JPQL 사용법 및 문법 정리

0. JPQL

JPQL은 "Java Persistence Query Language"의 약자로, 자바 기반의 영속성(Persistence) 프레임워크인 JPA(Java Persistence API)에서 사용되는 쿼리 언어이다. JPQL은 SQL과 비슷한 구문을 가지고 있지만, 객체 지향 프로그래밍 언어의 특성을 반영하여 엔티티 객체를 대상으로 쿼리를 작성한다. 따라서 데이터베이스 테이블이 아닌 엔티티 객체에 대해 쿼리 실행이 가능하다.

JPQL을 사용하면 특정 데이터베이스 벤더에 독립적인 쿼리를 작성할 수 있다. 즉, 애플리케이션에서 다양한 데이터베이스 시스템을 사용하더라도 동일하게 실행할 수 있다.

JPQL 쿼리는 런타임에 SQL 쿼리로 변환되어 데이터베이스에 전달되고 결과를 받아온다.

1. JPQL 문법

가. 대소문자 구분

- 엔티티와 속성은 대소문자를 구분한다.
- JPQL 키워드는 대소문자를 구분하지 않는다.

나. 엔티티 이름

- FROM 절에 오는 대상은 테이블 이름이 아니라 엔티티 이름이다. (ex: @Entity(name="엔티티이름"))
엔티티명을 지정하지 않으면 클래스명을 기본값으로 사용한다.

다. 별칭은 필수
- FROM 절에 엔티티의 이름과 해당 엔티티의 별칭(Alias)를 필수로 지정해야 한다.

```
SELECT e
FROM Employee e
WHERE e.department = :department AND e.salary > :salary
ORDER BY e.salary DESC
```

1.1. 서브 쿼리와 서브 쿼리 함수

JPQL도 SQL처럼 서브 쿼리를 지원하고, EXISTS, ALL, ANY, SOME, IN과 같은 서브 쿼리 함수를 지원한다.
다만, 서브 쿼리는 WHERE, HAVING 절에서만 사용할 수 있고 SELECT, FROM 절에서는 사용할 수 없다.

- EXISTS: 서브쿼리가 결과에 존재하면 참
- ALL: 조건을 모두 만족하면 참

- ANY, SOME: 둘은 같은 의미로, 조건을 하나라도 만족하면 참
- IN: 서브쿼리의 결과 중 하나라도 같은 것이 있으면 참

```
// 서브쿼리: 나이가 평균보다 많은 회원 조회
나이가 평균보다 많은 회원 찾기
SELECT m
FROM Member m
WHERE m.age > (
    SELECT avg(m2.age)
    FROM Member m2
    )
```

```
// EXISTS: teamA에 소속인 회원 조회
SELECT m
FROM Member m
WHERE EXISTS (
    SELECT t
    FROM m.team t
    WHERE t.name = 'teamA'
    )
```

```
// ALL: 전체 상품 각각의 제고보다 주문량이 많은 주문들
SELECT o
FROM Order o
WHERE o.orderAmount > ALL (SELECT p.stockAmount FROM Product p)
```

```
// SOME: 어떤 팀이든 팀에 소속된 회원
SELECT m
FROM Member m
WHERE m.team = ANY (SELECT t FROM Team t)
```

1.2. 집합과 정렬

- JPQL도 SQL처럼 집합과 정렬을 지원하고, 그룹 함수도 동일하게 제공한다.
- 그룹핑 키워드(GROUP BY, HAVING), 그룹 함수(COUNT, MAX, MIN, AVG, SUM), 정렬(ORDER BY) 모두 지원

- 집합 함수 사용시 NULL 값은 제외되고 통계를 내어 값이 계산된다. COUNT는 예외적으로 NULL 값도 포함되며, 값이 없어도 0을 반환한다.

- SQL 쿼리 순서는 FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY 순서로 진행된다.

```
// 팀 이름별로 그룹화하고, 나이의 평균이 10살 이상인 그룹에 대해서 통계값을 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 AVG(m.age) >= 10
```

1.3. 기본 연산자

- JPQL도 SQL과 동일하게 EXISTS, IN, AND, OR, NOT, =, >, >=, <, <=, <>, BETWEEN, LIKE, IS NULL 등의 문법을 지원한다.

- IS [NOT] EMPTY, [NOT] MEMBER [OF] 만 빼면 사용법은 일반적인 SQL과 동일하다. 이 두개는 컬렉션 식으로써 JPA에서 제공하는, 컬렉션에만 사용가능한 특별 기능이다.

