3. 기본 문법

#1.인강/JPA활용편/querydsl/강의

- /시작 - JPQL vs Querydsl
- /기본 Q-Type 활용
- /검색 조건 쿼리
- /결과 조회
- /정렬
- /페이징
- /집합
- /조인 - 기본 조인
- /조인 - on절
- /조인 - 페치 조인
- /서브 쿼리
- /Case 문
- /상수, 문자 더하기

시작 - JPQL vs Querydsl

테스트 기본 코드

```
package study.querydsl;

import com.querydsl.jpa.impl.JPAQueryFactory;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.transaction.annotation.Transactional;
import study.querydsl.entity.Member;
import study.querydsl.entity.QMember;
import study.querydsl.entity.Team;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

import static org.assertj.core.api.Assertions.*;
import static study.querydsl.entity.QMember.*;
```

```java
@SpringBootTest
@Transactional
public class QuerydslBasicTest {

    @PersistenceContext
    EntityManager em;

    @BeforeEach
    public void before() {
        Team teamA = new Team("teamA");
        Team teamB = new Team("teamB");
        em.persist(teamA);
        em.persist(teamB);

        Member member1 = new Member("member1", 10, teamA);
        Member member2 = new Member("member2", 20, teamA);
        Member member3 = new Member("member3", 30, teamB);
        Member member4 = new Member("member4", 40, teamB);

        em.persist(member1);
        em.persist(member2);
        em.persist(member3);
        em.persist(member4);
    }

}
```

- 지금부터는 이 예제로 실행

Querydsl vs JPQL

```java
@Test
public void startJPQL() {
    //member1을 찾아라.
    String qlString =
            "select m from Member m " +
            "where m.username = :username";

    Member findMember = em.createQuery(qlString, Member.class)
            .setParameter("username", "member1")
```

```
            .getSingleResult();

    assertThat(findMember.getUsername()).isEqualTo("member1");

}

@Test
public void startQuerydsl() {
    //member1을 찾아라.
    JPAQueryFactory queryFactory = new JPAQueryFactory(em);
    QMember m = new QMember("m");

    Member findMember = queryFactory
            .select(m)
            .from(m)
            .where(m.username.eq("member1"))//파라미터 바인딩 처리
            .fetchOne();

    assertThat(findMember.getUsername()).isEqualTo("member1");

}
```

- `EntityManager`로 `JPAQueryFactory` 생성
- Querydsl은 JPQL 빌더
- JPQL: 문자(실행 시점 오류), Querydsl: 코드(컴파일 시점 오류)
- JPQL: 파라미터 바인딩 직접, Querydsl: 파라미터 바인딩 자동 처리

JPAQueryFactory를 필드로

```
package study.querydsl;

import com.querydsl.jpa.impl.JPAQueryFactory;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.transaction.annotation.Transactional;
import study.querydsl.entity.Member;
import study.querydsl.entity.QMember;
import study.querydsl.entity.Team;
```

```java
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

import static org.assertj.core.api.Assertions.*;
import static study.querydsl.entity.QMember.*;

@SpringBootTest
@Transactional
public class QuerydslBasicTest {

    @PersistenceContext
    EntityManager em;

    JPAQueryFactory queryFactory;

    @BeforeEach
    public void before() {
        queryFactory = new JPAQueryFactory(em);
        //…
    }

    @Test
    public void startQuerydsl2() {
        //member1을 찾아라.
        QMember m = new QMember("m");

        Member findMember = queryFactory
                .select(m)
                .from(m)
                .where(m.username.eq("member1"))
                .fetchOne();
        assertThat(findMember.getUsername()).isEqualTo("member1");
    }

}
```

- JPAQueryFactory를 필드로 제공하면 동시성 문제는 어떻게 될까? 동시성 문제는 JPAQueryFactory를 생성할 때 제공하는 EntityManager(em)에 달려있다. 스프링 프레임워크는 여러 쓰레드에서 동시에 같은 EntityManager에 접근해도, 트랜잭션 마다 별도의 영속성 컨텍스트를 제공하기 때문에, 동시성 문제는 걱정하

지 않아도 된다.

