실패
```

`t.members`처럼 컬렉션까지는 경로 탐색이 가능하다. 하지만 `t.members.username`처럼 컬렉션에서 경로 탐색을 시작하는 것은 허락하지 않는다. 만약 컬렉션에서 경로 탐색을 하고 싶으면 다음 코드처럼 조인을 사용해서 새로운 별칭을 획득해야 한다.

```
select m.username from Team t join t.members m
```

`join t.members m`으로 컬렉션에 새로운 별칭을 얻었다. 이제 별칭 `m`부터 다시 경로 탐색을 할 수 있다.

참고로 컬렉션은 컬렉션의 크기를 구할 수 있는 `size` 라는 특별한 기능을 사용할 수 있다. `size` 를 사용하면 `COUNT` 함수를 사용하는 SQL로 적절히 변환된다.

```
select t.members.size from Team t
```

– 경로 탐색을 사용한 묵시적 조인시 주의사항

경로 탐색을 사용하면 묵시적 조인이 발생해서 SQL에서 내부 조인이 일어날 수 있다. 이때 주의사항은 다음과 같다.

- 항상 내부 조인이다.
- 컬렉션은 경로 탐색의 끝이다. 컬렉션에서 경로 탐색을 하려면 명시적으로 조인해서 별칭을 얻어야 한다.
- 경로 탐색은 주로 SELECT, WHERE 절(다른 곳에서도 사용됨)에서 사용하지만, 묵시적 조인으로 인해 SQL의 FROM 절에 영향을 준다.

조인이 성능상 차지하는 부분은 아주 크다. 묵시적 조인은 조인이 일어나는 상황을 한눈에 파악하기 어렵다는 단점이 있다. 따라서 단순하고 성능에 이슈가 없으면 크게 문제가 안 되지만 성능이 중요하면 분석하기 쉽도록 묵시적 조인보다는 명시적 조인을 사용하자.

서브 쿼리

JPQL도 SQL처럼 서브 쿼리를 지원한다. 여기에는 몇 가지 제약이 있는데, 서브쿼리를 WHERE, HAVING 절에서만 사용할 수 있고 SELECT, FROM 절에서는 사용할 수 없다.

> **참고:** HQL은 SELECT 절의 서브 쿼리도 허용한다. 하지만 아직까지 FROM 절의 서브 쿼리는 지원하지 않는다. 일부 JPA 구현체는 FROM 절의 서브 쿼리도 지원한다.

서브 쿼리 사용 예

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
```

참고로 이 쿼리는 다음처럼 컬렉션 값 연관 필드의 `size` 기능을 사용해도 같은 결과를 얻을 수 있다. (실행되는 SQL도 같다.)

```
select m from Member m
where m.orders.size > 0
```

- 서브 쿼리 함수

서브쿼리는 다음 함수들과 같이 사용할 수 있다.

- [NOT] EXISTS (subquery)
- {ALL | ANY | SOME} (subquery)
- [NOT] IN (subquery)

– EXISTS

- 문법 : [NOT] EXISTS (subquery)
- 설명 : 서브쿼리에 결과가 존재하면 참이다. NOT은 반대

예) : 팀A 소속인 회원

```
select m from Member m
where exists (select t from m.team t where t.name = '팀A')
```

– {ALL | ANY | SOME}

- 문법 : {ALL | ANY | SOME} (subquery)
- 설명 : 비교 연산자와 같이 사용한다. `{= | > | >= | < | <= | <>}`
  - ALL: 조건을 모두 만족하면 참이다.
  - ANY or SOME: 둘은 같은 의미다. 조건을 하나라도 만족하면 참이다.

예) : 전체 상품 각각의 재고보다 주문량이 많은 주문들

```
select o from Order o
where o.orderAmount > ALL (select p.stockAmount from Product p)
```

예) : 어떤 팀이든 팀에 소속된 회원

```
select m from Member m
where m.team = ANY (select t from Team t)
```

– IN

- 문법 : [NOT] IN (subquery)
- 설명 : 서브쿼리의 결과중 하나라도 같은 것이 있으면 참이다. 참고로 IN은 서브쿼리가 아닌 곳에서도 사용한다.