- 연산자 우선 순위는 아래와 같다.

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

1.4. 컬렉션 식

- 컬렉션에만 사용할 수 있는 조건식으로, 컬렉션이 아닌 곳에 사용하면 오류가 발생한다.
- 빈 컬렉션 비교 식: {컬렉션 값 연관 경로} IS [NOT] EMPTY
- 컬렉션 멤버 식: {엔티티나 값} [NOT] MEMBER [OF] {컬렉션 값 연관경로}

```
// 빈 컬렉션 비교 식 - 컬렉션에 값이 비어있으면 참
SELECT m
FROM Member m
WHERE m.orders IS EMPTY
```

```
// 실제 동작 SQL
```

```
// NOT EXISTS 서브 쿼리를 이용하여 주문이 존재하지 않는 경우 참이 된다. 즉, orders 컬렉션이 비어있는 Member를 검색한다.
SELECT m.*
FROM members m
WHERE NOT EXISTS (SELECT 1 FROM orders o WHERE o.member_id = m.id)
```

```
// 컬렉션 멤버 식 - 전달된 멤버가 포함되어 있는 팀 조회
SELECT t
FROM Team t
WHERE :memberParam MEMBER OF t.members
```

1.5. 스칼라 식

- 숫자, 문자, 날짜, case, 엔티티 타입 같은 가장 기본적인 타입들을 스칼라 타입이라고 한다.
- 문자 함수

- CONCAT: 문자열 합치기
- SUBSTRING: 문자열 자르기
- TRIM: 공백 제거
- LOWER: 소문자 변환
- UPPER: 대문자 변환
- LENGTH: 문자열 길이
- LOCATE: 검색 위치부터 문자를 찾는다. 1부터 시작하고, 못 찾으면 0을 반환

- 수학 함수

- ABS: 절대값
- SQRT: 제곱근
- MOD: 나머지

- SIZE: 컬렉션의 크기

- INDEX: LIST 타입 컬렉션의 위치값을 반환, @OrderColumn을 사용하는 LIST 타입 일때만 사용한다.

- 날짜 함수

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

- 그 외

- CASE 식: CASE WHEN <조건식> THEN <스칼라식> ELSE <스칼라식> END
- COALESCE: 스칼라식을 차례대로 조회해서 null이 아니면 반환한다
- NULLIF: 두 값이 같으면 null 반환, 다르면 첫번째 값을 반환한다.

2. 조인

- 내부 조인 사용시 INNER 키워드는 생략이 가능하고, 외부 조인 사용시 OUTER 키워드는 생략이 가능하다.

- SQL 조인과 다르게 연관 필드를 사용해서 조인을 하기 때문에 엔티티에 연관 관계 명시가 필수적으로 되어 있어야 한다.

- 내부 조인: FROM Member m INNER JOIN m.team t

- 내부 조인: FROM Member m JOIN m.team t

- 외부 조인: FROM Member m LEFT OUTER JOIN m.team t

- 외부 조인: FROM Member m LEFT JOIN m.team t

```
// 내부 조인(INNER JOIN)
SELECT m
FROM Member m
    INNER JOIN m.team t

// 외부 조인(LEFT OUTER JOIN)
SELECT m
FROM Member m
    LEFT JOIN m.team t
```

2.1. 컬렉션 조인

- 일대다 관계나 다대다 관계처럼 컬렉션을 사용하는 곳에 조인하는 것을 말한다.

- 즉, 컬렉션 값 연관 필드를 이용하여 조인한다.

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

- Team과 Member 클래스는 양방향 관계를 가지고 있다.
- Team 엔티티는 members 필드를 통해 여러 개의 Member 엔티티와 일대다 관계를 가지고 있다.
이는 @OneToMany 어노테이션으로 표현된다.

- Member 엔티티는 team 필드를 통해 한 개의 Team 엔티티와 다대일 관계를 가지고 있다.

이는 @ManyToOne 어노테이션으로 표현되며, @JoinColumn 어노테이션을 통해 외래 키를 지정한다.

```
@Entity
@Table(name = "teams")
public class Team {

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

    private String name;

    @OneToMany(mappedBy = "team")
    private List<Member> members;

    // Constructors, getters, setters, and other methods...
}

@Entity
@Table(name = "members")
public class Member {

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

    private String name;

    @ManyToOne
    @JoinColumn(name = "team_id")
    private Team team;

    // Constructors, getters, setters, and other methods...
}
```

2.1. 조인 ON 절

- JPA 2.1부터 조인할 때 ON 절을 지원한다.