기본 Q-Type 활용

Q클래스 인스턴스를 사용하는 2가지 방법

```
QMember qMember = new QMember("m"); //별칭 직접 지정
QMember qMember = QMember.member; //기본 인스턴스 사용
```

기본 인스턴스를 static import와 함께 사용

```
import static study.querydsl.entity.QMember.*;

@Test
public void startQuerydsl3() {
    //member1을 찾아라.
    Member findMember = queryFactory
            .select(member)
            .from(member)
            .where(member.username.eq("member1"))
            .fetchOne();

    assertThat(findMember.getUsername()).isEqualTo("member1");
}
```

다음 설정을 추가하면 실행되는 JPQL을 볼 수 있다.

```
spring.jpa.properties.hibernate.use_sql_comments: true
```

> 참고: 같은 테이블을 조인해야 하는 경우가 아니면 기본 인스턴스를 사용하자

검색 조건 쿼리

기본 검색 쿼리

```
@Test
public void search() {
```

```
    Member findMember = queryFactory
            .selectFrom(member)
            .where(member.username.eq("member1")
                    .and(member.age.eq(10)))
            .fetchOne();

    assertThat(findMember.getUsername()).isEqualTo("member1");
}
```

- 검색 조건은 `.and()`, `.or()`를 메서드 체인으로 연결할 수 있다.

> 참고: `select`, `from`을 `selectFrom`으로 합칠 수 있음

JPQL이 제공하는 모든 검색 조건 제공

```
member.username.eq("member1") // username = 'member1'
member.username.ne("member1") //username != 'member1'
member.username.eq("member1").not() // username != 'member1'

member.username.isNotNull() //이름이 is not null

member.age.in(10, 20) // age in (10,20)
member.age.notIn(10, 20) // age not in (10, 20)
member.age.between(10,30) //between 10, 30

member.age.goe(30) // age >= 30
member.age.gt(30) // age > 30
member.age.loe(30) // age <= 30
member.age.lt(30) // age < 30

member.username.like("member%") //like 검색
member.username.contains("member") // like ‘%member%’ 검색
member.username.startsWith("member") //like ‘member%’ 검색
...
```

AND 조건을 파라미터로 처리

```
@Test
public void searchAndParam() {
    List<Member> result1 = queryFactory
            .selectFrom(member)
            .where(member.username.eq("member1"),
```

```
                        member.age.eq(10))
                .fetch();
        assertThat(result1.size()).isEqualTo(1);
    }
```

- `where()`에 파라미터로 검색조건을 추가하면 `AND` 조건이 추가됨
- 이 경우 `null` 값은 무시 ➜ 메서드 추출을 활용해서 동적 쿼리를 깔끔하게 만들 수 있음 ➜ 뒤에서 설명

결과 조회

- `fetch()` : 리스트 조회, 데이터 없으면 빈 리스트 반환
- `fetchOne()` : 단 건 조회
  - 결과가 없으면 : `null`
  - 결과가 둘 이상이면 : `com.querydsl.core.NonUniqueResultException`
- `fetchFirst()` : `limit(1).fetchOne()`
- `fetchResults()` : 페이징 정보 포함, total count 쿼리 추가 실행
- `fetchCount()` : count 쿼리로 변경해서 count 수 조회

```
//List
List<Member> fetch = queryFactory
        .selectFrom(member)
        .fetch();

//단 건
Member findMember1 = queryFactory
        .selectFrom(member)
        .fetchOne();

//처음 한 건 조회
Member findMember2 = queryFactory
        .selectFrom(member)
        .fetchFirst();

//페이징에서 사용
QueryResults<Member> results = queryFactory
        .selectFrom(member)
        .fetchResults();
```

```
//count 쿼리로 변경
long count = queryFactory
        .selectFrom(member)
        .fetchCount();
```

