- **Criteria 쿼리 (Criteria Query)**
  - **Criteria 기초**
  - **Criteria 쿼리 생성**
  - **조회**
    - - 조회 대상을 한 건, 여러 건 지정
    - - DISTINCT
    - - NEW, construct()
    - - 튜플
  - **집합**
    - - GROUP BY
    - - HAVING
  - **정렬**
  - **조인**
  - **서브 쿼리**
  - **IN 식**
  - **CASE 식**
  - **파라미터 정의**
  - **네이티브 함수 호출**
  - **동적 쿼리**
  - **함수 정리**
  - **Criteria 메타 모델 API**

Criteria 쿼리 (Criteria Query)

Criteria 쿼리는 JPQL을 자바 코드로 작성하도록 도와주는 빌더 클래스 API다. Criteria를 사용하면 문자가 아닌 코드로 JPQL을 작성하므로 문법 오류를 컴파일 단계에서 잡을 수 있고 문자 기반의 JPQL보다 동적 쿼리를 안전하게 생성할 수 있는 장점이 있다. 하지만 실제 Criteria를 사용해서 개발해보면 코드가 복잡하고 장황해서 직관적으로 이해가 힘들다는 단점도 있다.

Criteria는 결국 JPQL의 생성을 돕는 클래스 모음이다. 따라서 내용 대부분이 JPQL과 중복되므로 사용법 위주로 알아보자.

Criteria 기초

Criteria API는 `javax.persistence.criteria` 패키지에 있다.

가장 단순한 Criteria 쿼리부터 살펴보자.

====== Criteria 쿼리 시작 ======

```
//JPQL: select m from Member m

CriteriaBuilder cb = em.getCriteriaBuilder(); //1. Criteria 쿼리 빌더

CriteriaQuery<Member> cq = cb.createQuery(Member.class); //2. Criteria 생성, 반환
Root<Member> m = cq.from(Member.class); //3. FROM 절
cq.select(m); //4. SELECT 절

TypedQuery<Member> query = em.createQuery(cq);
List<Member> members = query.getResultList();
```

모든 회원 엔티티를 조회하는 단순한 JPQL을 Criteria로 작성해보자. 이해를 돕기 위해 Criteria의 결과로 생성된 JPQL을 첫 줄에 주석으로 남겨두었다.

1. Criteria 쿼리를 생성하려면 먼저 Criteria 빌더(`CriteriaBuilder`)를 얻어야 한다. Criteria 빌더는 `EntityManager` 나 `EntityManagerFactory` 에서 얻을 수 있다.
2. Criteria 쿼리 빌더에서 Criteria 쿼리(`CriteriaQuery`)를 생성한다. 이때 반환 타입을 지정할 수 있다.
3. FROM 절을 생성한다. 반환된 값 `m` 은 Criteria에서 사용하는 특별한 별칭이다. `m` 을 조회의 시작점이라는 의미로 쿼리 루트(`Root`)라 한다.

4. SELECT 절을 생성한다.

이렇게 Criteria 쿼리를 완성하고 나면 다음 순서는 JPQL과 같다. `em.createQuery(cq)`에 완성된 Criteria 쿼리를 넣어주기만 하면 된다.

이번에는 검색 조건(`where`)과 정렬(`order by`)을 추가해보자.

====== 검색 조건 추가 ======

```
// JPQL
// select m from Member m
// where m.username='회원1'
// order by m.age desc

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Member> cq = cb.createQuery(Member.class);
Root<Member> m = cq.from(Member.class); //FROM 절 생성

Predicate usernameEqual = cb.equal(m.get("username"), "회원1"); // 1. 검색 조건 정의
javax.persistence.criteria.Order ageDesc = cb.desc(m.get("age")); // 2. 정렬 조건

//3. 쿼리 생성
cq.select(m)
    .where(usernameEqual) //WHERE 절 생성
    .orderBy(ageDesc); //ORDER BY 절 생성

List<Member> resultList = em.createQuery(cq).getResultList();
```

이전에 보았던 기본 쿼리에 검색 조건과 정렬 조건을 추가했다.