- ON 절을 사용하면 조인 대상을 필터링할 수 있고, 연관 관계없는 엔티티를 외부 조인할 수 있다.

```
// ON 절 미사용: WHERE로 조회
SELECT m, t
FROM Member m
    LEFT JOIN m.team t
WHERE t.name = 'A'
```

```
// 실행 쿼리: 회원 테이블에서 팀 정보를 조인하여 가져온다.
// 이후 WHERE 조건으로 팀 이름이 'A'인 회원 정보만 출력된다.
SELECT m.*, t.*
FROM Member m
    LEFT JOIN Team t ON m.team_id = t.id
WHERE t.name = 'A'
```

```
// ON 절 사용
SELECT m, t
FROM Member m
    LEFT JOIN m.team t ON t.name = 'A'
```

```
// 실행 쿼리: 회원 테이블에서 팀 정보를 조인하여 가져오는데, 팀 이름이 'A'인 팀 정보만을 조인한다.
// 팀 이름이 'A'인 회원 정보도 출력된다.
SELECT m.*, t.*
FROM Member m
    LEFT JOIN Team t ON m.team_id = t.id AND t.name = 'A'
```

2.2. 패치 조인

- JPQL에서 패치 조인(Fetch Join)은 엔티티와 연관된 컬렉션을 함께 로드하는 방법을 말한다. 일반적인 조인과 달리, 패치 조인은 연관된 엔티티를 지연 로딩(Lazy Loading) 대신 즉시 로드(Eager Loading)하여 성능을 최적화하는 데 사용된다.

- 연관된 엔티티의 컬렉션을 패치 조인으로 함께 로드하면, 데이터베이스에서 추가 쿼리를 실행하지 않고도 연관된 데이터를 한 번에 가져올 수 있다. 이는 N+1 문제를 방지하여 애플리케이션의 성능을 향상시킨다. N+1 문제는 연관된 엔티티를 하나씩 가져올 때 발생하는 쿼리 실행 횟수의 증가로 인해 성능 저하가 발생하는 상황을 의미한다.

- 패치 조인은 JOIN FETCH 키워드를 사용하여 수행된다. JOIN FETCH를 사용하여 연관된 엔티티를 패치 조인으로 함께 로드하면, 해당 연관된 엔티티는 즉시 로드되며, 지연 로딩이 발생하지 않는다.

- Eager(즉시 로딩): Eager 패치 전략은 연관된 엔티티를 쿼리할 때 함께 즉시 로딩하는 방식입니다. 쿼리할 때 연관된 엔티티도 함께 로드되므로, 즉시 필요한 데이터를 한 번의 쿼리로 가져올 수 있습니다. 하지만 관련된 엔티티가 많거나 복잡한 경우 성능 문제를 일으킬 수 있습니다.

- Lazy(지연 로딩): Lazy 패치 전략은 연관된 엔티티를 실제로 사용하는 순간까지 로딩을 미루는 방식입니다. 즉, 데이터가 실제로 필요한 시점에 SQL 쿼리를 실행하여 연관된 엔티티를 로드합니다. Lazy 로딩은 성능 최적화에 도움이 되며, 연관된 엔티티가 많은 경우에 유용합니다.

가. 일대다 관계의 패치 조인

- Department 엔티티와 Employee 엔티티는 일대다 관계입니다. 따라서 Department 엔티티의 employees 필드는 컬렉션 타입인 List<Employee>로 정의되어 있다. JPQL에서 JOIN FETCH를 사용하여 employees 컬렉션을 함께 로드하면, 관련된 모든 직원들을 한 번의 쿼리로 가져올 수 있다.

```
// JPQL query with collection join (Eager Fetching)
SELECT d
FROM Department d
    JOIN FETCH d.employees
WHERE d.id = :deptId
@Entity
public class Department {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @OneToMany(mappedBy = "department")
    private List<Employee> employees;

    // Getters, setters, and other methods...
}
```

```
@Entity
public class Employee {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    // Getters, setters, and other methods...
}
```

나. 다대다 관계의 패치 조인

- Student 엔티티와 Course 엔티티는 다대다 관계입니다. 따라서 Student 엔티티의 courses 필드는 컬렉션 타입인 List<Course>로 정의되어 있다. JPQL에서 JOIN FETCH를 사용하여 courses 컬렉션을 함께 로드하면, 관련된 모든 과목들을 한 번의 쿼리로 가져올 수 있다.

```
// JPQL query with collection join (Eager Fetching)
SELECT s
FROM Student s
    JOIN FETCH s.courses
WHERE s.id = :studentId
@Entity
public class Student {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @ManyToMany
    @JoinTable(name = "student_course",
               joinColumns = @JoinColumn(name = "student_id"),
               inverseJoinColumns = @JoinColumn(name = "course_id"))
    private List<Course> courses;

    // Getters, setters, and other methods...
```

```
}

@Entity
public class Course {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;

    // Getters, setters, and other methods...
}
```