정렬

```
/**
 * 회원 정렬 순서
 * 1. 회원 나이 내림차순(desc)
 * 2. 회원 이름 올림차순(asc)
 * 단 2에서 회원 이름이 없으면 마지막에 출력(nulls last)
 */
@Test
public void sort() {
    em.persist(new Member(null, 100));
    em.persist(new Member("member5", 100));
    em.persist(new Member("member6", 100));

    List<Member> result = queryFactory
            .selectFrom(member)
            .where(member.age.eq(100))
            .orderBy(member.age.desc(), member.username.asc().nullsLast())
            .fetch();

    Member member5 = result.get(0);
    Member member6 = result.get(1);
    Member memberNull = result.get(2);
    assertThat(member5.getUsername()).isEqualTo("member5");
    assertThat(member6.getUsername()).isEqualTo("member6");
    assertThat(memberNull.getUsername()).isNull();
}
```

- `desc()`, `asc()`: 일반 정렬
- `nullsLast()`, `nullsFirst()`: null 데이터 순서 부여

페이징

조회 건수 제한

```
@Test
public void paging1() {
    List<Member> result = queryFactory
            .selectFrom(member)
            .orderBy(member.username.desc())
            .offset(1) //0부터 시작(zero index)
            .limit(2) //최대 2건 조회
            .fetch();

    assertThat(result.size()).isEqualTo(2);
}
```

전체 조회 수가 필요하면?

```
@Test
public void paging2() {
    QueryResults<Member> queryResults = queryFactory
            .selectFrom(member)
            .orderBy(member.username.desc())
            .offset(1)
            .limit(2)
            .fetchResults();

    assertThat(queryResults.getTotal()).isEqualTo(4);
    assertThat(queryResults.getLimit()).isEqualTo(2);
    assertThat(queryResults.getOffset()).isEqualTo(1);
    assertThat(queryResults.getResults().size()).isEqualTo(2);
}
```

> 주의: count 쿼리가 실행되니 성능상 주의!

> 참고: 실무에서 페이징 쿼리를 작성할 때, 데이터를 조회하는 쿼리는 여러 테이블을 조인해야 하지만, count 쿼리는 조인이 필요 없는 경우도 있다. 그런데 이렇게 자동화된 count 쿼리는 원본 쿼리와 같이 모두 조인을 해버리기 때문에 성능이 안나올 수 있다. count 쿼리에 조인이 필요없는 성능 최적화가 필요하다면, count 전용 쿼리를 별도로 작성해야 한다.

집합

집합 함수

```
/**
 * JPQL
 * 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
 */
@Test
public void aggregation() throws Exception {
    List<Tuple> result = queryFactory
            .select(member.count(),
                    member.age.sum(),
                    member.age.avg(),
                    member.age.max(),
                    member.age.min())
            .from(member)
            .fetch();

    Tuple tuple = result.get(0);
    assertThat(tuple.get(member.count())).isEqualTo(4);
    assertThat(tuple.get(member.age.sum())).isEqualTo(100);
    assertThat(tuple.get(member.age.avg())).isEqualTo(25);
    assertThat(tuple.get(member.age.max())).isEqualTo(40);
    assertThat(tuple.get(member.age.min())).isEqualTo(10);
}
```

- JPQL이 제공하는 모든 집합 함수를 제공한다.
- tuple은 프로젝션과 결과반환에서 설명한다.

GroupBy 사용

```
/**
 * 팀의 이름과 각 팀의 평균 연령을 구해라.
 */
@Test
```

```java
public void group() throws Exception {
    List<Tuple> result = queryFactory
            .select(team.name, member.age.avg())
            .from(member)
            .join(member.team, team)
            .groupBy(team.name)
            .fetch();

    Tuple teamA = result.get(0);
    Tuple teamB = result.get(1);

    assertThat(teamA.get(team.name)).isEqualTo("teamA");
    assertThat(teamA.get(member.age.avg())).isEqualTo(15);

    assertThat(teamB.get(team.name)).isEqualTo("teamB");
    assertThat(teamB.get(member.age.avg())).isEqualTo(35);
}
```

`groupBy`, 그룹화된 결과를 제한하려면 `having`

groupBy(), having() 예시

```
  …
    .groupBy(item.price)
    .having(item.price.gt(1000))
  …
```

조인 - 기본 조인

기본 조인

조인의 기본 문법은 첫 번째 파라미터에 조인 대상을 지정하고, 두 번째 파라미터에 별칭(alias)으로 사용할 Q 타입을 지정하면 된다.

```
join(조인 대상, 별칭으로 사용할 Q타입)
```