예) : 20세 이상을 보유한 팀

```
select t from Team t
where t IN (select t2 From Team t2 JOIN t2.members m2 where m2.age >= 20)
```

조건식(Conditional Expression)

- 타입 표현

JPQL에서 사용하는 타입은 다음과 같이 표시한다. 대소문자는 구분하지 않는다.

종류	설명	예제
문자	작은 따옴표 사이에 표현 작은 따옴표를 표현하고 싶으면 작은 따옴표 연속 두 개(`''`)사용	`'HELLO'` `'She''s '`
숫자	L (Long 타입 지정) D (Double 타입 지정) F (Float 타입 지정)	10L 10D 10F

날짜	DATE {d ‘yyyy-mm-dd’}
	TIME {t ‘hh-mm-ss’}
	DATETIME {ts ‘yyyy-mm-dd hh:mm:ss.f’}
		`{d '2012-03-24'}`
		`{t '10-11-11'}`
		`{ts '2012-03-24 10-11-11.123'}`
		`m.createDate = {d '2012-03-24'}`

Boolean	TRUE, FALSE	
Enum	패키지명을 포함한 전체 이름을 사용해야 한다.	`jpabook.MemberType.Admin`
엔티티 타입	엔티티의 타입을 표현한다. 주로 상속과 관련해서 사용한다.	`TYPE(m) = Member`

- 논리 연산(조건식 합성)

- `AND` : 둘다 만족하면 참
- `OR` : 둘 중 하나만 만족해도 참
- `NOT` : 조건식의 결과 반대

- 비교식

비교식은 다음과 같다.

`= | > | >= | < | <= | <>`

- 연산자 우선 순위

연산자 우선 순위는 다음과 같다.

1. 경로 탐색 연산 (.)
2. 수학 연산:
   +, -(단항 연산자), *, /, +, -
3. 비교 연산 : =, >, >=, <, <=, <>(다름), [NOT] BETWEEN, [NOT] LIKE, [NOT]IN,IS [NOT] NULL, IS [NOT] EMPTY,[NOT] MEMBER [OF],[NOT] EXISTS
4. 논리 연산: NOT, AND, OR

- Between 식

문법: X [NOT] BETWEEN A AND B
설명: X 는 A ~ B 사이의 값이면 참 (A,B 값 포함)

예 : 나이가 10 ~ 20 인 회원을 찾아라.

```
select m from Member m
where m.age between 10 and 20
```

- IN 식

문법: X [NOT] IN (리스트)
설명: X 와 같은 값이 리스트에 하나라도 있으면 참이다. IN 식의 리스트에는 서브쿼리를 사용할 수 있다.

예: 이름이 회원1이나 회원2인 회원을 찾아라.

```
select m from Member m
where m.username in ('회원1','회원2')
```

- Like 식

문법: 문자표현식 [NOT] LIKE 패턴값 [ESCAPE 이스케이프문자]
설명: 문자표현식과 패턴값을 비교한다.

- %(퍼센트): 아무 값들이 입력되어도 된다.(값이 없어도 됨)
- _(언더라인): 한 글자는 아무 값이 입력되어도 되지만 값이 있어야 한다.

예:

```
//중간에 원이라는 단어가 들어간 회원 (좋은회원, 회원, 원)
select m from Member m
where m.username like '%원%'

//처음에 회원이라는 단어가 포함 (회원1, 회원ABC)
where m.username like '회원%'

//마지막에 회원이라는 단어가 포함 (좋은 회원, A회원)
where m.username like '%회원'

//회원A, 회원1 (회원:X)
where m.username like '회원_'

//회원3
where m.username like '__3'

//회원%
where m.username like '회원\%' ESCAPE '\'
```

- NULL 비교식

문법: {단일값 경로 I 입력 파라미터 } IS [NOT] NULL
설명: `NULL` 인지 비교한다. NULL은 `=` 으로 비교하면 안되고 꼭 `IS NULL` 을 사용해야 한다.

```
where m.username is null
where null = null // 거짓
where 1=1 // 참
```

- 컬렉션 식

컬렉션 식은 컬렉션에만 사용하는 특별한 기능이다. 참고로 컬렉션은 컬렉션 식 이외에 다른 식은 사용할 수 없다.