2.4. 패치 조인과 일반 조인의 차이

- 일반 조인은 연관된 엔티티를 함께 로드하는 것이 아니라 해당 엔티티들을 사용하여 결과 집합을 생성한다. 즉, 필요한 순간까지 연관된 테이블의 데이터를 로드하지 않고, 데이터가 실제로 사용되는 시점에 SQL 쿼리를 실행하여 데이터를 로드하게 된다. 이러한 특징으로 일대다 연관 테이블에서 반복 사용하게 되면, 처음에 기준 테이블 SQL이 1번 실행되고, 다중으로 연관된 테이블 N번 조회 만큼 SQL이 실행되어 N + 1 문제가 발생할 수 있다.

- 패치 조인(Fetch Join)은 'JOIN FETCH' 키워드를 사용하여 연관된 엔티티의 컬렉션을 함께 로드하는 조인 방식이다. 즉, 즉시 로딩(Eager Loading)을 통해 일대다 혹은 다대다 관계의 연관 테이블이 있더라도 실행 시점에 모든 데이터를 조회하는 SQL이 실행된다. 때문에, 추가적인 쿼리를 실행하지 않고 모든 데이터를 함께 로드하므로 N + 1 문제를 해결할 수 있다.

```
// 일반 조인
// Member 엔티티의 내용만 가져온다. 조인된 테이블(TEAM)의 내용은 가져오지 않는다.
SELECT m
FROM Member m INNER JOIN m.team t

// 패치 조인
// Member 엔티티와 조인된 테이블(TEAM)의 내용을 가져온다.
SELECT m
FROM Member m INNER JOIN FETCH m.team t
```

2.5. 패치 조인의 특징과 주의점

가. 데이터 크기와 성능 문제

- 패치 조인은 연관된 엔티티나 컬렉션을 함께 로드하기 때문에, 데이터 크기가 크고 성능에 영향을 미칠 수 있는 경우가 있다.

- 예를 들어, 연관된 엔티티가 많은 경우에는 쿼리 결과 집합의 크기가 증가하여 성능에 부정적인 영향을 미칠 수 있다.

나. 중첩 로딩 문제

- 패치 조인은 연관된 엔티티를 함께 로드하기 때문에, 중첩 로딩(Nested Loading) 문제가 발생할 수 있다.

예를 들어, 여러 컬렉션과 깊게 연관된 엔티티들을 패치 조인으로 로드하면, 불필요한 데이터를 로드하여 성능에 영향을 미칠 수 있다.

2.6. 프로젝션

SELECT 절에 조회할 대상을 지정하는 것을 프로젝션(projection)이라 한다.

JPQL에서 프로젝션의 대상은 엔티티, 임베디드 타입, 스칼라 타입 3가지로 나눌 수 있다.

2.7. 엔티티 프로젝션

원하는 객체를 조회한다고 생각할 수 있다.

조회한 엔티티는 영속성 컨텍스트에서 관리된다.

```
SELECT m FROM Member m // member
SELECT m.team FROM Memher m // team, INNER JOIN이 발생한다.
SELECT t FROM Member m  JOIN m.team t // 명시적으로 INNER JOIN 사용이 권장된다.
```

2.8. 임베디드 타입 프로젝션

임베디드 타입은 조회의 시작점이 될 수 없다.

임베디드 타입은 엔티티 타입이 아닌 값 타입으로 영속성 컨텍스트에서 관리되지 않는다.

```
// "Member" 엔티티의 "name" 필드만을 프로젝션하여 쿼리 결과로 가져온다.
// 결과는 "name" 필드만을 갖는 컬렉션 또는 리스트로 반환될 수 있다.
SELECT m.name FROM Member m
```

2.9. 스칼라 타입 프로젝션

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

```
SELECT m.username FROM Member m
SELECT DISTINCT m.username FROM Member m
SELECT AVG(o.orderAmount) FROM Order o
```

2.10. 스칼라 타입 프로젝션 결과 조회 방법

TypedQuery: 컬럼 하나 조회

Query: 컬럼 여러개 조회 (Object[], List<Object[]>)

NEW 명령어: 컬럼 여러개를 객체(DTO)로 조회

DTO 클래스에는 패키지 명을 포함한 전체 클래스명을 입력한다.