기본 조인

```
/**
 * 팀 A에 소속된 모든 회원
 */
@Test
public void join() throws Exception {
    QMember member = QMember.member;
    QTeam team = QTeam.team;

    List<Member> result = queryFactory
            .selectFrom(member)
            .join(member.team, team)
            .where(team.name.eq("teamA"))
            .fetch();

    assertThat(result)
            .extracting("username")
            .containsExactly("member1", "member2");
}
```

- `join()`, `innerJoin()` : 내부 조인(inner join)
- `leftJoin()` : left 외부 조인(left outer join)
- `rightJoin()` : rigth 외부 조인(rigth outer join)
- JPQL의 `on`과 성능 최적화를 위한 `fetch` 조인 제공 ➜ 다음 on 절에서 설명

세타 조인

연관관계가 없는 필드로 조인

```
/**
 * 세타 조인(연관관계가 없는 필드로 조인)
 * 회원의 이름이 팀 이름과 같은 회원 조회
 */
@Test
public void theta_join() throws Exception {
    em.persist(new Member("teamA"));
    em.persist(new Member("teamB"));

    List<Member> result = queryFactory
            .select(member)
            .from(member, team)
```

```
                .where(member.username.eq(team.name))
                .fetch();

        assertThat(result)
                .extracting("username")
                .containsExactly("teamA", "teamB");
    }
```

- from 절에 여러 엔티티를 선택해서 세타 조인
- 외부 조인 불가능 ➜ 다음에 설명할 조인 on을 사용하면 외부 조인 가능

조인 - on절

- ON절을 활용한 조인(JPA 2.1부터 지원)
    1. 조인 대상 필터링
    2. 연관관계 없는 엔티티 외부 조인

1. 조인 대상 필터링

예) 회원과 팀을 조인하면서, 팀 이름이 teamA인 팀만 조인, 회원은 모두 조회

```
/**
 * 예) 회원과 팀을 조인하면서, 팀 이름이 teamA인 팀만 조인, 회원은 모두 조회
 * JPQL: SELECT m, t FROM Member m LEFT JOIN m.team t on t.name = 'teamA'
 * SQL: SELECT m.*, t.* FROM Member m LEFT JOIN Team t ON m.TEAM_ID=t.id and
t.name='teamA'
 */
@Test
public void join_on_filtering() throws Exception {
    List<Tuple> result = queryFactory
            .select(member, team)
            .from(member)
            .leftJoin(member.team, team).on(team.name.eq("teamA"))
            .fetch();

    for (Tuple tuple : result) {
        System.out.println("tuple = " + tuple);
```

```
    }

}
```

결과

```
t=[Member(id=3, username=member1, age=10), Team(id=1, name=teamA)]
t=[Member(id=4, username=member2, age=20), Team(id=1, name=teamA)]
t=[Member(id=5, username=member3, age=30), null]
t=[Member(id=6, username=member4, age=40), null]
```

> 참고: on 절을 활용해 조인 대상을 필터링 할 때, 외부조인이 아니라 내부조인(inner join)을 사용하면, where 절에서 필터링 하는 것과 기능이 동일하다. 따라서 on 절을 활용한 조인 대상 필터링을 사용할 때, 내부조인 이면 익숙한 where 절로 해결하고, 정말 외부조인이 필요한 경우에만 이 기능을 사용하자.

2. 연관관계 없는 엔티티 외부 조인

예) 회원의 이름과 팀의 이름이 같은 대상 **외부 조인**

```
/**
 * 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
 */
@Test
public void join_on_no_relation() throws Exception {
    em.persist(new Member("teamA"));
    em.persist(new Member("teamB"));

    List<Tuple> result = queryFactory
            .select(member, team)
            .from(member)
            .leftJoin(team).on(member.username.eq(team.name))
            .fetch();

    for (Tuple tuple : result) {
        System.out.println("t=" + tuple);
    }
```

```
}
```