빈 컬렉션 비교 식

문법: {컬렉션 값 연관 경로} IS [NOT] EMPTY
설명: 컬렉션에 값이 비었으면 참

```
//JPQL: 주문이 하나라도 있는 회원 조회
select m from Member m
```

```
where m.orders is not empty

//실행된 SQL
select m.* from Member m
where
    exists (
        select o.id
        from Orders o
        where m.id=o.member_id
    )

//주의! 컬렉션은 컬렉션 식만 사용할 수 있다. 다음의 is null처럼 컬렉션 식이 아닌 것은 사용할 수 없다
select m from Member m where m.orders is null (오류!)
```

컬렉션의 멤버 식

문법: {엔티티나 값} [NOT] MEMBER [OF] {컬렉션 값 연관 경로}
설명: 엔티티나 값이 컬렉션에 포함되어 있으면 참

```
select t from Team t where :memberParam member of t.members
```

- 스칼라 식

스칼라는 숫자, 문자, 날짜, case, 엔티티 타입(엔티티의 타입 정보) 같은 가장 기본적인 타입들을 말한다. 스칼라 타입에 사용하는 식을 알아보자.

– 수학 식

`+`, `-` 단항 연산자
`*`, `/`, `+`, `-` 사칙연산

– 문자함수

함수	설명	예제
CONCAT(문자1, 문자2, …)	문자를 합한다.	CONCAT('A','B') = AB
SUBSTRING(문자, 위치, [길이])	위치부터 시작해 길이만큼 문자를 구한다. 길이 값이 없으면 나머지 전체 길이를 뜻한다.	SUBSTRING('ABCDEF', 2, 3) = BCD
TRIM([[LEADING | TRAILING | BOTH] [트림 문자] FROM] 문자)	LEADING: 왼쪽만 TRAILING: 오른쪽만 BOTH:양쪽다 트림문자를 제거한다. 기본값은 BOTH. 트림 문자의 기본값은 공백(SPACE)다.	TRIM(' ABC ') = 'ABC'
LOWER(문자)	소문자로 변경	LOWER('ABC') = 'abc'
UPPER(문자)	대문자로 변경	UPPER('abc') = 'ABC'
LENGTH(문자)	문자 길이	LENGTH('ABC') = 3
LOCATE(찾을 문자, 원본 문자, [검색시작위치])	검색위치부터 문자를 검색한다. 1부터 시작, 못찾으면 0 반환	LOCATE('DE', 'ABCDEFG') = 4

> **참고:** HQL은 CONCAT 대신에 `||` 도 사용할 수 있다.

– 수학함수

함수	설명	예제
ABS(수학식)	절대값을 구한다.	ABS(–10) = 10
SQRT(수학식)	제곱근을 구한다.	SQRT(4) = 2.0
MOD(수학식, 나눌수)	나머지를 구한다.	MOD(4,3) = 1
SIZE(컬렉션 값 연관 경로식)	컬렉션의 크기를 구한다.	SIZE(t.members)
INDEX(별칭)	LIST 타입 컬렉션의 위치값을 구함, 단 컬렉션이 `@OrderColumn`을 사용하는 LIST 타입일 때만 사용할 수 있다.	`t.members m where INDEX(m) > 3`

– 날짜함수

날짜함수는 데이터베이스의 현재 시간을 조회한다.

- `CURRENT_DATE` : 현재 날짜
- `CURRENT_TIME` : 현재 시간
- `CURRENT_TIMESTAMP` : 현재 날짜 시간

예)

```
select CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP from Team t
//결과: 2013-08-19, 23:38:17, 2013-08-19 23:38:17.736
```

예) 종료 이벤트 조회

```
select e from Event e where e.endDate < CURRENT_DATE
```

하이버네이트는 날짜 타입에서 년, 월, 일, 시간, 분, 초 값을 구하는 기능을 지원한다.