순서와 타입이 일치하는 생성자가 필요하다.

```
// 단건 조회: 조회되는 컬럼이 1건으로 TypedQuery를 이용하여 반환 타입을 지정한다.
TypedQuery<String> query = em.createQuery("SELECT m.username FROM Member m", String.class);
List<String> resultList = query.getResultList();
```

```
// 여러 값 조회: Query를 이용하여 반환 타입을 Object[]로 받는다. 컬럼 순서에 따라 형변환을 한다.
Query query = em.createQuery("SELECT m.username, m.age, m.team FROM Member m");
List<Object[]> resultList = query.getResultList();

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

```
    Integer age = (Integer) row[1];
    Team team = (Team) row[2];
}
```

```
// new 명령어: NEW 명령어를 사용하면 Object[] 대신 바로 객체로 반환 받을 수 있다.
TypedQuery<UserDTO> query =
    em.createQuery("SELECT NEW jpabook.dto.UserDTO(m.username, m.age, m.team) FROM Member m",
UserDTO.class);

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

3. 쿼리 객체 (Query, TypedQuery)

JPQL의 쿼리 객체는 JPQL 쿼리를 생성하고 실행하는 데 사용되는 객체이다. JPA(Java Persistence API)에서는 javax.persistence.Query 인터페이스를 기반으로 쿼리 객체를 생성한다.

javax.persistence.Query 인터페이스는 JPQL 쿼리를 정의하고 데이터베이스에서 결과를 검색하기 위한 다양한 메서드를 제공. 이 인터페이스를 사용하여 동적인 쿼리를 만들고 파라미터 바인딩을 처리할 수 있다.

작성한 JPQL을 실행하기 위해서는 쿼리 객체를 만들어야 한다.

쿼리 객체는 TypedQuery와 Query 두 가지가 제공된다.

● TypedQuery: 반환 타입을 명확하게 지정할 수 있을 때 사용되는 객체

● Query: 반환 타입을 명확하게 지정할 수 없을 때 사용되는 객체로 조회 대상이 둘 이상이면 Object[]를 반환하고, 조회 대상이 하나이면 Object를 반환한다.

```
// TypedQuery: 반환 타입 지정
TypedQuery<Member> query = em.createQuery("select m from Member as m", Member.class);
List<Member> members = em.createQuery(query, Member.class).getResultList();
for (Member member : members) {
    ..
}
```

```
// Query: 반환 타입 미지정 (Object)
Query query = em.createQuery("select m.username, m.age from Member m");
List resultList = query.getResultList();
```

```
for (Object o : resultList) {
    Object[] objects = (Object[]) o;
    String userName = (String) objects[0];
    Integer age = (Integer) objects[1];
}
```

3.1. 결과 조회

쿼리 객체를 사용하여 JPQL 쿼리를 생성하고 실행함으로써, 엔티티 객체들을 검색하거나 조작하는 데 사용할 수 있다.

● getResultList(): 쿼리의 결과를 List로 반환한다.

```
- 결과가 하나 이상일 때 사용한다.
- 결과를 리스트로 반환하고, 결과가 없다면 빈 리스트를 반환한다.
```

● getSingleResult(): 단일 결과를 반환하며, 단일 결과가 없거나 여러 개의 결과가 있다면 예외가 발생함.

```
- 결과가 정확히 하나일 때 사용한다. 단일 객체를 반환한다.
- 만약, 결과가 없다면 javax.persistence.NoResultException 예외가 발생한다.
- 만약, 결과가 2개 이상이면 javax.persistence.NonUniqueResultException 예외가 발생한다.
```

● setParameter(): 쿼리에 파라미터를 바인딩 할 때 사용
- 이름과 위치 2가지 파라미터 바인딩 방식을 제공한다.

기타 실행과 관련된 메서드들 (예: executeUpdate(), executeDelete() 등)

```
//  query.getResultList(): 결과가 하나 이상 일 때
TypedQuery<Member> query = em.createQuery("select m from Member as m", Member.class);
List<Member> resultList = query.getResultList();

// query.getSingleResult(): 결과가 정확히 하나 일 때
TypedQuery<Member> query = em.createQuery("select m from Member as m where m.id = 1L",
Member.class);
Member result = query.getSingleResult();
```

3.2. 파라미터 바인딩

JPQL은 이름, 위치 2가지 파라미터 바인딩을 지원한다.