- 하이버네이트 5.1부터 `on`을 사용해서 서로 관계가 없는 필드로 외부 조인하는 기능이 추가되었다. 물론 내부 조인도 가능하다.
- 주의! 문법을 잘 봐야 한다. **leftJoin()** 부분에 일반 조인과 다르게 엔티티 하나만 들어간다.
  - 일반조인: `leftJoin(member.team, team)`
  - on조인: `from(member).leftJoin(team).on(xxx)`

결과

```
t=[Member(id=3, username=member1, age=10), null]
t=[Member(id=4, username=member2, age=20), null]
t=[Member(id=5, username=member3, age=30), null]
t=[Member(id=6, username=member4, age=40), null]
t=[Member(id=7, username=teamA, age=0), Team(id=1, name=teamA)]
t=[Member(id=8, username=teamB, age=0), Team(id=2, name=teamB)]
```

조인 - 페치 조인

페치 조인은 SQL에서 제공하는 기능은 아니다. SQL조인을 활용해서 연관된 엔티티를 SQL 한번에 조회하는 기능이다. 주로 성능 최적화에 사용하는 방법이다.

페치 조인 미적용

지연로딩으로 Member, Team SQL 쿼리 각각 실행

```
@PersistenceUnit
EntityManagerFactory emf;

@Test
public void fetchJoinNo() throws Exception {
    em.flush();
    em.clear();

    Member findMember = queryFactory
            .selectFrom(member)
            .where(member.username.eq("member1"))
            .fetchOne();

    boolean loaded =
emf.getPersistenceUnitUtil().isLoaded(findMember.getTeam());
    assertThat(loaded).as("페치 조인 미적용").isFalse();
```

```
}
```

페치 조인 적용

즉시로딩으로 Member, Team SQL 쿼리 조인으로 한번에 조회

```
@Test
public void fetchJoinUse() throws Exception {
    em.flush();
    em.clear();

    Member findMember = queryFactory
            .selectFrom(member)
            .join(member.team, team).fetchJoin()
            .where(member.username.eq("member1"))
            .fetchOne();

    boolean loaded =
emf.getPersistenceUnitUtil().isLoaded(findMember.getTeam());
    assertThat(loaded).as("페치 조인 적용").isTrue();
}
```

사용방법

- `join(), leftJoin()` 등 조인 기능 뒤에 `fetchJoin()` 이라고 추가하면 된다.

> 참고: 페치 조인에 대한 자세한 내용은 JPA 기본편이나, 활용2편을 참고하자

서브 쿼리

`com.querydsl.jpa.JPAExpressions` 사용

서브 쿼리 eq 사용

```
/**
 * 나이가 가장 많은 회원 조회
 */
@Test
public void subQuery() throws Exception {
```

```
    QMember memberSub = new QMember("memberSub");

    List<Member> result = queryFactory
            .selectFrom(member)
            .where(member.age.eq(
                    JPAExpressions
                            .select(memberSub.age.max())
                            .from(memberSub)
            ))
            .fetch();

    assertThat(result).extracting("age")
            .containsExactly(40);
}
```

서브 쿼리 goe 사용

```
/**
 * 나이가 평균 나이 이상인 회원
 */
@Test
public void subQueryGoe() throws Exception {
    QMember memberSub = new QMember("memberSub");

    List<Member> result = queryFactory
            .selectFrom(member)
            .where(member.age.goe(
                    JPAExpressions
                            .select(memberSub.age.avg())
                            .from(memberSub)
            ))
            .fetch();

    assertThat(result).extracting("age")
            .containsExactly(30,40);
}
```

서브쿼리 여러 건 처리 in 사용

```
/**
 * 서브쿼리 여러 건 처리, in 사용
 */
@Test
public void subQueryIn() throws Exception {
```

```
    QMember memberSub = new QMember("memberSub");

    List<Member> result = queryFactory
            .selectFrom(member)
            .where(member.age.in(
                    JPAExpressions
                            .select(memberSub.age)
                            .from(memberSub)
                            .where(memberSub.age.gt(10))
            ))
            .fetch();

    assertThat(result).extracting("age")
            .containsExactly(20, 30, 40);
}
```