`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`

예)

```
select year(CURRENT_TIMESTAMP), month(CURRENT_TIMESTAMP), day(CURRENT_TIMESTAMP)
from Member
```

데이터베이스들은 각자의 방식으로 더 많은 날짜 함수를 지원한다. 그리고 각각의 날짜 함수는 데이터베이스 방언에 등록되어 있다. 예를 들어 Oracle 방언을 사용하면 `to_date`, `to_char` 함수를 사용할 수 있다. 물론 다른 데이터베이스를 사용하면 동작하지 않는다.

- CASE 식(Case Expression)

특정 조건에 따라 분기할 때 CASE 식을 사용한다. CASE 식은 4가지 종류가 있다.

- 기본 CASE
- 심플 CASE
- COALESCE
- NULLIF

순서대로 하나씩 알아보자.

기본 CASE

문법:

```
CASE
    {WHEN <조건식> THEN <스칼라식>}+
    ELSE <스칼라식>
END
```

예):

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

심플 CASE

심플 CASE는 조건식을 사용할 수 없지만, 문법이 단순하다. 참고로 자바의 switch case 문과 비슷하다.

문법 :

```
CASE <조건대상>
    {WHEN <스칼라식1> THEN <스칼라식2>}+
    ELSE <스칼라식>
END
```

예):

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

> **참고:** 표준 명세의 문법 정의는 다음과 같다.
>
> ```
> 기본 CASE 식::=
> CASE when_절 {when_절}* ELSE 스칼라식 END
> when_절::= WHEN 조건식 THEN 스칼라식
>
> 심플 CASE 식::=
> CASE case_피연산자 심플_when_절 {심플_when_절}* ELSE 스칼라식 END
> case_피연산자::= 상태 필드 경로식 | 타입 구분자
> 심플_when_절::= WHEN 스칼라식 THEN 스칼라식
> ```

COALESCE

문법 : COALESCE(<스칼라식> {,<스칼라식>}+)
설명 : 스칼라식을 차례대로 조회해서 `null` 이 아니면 반환한다.

예): `m.username`이 `null`이면 '이름 없는 회원'을 반환해라

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

NULLIF

문법 : NULLIF(<스칼라식>, <스칼라식>)
설명 : 두 값이 같으면 `null`을 반환하고 다르면 첫번째 값을 반환한다. 집합 함수는 `null`을 포함하지 않으므로 보통 집함 함수와 함께 사용한다.

예): 사용자 이름이 '관리자'면 `null`을 반환하고 나머지는 본인의 이름을 반환해라.

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

다형성 쿼리

JPQL로 부모 엔티티를 조회하면 그 자식 엔티티도 함께 조회한다.

예를 들어 `Item`의 자식으로 `Book`, `Album`, `Movie`가 있다.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "DTYPE")
public abstract class Item {...}

@Entity
@DiscriminatorValue("B")
public class Book extends Item {
    ...
    private String author;
}

//Album, Movie 생략
```

다음과 같이 조회하면 `Item`의 자식도 함께 조회한다.

```
List resultList = em.createQuery("select i from Item i").getResultList();
```

단일 테이블 전략(`InheritanceType.SINGLE_TABLE`)을 사용할 때 실행되는 SQL은 다음과 같다.

```
//SQL
SELECT * FROM ITEM
```

조인 전략(`InheritanceType.JOINED`)을 사용할 때 실행되는 SQL은 다음과 같다.

```
//SQL
SELECT
    i.ITEM_ID, i.DTYPE, i.name, i.price, i.stockQuantity,
    b.author, b.isbn,
    a.artist, a.etc,
    m.actor, m.director
FROM
    Item i
left outer join
    Book b on i.ITEM_ID=b.ITEM_ID
left outer join
    Album a on i.ITEM_ID=a.ITEM_ID
left outer join
    Movie m on i.ITEM_ID=m.ITEM_ID
```

- TYPE

`TYPE` 은 엔티티의 상속 구조에서 조회 대상을 특정 자식 타입으로 한정할 때 주로 사용한다.

예): `Item` 중에 `Book` , `Movie` 를 조회해라.