가. 이름 기준

- 이름 기준 파라미터는 앞에 : 를 사용한다.
- 파라미터 별칭(=:) 사이에는 공백이 있어도 상관없다. (ex: '=:empname', '= :empname', '= : empname')

```
Query query = em.createQuery("SELECT e FROM Employee e WHERE e.name = :empName");
query.setParameter("empName", "John Doe");
```

나. 위치 기준
- 위치 기준 파라미터는 ? 다음에 위치 값을 준다.
- 위치 값은 1부터 시작한다.

```
Query query = em.createQuery("SELECT e FROM Employee e WHERE e.name = ?1");
query.setParameter(1, "John Doe");
```

3.3. 페이징 API

- JPQL 쿼리 객체는 JPQL 쿼리 결과를 페이징하여 가져올 수 있는 2개의 메소드를 제공한다.

```
setFirstResult(int startPosition) : 조회 시작 위치(0부터 시작)
setMaxResult(int maxResult) : 조회할 데이터 수
```

```
int pageNumber = 2; // 가져올 페이지 번호
int pageSize = 10; // 한 페이지에 표시할 데이터 개수
int startPosition = (pageNumber - 1) * pageSize; // (2-1) * 10 = 10

// 11번째 데이터부터 10개를 조회한다. 즉, 11 ~ 20번 데이터를 조회한다.
TypedQuery<Member> query =
    em.createQuery("SELECT m FROM Member m ORDER BY m.username DESC", Member.class);
      .setFirstResult(startPosition);
      .setMaxResult(pageSize);
      .getResultList();
```

3.4. 벌크 연산

벌크 연산은 데이터베이스에 대량의 데이터를 한 번에 변경 또는 삭제하는 작업을 의미한다.

- JPQL 쿼리 객체는 벌크 연산을 위한 executeUpdate() 메소드를 제공한다. UPDATE, DELETE, INSERT 쿼리에 사용할 수 있다.

- 벌크 연산은 영속성 컨텍스트와 2차 캐시를 무시하고 데이터베이스에 직접 쿼리를 수행한다.

```
// 벌크 UPDATE 연산: "IT" 부서에 속한 모든 직원의 급여를 10% 인상시키는 벌크 UPDATE 연산을 수행한다.
// executeUpdate() 메서드를 사용하여 벌크 UPDATE 쿼리를 실행한다.
String jpqlUpdate = "UPDATE Employee e SET e.salary = e.salary * 1.1 WHERE e.department = :dept";
entityManager.createQuery(jpqlUpdate)
        .setParameter("dept", "IT")
        .executeUpdate();

// 벌크 DELETE 연산: "Sales" 부서에 속한 모든 직원을 벌크 DELETE 연산을 통해 삭제한다.
// executeUpdate() 메서드를 사용하여 벌크 DELETE 쿼리를 실행합니다.
String jpqlDelete = "DELETE FROM Employee e WHERE e.department = :dept";
entityManager.createQuery(jpqlDelete)
        .setParameter("dept", "Sales")
        .executeUpdate();
```

3.5. 벌크 연산 주의점

- JPA의 변경 감지 기능으로 많은 연산을 수행하려면 너무 많은 SQL이 실행되고, 성능이 좋지 않다. 때문에, 영속성 컨텍스트와 2차 캐시를 사용하지 않고 데이터베이스의 직접 쿼리를 수행하는 벌크 연산 메소드가 제공된다.

- 즉, 벌크 연산은 영속성 컨텍스트와 2차 캐시를 사용하지 않는데, 이러한 점에서 영속성 컨텍스트와 데이터베이스 간에 데이터 차이가 발생할 수 있다.

- EntityManager.refrest(entity): 벌크 연산 직후에 해당 메소드를 사용하여 데이터베이스에서 다시 해당 엔티티를 조회하도록 한다.

- 벌크 연산 먼저 실행: 벌크 연산을 가장 먼저 실행하면 이미 데이터베이스에 내용이 적용된 상태로, 이후 영속성 컨텍스트를 변경된 내용을 가져오게 된다.