1. 검색 조건을 정의한 부분을 보면 `m.get("username")`으로 되어 있는데 `m`은 회원 엔티티의 별칭이다. 이것은 JPQL 에서 `m.username`과 같은 표현이다. 그리고 `cb.equal(A,B)`는 이름 그대로 A = B 라는 뜻이다. 따라서 `cb.equal(m.get("username"), "회원1")`는 JPQL에서 `m.username = '회원1'` 과 같은 표현이다.
2. 정렬 조건을 정의하는 코드인 `cb.desc(m.get("age"))`는 JPQL의 `m.age desc`와 같은 표현이다.
3. 만들어둔 조건을 `where` , `orderBy` 에 넣어서 원하는 쿼리를 생성한다.

Criteria는 검색 조건부터 정렬까지 Criteria 빌더(`CriteriaBuilder`)를 사용해서 코드를 완성한다.

쿼리 루트(Query Root)와 별칭

- `Root<Member> m = cq.from(Member.class);` 여기서 `m`이 쿼리 루트다.
- 쿼리 루트는 조회의 시작점이다.
- Criteria에서 사용되는 특별한 별칭이다. JPQL의 별칭이라 생각하면 된다.
- 별칭은 엔티티에만 부여할 수 있다.

경로 표현식(Path Expression)

Criteria는 코드로 JPQL을 완성하는 도구다. 따라서 경로 표현식도 있다.

- `m.get("username")` 는 JPQL의 `m.username`과 같다.
- `m.get("team").get("name")` 는 JPQL의 `m.team.name`과 같다.

다음으로 10살을 초과하는 회원을 조회하고 나이 역순으로 정렬해보자.

==== 숫자 타입 검색 ====

```
// select m from Member m
// where m.age > 10 order by m.age desc

Root<Member> m = cq.from(Member.class);
Predicate ageGt = cb.greaterThan(m.<Integer>get("age"), 10); // 타입 정보 필요

cq.select(m);
cq.where(ageGt);
cq.orderBy(cb.desc(m.get("age")));
```

`cb.greaterThan(m.<Integer>get("age"), 10)`를 보면 메서드 이름을 보고 `A > B`라는 것이 바로 이해가 될 것이다. 약간 의아한 부분은 `m.<Integer>get("age")`에서 제네릭으로 타입 정보를 준 코드다. `m.get("age")`에서는 “age”의 타입 정보를 알지 못한다. 따라서 제네릭으로 반환 타입 정보를 명시해야 한다. (보통 `String` 같은 문자 타입은 지정하지 않아도 된다.)

참고로 `greaterThan()` 대신에 `gt()`를 사용해도 된다.

이제 본격적으로 Criteria API를 살펴보자.

Criteria 쿼리 생성

Criteria를 사용하려면 `CriteriaBuilder.createQuery()` 메서드로 Criteria 쿼리 (`CriteriaQuery`)를 생성하면 된다.

```
public interface CriteriaBuilder {

        CriteriaQuery<Object> createQuery(); // 조회값 반환 타입 : Object
        <T> CriteriaQuery<T> createQuery(Class<T> resultClass); // 조회값 반환 타입
        CriteriaQuery<Tuple> createTupleQuery(); // 조회값 반환 타입 : Tuple
        ...
}
```

Criteria 쿼리를 생성할 때 파라미터로 쿼리 결과에 대한 반환 타입을 지정할 수 있다. 예를 들어 `CriteriaQuery`를 생성할 때 `Member.class`를 반환 타입으로 지정하면 `em.createQuery(cq)`에서 반환 타입을 지정하지 않아도 된다.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Member> cq = cb.createQuery(Member.class); //Member를 반환 타입으로 지
...
//위에서 Member를 타입으로 지정했으므로 지정하지 않아도 Member 타입을 반환
List<Member> resultList = em.createQuery(cq).getResultList();
```

반환 타입을 지정할 수 없거나 반환 타입이 둘 이상이면 타입을 지정하지 않고 `Object`로 반환받으면 된다.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object> cq = cb.createQuery(); // 조회값 반환 타입 : Object
...
List<Object> resultList = em.createQuery(cq).getResultList();
```