```
//JPQL
select i from Item i
where type(i) IN (Book, Movie)

//SQL
SELECT i FROM Item i
WHERE i.DTYPE in ('B', 'M')
```

- TREAT (JPA 2.1)

`TREAT`는 JPA 2.1에 추가된 기능인데 자바의 타입 캐스팅과 비슷하다. 상속 구조에서 부모 타입을 특정 자식 타입으로 다룰 때 사용한다. JPA 표준은 FROM, WHERE 절에서 사용할 수 있지만, 하이버네이트는 SELECT 절에서도 TREAT를 사용할 수 있다.

예): 부모인 `Item`과 자식 `Book`이 있다.

```
//JPQL
select i from Item i where treat(i as Book).author = 'kim'

//SQL
select i.* from Item i
where
    i.DTYPE='B'
    and i.author='kim'
```

JPQL을 보면 `treat`를 사용해서 부모 타입인 `Item`을 자식 타입인 `Book`으로 다룬다. 따라서 `author` 필드에 접근할 수 있다.

사용자 정의 함수 호출 (JPA 2.1)

JPA 2.1부터 사용자 정의 함수를 지원한다.

문법

```
function_invocation::= FUNCTION(function_name {, function_arg}*)
```

예제

```
select function('group_concat', i.name) from Item i
```

하이버네이트 구현체를 사용하면 방언 클래스를 상속해서 사용할 데이터베이스 함수를 미리 등록해야 한다.

====== 방언 클래스 상속 ======

```
public class MyH2Dialect extends H2Dialect {
```

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

====== 상속한 방언 클래스 등록(`persistence.xml`) =====

```
<property name="hibernate.dialect" value="hello.MyH2Dialect" />
```

하이버네이트 구현체를 사용하면 다음과 같이 축약해서 사용할 수 있다.

```
select group_concat(i.name) from Item i
```

기타 정리

- `enum` 은 `=` 비교 연산만 지원한다.
- 임베디드 타입은 비교를 지원하지 않는다.

- EMPTY STRING

JPA 표준은 `''` 을 길이 0인 Empty String으로 정했지만 데이터베이스에 따라 `''` 를 NULL 로 사용하는 데이터베이스도 있으므로 확인하고 사용해야 한다.

- NULL 정의

- 조건을 만족하는 데이터가 하나도 없으면 `NULL` 이다.
- `NULL` 은 알 수 없는 값(unknown value)이다. `NULL` 과의 모든 수학적 계산 결과는 `NULL` 이 된다.
- `Null == Null` 은 알 수 없는 값이다.
- `Null is Null` 은 참이다.

JPA 표준 명세는 `Null` (U) 값과 `TRUE` (T), `FALSE` (F)의 논리 계산을 다음과 같이 정의했다.

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

NOT	
T	F
F	T
U	U

엔티티를 직접 사용하기

- 기본 키 값

객체 인스턴스는 참조 값으로 식별하고 테이블 로우는 기본 키 값으로 식별한다. 따라서 JPQL에서 엔티티 객체를 직접 사용하면 SQL에서는 해당 엔티티의 기본 키 값을 사용한다. 다음 예제를 보자.

====== JPQL 예제 ======

```
select count(m.id) from Member m // 엔티티의 아이디를 사용
select count(m) from Member m //엔티티를 직접 사용
```

두 번째의 `count(m)`을 보면 엔티티의 별칭을 직접 넘겨주었다. 이렇게 엔티티를 직접 사용하면 JPQL이 SQL로 변환될 때 해당 엔티티의 기본 키를 사용한다. 따라서 실제 실행된 SQL은 둘 다 같다.

====== 실행된 SQL ======

```
select count(m.id) as cnt
from Member m
```

JPQL의 `count(m)` 이 SQL에서 `count(m.id)` 로 변환된 것을 확인할 수 있다.

이번에는 엔티티를 파라미터로 직접 받아보자.

====== 엔티티를 파라미터로 ====

```
String qlString = "select m from Member m where m = :member";
List resultList = em.createQuery(qlString)
    .setParameter("member", member)
    .getResultList();
```

====== 실행된 SQL ======

```
select m.*
from Member m
where m.id=?
```

JPQL과 SQL을 비교해보면 JPQL에서 `where m = :member` 로 엔티티를 직접 사용하는 부분이 SQL에서 `where m.id=?` 로 기본 키 값을 사용하도록 변환된 것을 확인할 수 있다.

물론 다음과 같이 식별자 값을 직접 사용해도 결과는 같다.