- 벌크 연산 수행 후 영속성 컨텍스트 초기화: 영속성 컨텍스트가 초기화되면, 데이터베이스에서 다시 조회하게 된다.

```
Product product = em.find(Product.class, 1); // Product 조회, 영속성 컨텍스트에 보관된다.
assertThat(product.getPrice(), is(1000));

String sql = "UPDATE Product p SET p.prce = p.price * 1.1";
```

```
em.createQuery(sql).executeUpdate(); // 벌크 연산 수행, 실제 DB의 데이터가 바뀐다.
// A 지점

assertThat(product.getPrice(), is(1100)); // FAIL, 영속성 컨텍스트의 값은 변환되지 않았다.
// A 지점에서 em.refrest(product)를 수행하여 다시 조회하거나, 벌크 연산을 먼저 수행하는 방식으로 해결 가능하다.
```

4. JPQL의 쿼리를 실행하는 방법

JPQL의 쿼리를 실행하는 방법으로는 EntityManager의 createQuery 메소드와 createNamedQuery 메소드로 크게 두 가지가 있다.

또한, 스프링 Data JPA를 사용하는 경우 @Query 어노테이션을 사용할 수도 있다.

4.1. 동적 쿼리(createQuery)

EntityManager의 createQuery(jpql) 처럼 JPQL 쿼리를 직접 문자로 넘기는 방식을 동적 쿼리라고 한다.

```
String jpql = "SELECT e FROM Employee e WHERE e.department = :dept";
TypedQuery<Employee> query = entityManager.createQuery(jpql, Employee.class)
                        .setParameter("dept", department);
List<Employee> employees = query.getResultList();
```

4.2. 정적 쿼리(createNamedQuery)

@NamedQuery를 사용하여 엔티티 클래스에 미리 JPQL 쿼리를 정의하고, 미리 정의된 JPQL 쿼리를 이름으로 불러와 사용하는 방식을 정적 쿼리라고 한다.

Named 쿼리는 애플리케이션 로딩 시점에 JPQL 문법을 체크하고, 미리 파싱해두어 오류를 빨리 확인할 수 있고 사용하는 시점에는 파싱된 결과를 재사용하므로 성능상 이점이 있다.

정적 쿼리로는 @NamedQuery 어노테이션을 이용하여 자바 코드에 작성하는 방법이 있고, XML 문서에 작성하는 방법이 있다.

자바 코드에 작성하는 경우 엔티티 클래스에 @NamedQuery, @NamedQueries 어노테이션을 사용한다.

```
// 1개의 NamedQuery 작성
@Entity
@NamedQuery(name = "Member.findByUsername", query="SELECT m FROM Member WHERE
```

```
m.username = :username")
public class Member {
        ..
}

// 여러 개의 NamedQuery 작성
@Entity
@NamedQueries({
    @NamedQuery(name = "Member.findByUsername", query = "SELECT m FROM Member WHERE m.username = :username"),
    @NamedQuery(name = "Member.count", query = "SELECT COUNT(m) FROM Member m")
})
public class Member {
        ..
}

// @NamedQuery 실행
private static void findMember(EntityManager em, String username) {
    List<Member> resultList = em.createNamedQuery("Member.findByUsername", Member.class)
            .setParameter("username", username)
            .getResultList();
}
```

● 스프링 Data JPA 사용시

스프링 Data JPA에서 제공하는 @Query 어노테이션을 통해 JPQL 쿼리나 NativeSQL 쿼리를 직접 작성하여 사용할 수 있다.

@Query 어노테이션은 Repository 인터페이스의 메소드에 적용되며, 해당 메소드를 실행할 때 지정된 쿼리를 실행하게

● @Query와 @NamedQuery 차이점

@Query는 Repository 인터페이스의 메서드에 적용되고, @NamedQuery는 엔티티 클래스에 정의된다.

둘 다 JPQL을 사용하여 데이터베이스 작업을 정의하지만, 사용되는 컨텍스트와 쿼리를 호출하는 방식에 차이가 있다.

@Query는 JpaRepository 인터페이스를 통해 해당 메소드를 호출하는 방식으로 JPQL을 수행하지만,

@NamedQuery는 EntityManager나 TypedQuery를 사용하여 정의된 이름으로 쿼리를 호출해야 한다.

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    @Query("SELECT e FROM Employee e WHERE e.department = :dept")
    List<Employee> findByDepartment(@Param("dept") String department);
}
```

쿼리 실습

```
@Service
@Transactional
########## QueryService {
    @Autowired
    EntityManager em;
    // 동적쿼리 생성(장원영 검색하기)
    public List<Member> dynamicQuery(){
        String sql = "SELECT m FROM Member m WHERE m.memberId=:id";
        TypedQuery<Member> query = em.createQuery(sql, Member.class)
                .setParameter("id", "장원영");
        List<Member> memberList = query.getResultList();
        return memberList;
    }

    @Test
    @Transactional
    void dynamicQuery() {
        List<Member> memberList = queryService.dynamicQuery();
        for (Member member : memberList) {
            System.out.println(member.getMemberId());
            System.out.println(member.getName());
            System.out.println(member.getTeam().getTeamName());
        }
    }