select 절에 subquery

```
List<Tuple> fetch = queryFactory
        .select(member.username,
                JPAExpressions
                        .select(memberSub.age.avg())
                        .from(memberSub)
        ).from(member)
        .fetch();

for (Tuple tuple : fetch) {
    System.out.println("username = " + tuple.get(member.username));
    System.out.println("age = " +
tuple.get(JPAExpressions.select(memberSub.age.avg())
            .from(memberSub)));
}
```

static import 활용

```
import static com.querydsl.jpa.JPAExpressions.select;

List<Member> result = queryFactory
        .selectFrom(member)
        .where(member.age.eq(
                select(memberSub.age.max())
                        .from(memberSub)
```

```
        ))
        .fetch();
```

from 절의 서브쿼리 한계

JPA JPQL 서브쿼리의 한계점으로 from 절의 서브쿼리(인라인 뷰)는 지원하지 않는다. 당연히 Querydsl도 지원하지 않는다. 하이버네이트 구현체를 사용하면 select 절의 서브쿼리는 지원한다. Querydsl도 하이버네이트 구현체를 사용하면 select 절의 서브쿼리를 지원한다.

from 절의 서브쿼리 해결방안

1. 서브쿼리를 join으로 변경한다. (가능한 상황도 있고, 불가능한 상황도 있다.)
2. 애플리케이션에서 쿼리를 2번 분리해서 실행한다.
3. nativeSQL을 사용한다.

Case 문

select, 조건절(where), order by에서 사용 가능

단순한 조건

```
List<String> result = queryFactory
        .select(member.age
                .when(10).then("열살")
                .when(20).then("스무살")
                .otherwise("기타"))
        .from(member)
        .fetch();
```

복잡한 조건

```
List<String> result = queryFactory
        .select(new CaseBuilder()
                .when(member.age.between(0, 20)).then("0~20살")
                .when(member.age.between(21, 30)).then("21~30살")
                .otherwise("기타"))
        .from(member)
        .fetch();
```

orderBy에서 Case 문 함께 사용하기 예제

> **참고: 강의 이후 추가된 내용입니다.**

예를 들어서 다음과 같은 임의의 순서로 회원을 출력하고 싶다면?

1. 0 ~ 30살이 아닌 회원을 가장 먼저 출력
2. 0 ~ 20살 회원 출력
3. 21 ~ 30살 회원 출력

```
NumberExpression<Integer> rankPath = new CaseBuilder()
        .when(member.age.between(0, 20)).then(2)
        .when(member.age.between(21, 30)).then(1)
        .otherwise(3);

List<Tuple> result = queryFactory
        .select(member.username, member.age, rankPath)
        .from(member)
        .orderBy(rankPath.desc())
        .fetch();

for (Tuple tuple : result) {
    String username = tuple.get(member.username);
    Integer age = tuple.get(member.age);
    Integer rank = tuple.get(rankPath);
    System.out.println("username = " + username + " age = " + age + " rank = " +
rank);
}
```

Querydsl은 자바 코드로 작성하기 때문에 `rankPath` 처럼 복잡한 조건을 변수로 선언해서 `select` 절, `orderBy` 절에서 함께 사용할 수 있다.

```
결과
username = member4 age = 40 rank = 3
username = member1 age = 10 rank = 2
username = member2 age = 20 rank = 2
username = member3 age = 30 rank = 1
```

상수, 문자 더하기

상수가 필요하면 `Expressions.constant(xxx)` 사용

```
Tuple result = queryFactory
        .select(member.username, Expressions.constant("A"))
        .from(member)
        .fetchFirst();
```

> 참고: 위와 같이 최적화가 가능하면 SQL에 constant 값을 넘기지 않는다. 상수를 더하는 것 처럼 최적화가 어려우면 SQL에 constant 값을 넘긴다.

문자 더하기 concat

```
String result = queryFactory
        .select(member.username.concat("_").concat(member.age.stringValue()))
        .from(member)
        .where(member.username.eq("member1"))
        .fetchOne();
```

- 결과: member1_10

> 참고: `member.age.stringValue()` 부분이 중요한데, 문자가 아닌 다른 타입들은 `stringValue()` 로 문자로 변환할 수 있다. 이 방법은 ENUM을 처리할 때도 자주 사용한다.