```
String qlString = "select m from Member m where m.id = :memberId";
List resultList = em.createQuery(qlString)
    .setParameter("memberId", 4L)
    .getResultList();
```

- 외래 키 값

이번에는 외래 키를 사용하는 예를 보자. 다음 코드는 특정 팀에 소속된 회원을 찾는다.

```
Team team = em.find(Team.class, 1L);

String qlString = "select m from Member m where m.team = :team";
List resultList = em.createQuery(qlString)
    .setParameter("team", team)
    .getResultList();
```

기본 키 값이 `1L` 인 팀 엔티티를 파라미터로 사용하고 있다. `m.team` 은 현재 `team_id` 라는 외래 키와 매핑되어 있다. 따라서 다음과 같은 SQL이 실행된다.

====== 실행된 SQL ======

```
select m.*
from Member m
where m.team_id=?(팀 파라미터의 ID 값)
```

외래키 식별자 직접 사용

엔티티 대신 식별자 값을 직접 사용할 수 있다.

```
String qlString = "select m from Member m where m.team.id = :teamId";
List resultList = em.createQuery(qlString)
    .setParameter("teamId", 1L)
    .getResultList();
```

예제에서 `m.team.id` 를 보면 `Member` 와 `Team` 간에 묵시적 조인이 일어날 것 같지만 `MEMBER` 테이블이 `team_id` 외래키를 가지고 있으므로 묵시적 조인은 일어나지 않는다. 물론 `m.team.name` 을 호출하면 묵시적 조인이 일어난다.

따라서 `m.team` 을 사용하든 `m.team.id` 를 사용하든 생성되는 SQL은 같다.

Named 쿼리 - 정적 쿼리

JPQL 쿼리는 크게 동적 쿼리와 정적 쿼리로 나눌 수 있다.

동적 쿼리: `em.createQuery("select ..")` 처럼 JPQL을 문자로 완성해서 직접 넘기는 것을 동적 쿼리라 한다. 런타임에 특정 조건에 따라 JPQL을 동적으로 구성할 수 있다.

정적 쿼리: 미리 정의한 쿼리에 이름을 부여해서 필요할 때 사용할 수 있는데 이것을 Named 쿼리라 한다. Named 쿼리는 한번 정의하면 변경할 수 없는 정적인 쿼리다.

Named 쿼리는 애플리케이션 로딩 시점에 JPQL 문법을 체크하고 미리 파싱해 둔다. 따라서 오류를 빨리 확인할 수 있고, 사용하는 시점에는 파싱된 결과를 재사용하므로 성능상 이점도 있다. 그리고 Named 쿼리는 변하지 않는 정적 SQL이 생성되므로 데이터베이스의 조회 성능 최적화에도 도움이 된다.

Named 쿼리는 `@NamedQuery` 어노테이션을 사용해서 자바 코드에 작성하거나 또는 XML 문서에 작성할 수 있다.

- Named 쿼리를 어노테이션에 정의하기

Named 쿼리는 이름 그대로 쿼리에 이름을 부여해서 사용하는 방법이다. 먼저 `@NamedQuery` 어노테이션을 사용하는 예를 보자.

====== `@NamedQuery` 어노테이션으로 Named 쿼리 정의 ======

```
@Entity
@NamedQuery(
        name = "Member.findByUsername",
        query="select m from Member m where m.username = :username")
public class Member {
    ...
}
```

`@NamedQuery.name`에 쿼리 이름을 부여하고 `@NamedQuery.query`에 사용할 쿼리를 입력한다.

====== `@NamedQuery` 사용 ======