    // 동적쿼리 생성(팀전체 검색하기)
    public List<Team> findAllTeam(){
        String sql = "SELECT t FROM Team t";
        Query query = em.createQuery(sql);
        List<Team> teamList = query.getResultList();
        return teamList;
    }

    @Test
    @Transactional
    void findAllTeam() {
        List<Team> teamList = queryService.findAllTeam();
        for (Team team : teamList) {
            System.out.println(team.getTeamId());
```

```
            System.out.println(team.getTeamName());
            for (Member member : team.getMemberList()) {
                System.out.println(member.toString());
            }
        }
    }

    // Member table에서 멤버 이름만 검색
    public List<String> findMemberName(){
        String sql = "SELECT m.name FROM Member m ";
        TypedQuery<String> query = em.createQuery(sql, String.class);
        List<String> nameList = query.getResultList();
        return nameList;
    }
    @Test
    @Transactional
    void findMemberName(){
        List<String> names = queryService.findMemberName();
        for (String name : names) {
            System.out.println(name);
        }
    }

    // 멤버 테이블 중 newJeans 소속 만 이름 출력
    public List<String> findMemberNewJeans(){
        String sql = "SELECT m.name FROM Member m " +
                "WHERE m.team.teamId=:teamName";
        TypedQuery<String> query = em.createQuery(sql, String.class);
        query.setParameter("teamName", "newJeans");
        List<String> nameList = query.getResultList();
        return nameList;
    }
    @Test
    @Transactional
    void findMemberNewJeans(){
        List<String> names = queryService.findMemberNewJeans();
        for (String name : names) {
            System.out.println(name);
        }
    }
```

```
    // 뉴진스 멤버 인원수 구하기
    public Long newJeansMemberCount(){
        String sql = "SELECT COUNT(m) FROM Member m " +
                " WHERE m.team.teamId=:teamName";
        Query query = em.createQuery(sql);
        query.setParameter("teamName", "newJeans");
        Long result = (Long)query.getSingleResult();
        return result;
    }

    @Test
    @Transactional
    void newJeansMemberCount(){
        Long count = queryService.newJeansMemberCount();
        System.out.println("Member Count = " + count);
    }


    // Dto로 멤버이름과 팀이름을 리스트로 받기
    public List<MemberDto> getMemberDto(){
        String sql = "SELECT NEW " +
                "com.example.jpaTest.dto.MemberDto(m.name, m.team.teamName) " +
                "FROM Member m";
        TypedQuery<MemberDto> query = em.createQuery(sql, MemberDto.class);
        List<MemberDto> dtos = query.getResultList();
        return dtos;
    }

    @Test
    @Transactional
    void getMemberDto(){
        List<MemberDto> dtos = queryService.getMemberDto();
        for (MemberDto dto : dtos) {
            System.out.println(dto);
        }
    }

#########  Member Class 선언하기
```

```
    @NamedQuery(name = "Member.findByMemberName",
            query="SELECT m FROM Member m WHERE m.name = :username")
    @NamedQueries({
    @NamedQuery(name = "Member.findByUsername",
        query = "SELECT m FROM Member m WHERE m.name = :username"),
    @NamedQuery(name = "Member.count",
        query = "SELECT COUNT(m) FROM Member m")
})

// @Named Query로 Member.name = "원영"찾기
public List<Member> findMemberNameNamedQuery() {
    TypedQuery<Member> query =
                    em.createNamedQuery("Member.findByMemberName", Member.class);
    query.setParameter("username", "원영");
    List<Member> wonyoungs = query.getResultList();
    return wonyoungs;
}

@Test
@Transactional
void findMemberNameNamedQuery(){
    List<Member> wonyoungs = queryService.findMemberNameNamedQuery();
    System.out.println(wonyoungs);
}

public List<MemberDto> findNativeQuery(){
    String str = "select m.name, t.team_name from member m " +
            "inner join team t " +
            "on m.team_id = t.team_id " +
            "where t.team_id = :teamId";
    NativeQuery<MemberDto> query = (NativeQuery<MemberDto>)
            em.createNativeQuery(str, MemberDto.class);
    query.setParameter("teamId", "newJeans");
    List<MemberDto> dtos = query.getResultList();
    return dtos;
}
@Test
@Transactional
void findNativeQuery(){
    List<MemberDto> dtos = queryService.findNativeQuery();
```

```
        System.out.println(dtos);
    }
```