```
List<Member> resultList = em.createNamedQuery("Member.findByUsername", Member.cl
        .setParameter("username", "회원1")
        .getResultList();
```

Named 쿼리를 사용할 때는 `em.createNamedQuery()` 메서드에 Named 쿼리 이름을 입력하면 된다.

> **참고:** Named 쿼리 이름을 간단히 `findByUsername`이라 하지 않고 `Member.findByUsername`처럼 앞에 엔티티 이름을 주었는데 이것이 기능적으로 특별한 의미가 있는 것은 아니다. 하지만 **Named 쿼리는 영속성 유닛 단위로 관리**되므로 충돌을 방지하기 위해 엔티티 이름을 앞에 주었다. 그리고 엔티티 이름이 앞에 있으면 관리하기가 쉽다.

하나의 엔티티에 2개 이상의 Named 쿼리를 정의하려면 다음처럼 `@NamedQueries` 어노테이션을 사용하면 된다.

```
@Entity
@NamedQueries({
    @NamedQuery(
        name = "Member.findByUsername",
        query="select m from Member m where m.username = :username"),
    @NamedQuery(
        name = "Member.count",
        query="select count(m) from Member m")
})
public class Member {
```

`@NamedQuery` 어노테이션 분석

```
@Target({TYPE})
public @interface NamedQuery {

    String name(); //Named 쿼리 이름 (필수)
    String query(); //JPQL 정의 (필수)
    LockModeType lockMode() default NONE; //쿼리 실행시 락모드를 설정할 수 있다.
    QueryHint[] hints() default {}; //JPA 구현체에 쿼리 힌트를 줄 수 있다.
}
```

- `lockMode` : 쿼리 실행시 락을 건다. 자세한 내용은 고급 주제에서 다룬다.
- `hints` : 여기서 힌트는 SQL 힌트가 아니라 JPA 구현체에게 제공하는 힌트다. 보통 2차 캐시를 다룰 때 사용한다.

- Named 쿼리를 XML에 정의하기

JPA에서 어노테이션으로 작성할 수 있는 것은 XML로도 작성할 수 있다. 물론 어노테이션을 사용하는 것이 직관적이고 편리하다. 하지만 Named 쿼리를 작성할 때는 XML을 사용하는 것이 더 편리하다.

자바 언어로 멀티라인 문자를 다루는 것은 상당히 귀찮은 일이다. (어노테이션을 사용해도 마찬가지다.)

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

그루비처럼 멀티라인을 지원하는 언어도 있다.

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

자바에서 이런 불편함을 해결하려면 XML을 사용하는 것이 그나마 현실적인 대안이다.

====== META-INF/ormMember.xml, XML에 정의한 Named 쿼리 ======

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm" version="2.

    <named-query name="Member.findByUsername">
        <query><![CDATA[
            select m
            from Member m
            where m.username = :username
        ]]></query>
    </named-query>

    <named-query name="Member.count">
        <query>select count(m) from Member m</query>
    </named-query>

</entity-mappings>
```

> **참고:** XML에서 `&`, `<`, `>`는 XML 예약문자다. 대신에 `&`, `<`, `>`를 사용해야 한다. <![CDATA[ ]]>를 사용하면 그 사이에 문장을 그대로 출력하므로 예약문자도 사용할 수 있다.

그리고 정의한 `ormMember.xml`을 인식하도록 `persistence.xml`에 추가해야 한다.

====== META-INF/persistence.xml 에 추가 ======

```
<persistence-unit name="jpabook" >
    <mapping-file>META-INF/ormMember.xml</mapping-file>
    ...
```

> **참고:** `META-INF/orm.xml`은 JPA가 기본 매핑파일로 인식해서 별도의 설정을 하지 않아도 된다. 이름이나 위치가 다르면 설정을 추가해야 한다.

- 환경에 따른 설정

만약 XML과 어노테이션에 같은 설정이 있으면 **XML이 우선권**을 가진다. 예를 들어 같은 이름의 Named 쿼리가 있으면 XML에 정의한 것이 사용된다. 따라서 애플리케이션이 운영 환경에 따라 다른 쿼리를 실행해야 한다면 각 환경에 맞춘 XML을 준비해 두고 XML만 변경해서 배포하면 된다.