물론 반환 타입이 둘 이상이면 다음 코드처럼 `Object[]`을 사용하는 것이 편리하다.(`Object[]`을 반환 받는 이유는 JPQL에서 설명했다.) 반환 타입이 둘 이상인 예제는 바로 뒤에 있는 `multiselect`에서 보자.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class); // 조회값 반환 타입 : (
...
List<Object[]> resultList = em.createQuery(cq).getResultList();
```

반환 타입을 튜플로 받고 싶으면 튜플을 사용하면 된다. 튜플에 대해서는 조금 뒤에 알아보겠다.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Tuple> cq = cb.createTupleQuery(); // 조회값 반환 타입 : Tuple
...
TypedQuery<Tuple> query = em.createQuery(cq);
```

조회

이번에는 SELECT 절을 만드는 `select()` 에 대해서 알아보자.

```
public interface CriteriaQuery<T> extends AbstractQuery<T> {

        CriteriaQuery<T> select(Selection<? extends T> selection); // 한 건 지정
        CriteriaQuery<T> multiselect(Selection<?>... selections);  // 여러 건 지정
        CriteriaQuery<T> multiselect(List<Selection<?>> selectionList); // 여러 건
        ...
}
```

- 조회 대상을 한 건, 여러 건 지정

`select` 에 조회 대상을 하나만 지정하려면 다음처럼 작성하면 되고

```
cq.select(m) //JPQL : select m
```

조회 대상을 여러 건을 지정하려면 `multiselect` 를 사용하면 된다.

```
//JPQL : select m.username, m.age
cq.multiselect(m.get("username"), m.get("age"));
```

여러 건 지정은 다음처럼 `cb.array` 를 사용해도 된다.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
//JPQL : select m.username, m.age
cq.select( cb.array(m.get("username"), m.get("age")) );
```

- DISTINCT

`distinct`는 `select`, `multiselect` 다음에 `distinct(true)`를 사용하면 된다.

```
//JPQL : select distinct m.username, m.age
cq.multiselect(m.get("username"), m.get("age")).distinct(true);
```

완성된 코드를 보자.

====== 완성된 코드 ======

```
// JPQL : select distinct m.username, m.age from Member m

CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);
Root<Member> m = cq.from(Member.class);
cq.multiselect(m.get("username"), m.get("age")).distinct(true);
//cq.select(cb.array(m.get("username"), m.get("age"))).distinct(true); //위 코드와

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

- NEW, `construct()`

JPQL에서 `select new 생성자()` 구문을 Criteria에서는 `cb.construct(클래스 타입, ... )`로 사용한다.

```
<Y> CompoundSelection<Y> construct(Class<Y> resultClass, Selection<?>... selecti
```

실제 사용하는 코드를 보면 쉽게 이해될 것이다.

====== Criteria `construct()` ======

```
//JPQL: select new jpabook.domain.MemberDto(m.username, m.age) from Member m

CriteriaQuery<MemberDto> cq = cb.createQuery(MemberDto.class);
Root<Member> m = cq.from(Member.class);

cq.select(cb.construct(MemberDto.class, m.get("username"), m.get("age"))); //**
```

```
TypedQuery<MemberDto> query = em.createQuery(cq);
List<MemberDto> resultList = query.getResultList();
```

JPQL에서는 `select new jpabook.domain.MemberDto()` 처럼 패키지명을 다 적어 주었다. 하지만 Criteria는 코드를 직접 다루므로 `MemberDto.class` 처럼 간략하게 사용할 수 있다.

- 튜플

Criteria는 `Map` 과 비슷한 튜플이라는 특별한 반환 객체를 제공한다. 코드부터 보자.

====== 튜플 ======

```
// JPQL : select m.username, m.age from Member m

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Tuple> cq = cb.createTupleQuery();
//CriteriaQuery<Tuple> cq = cb.createQuery(Tuple.class); // 위와 같다

Root<Member> m = cq.from(Member.class);
cq.multiselect(
        m.get("username").alias("username"), // 1. 튜플에서 사용할 튜플 별칭
        m.get("age").alias("age")
);

TypedQuery<Tuple> query = em.createQuery(cq);
List<Tuple> resultList = query.getResultList();
for (Tuple tuple : resultList) {
    String username = tuple.get("username", String.class); // 2. 튜플 별칭으로 조회
    Integer age = tuple.get("age", Integer.class);
}
```

튜플을 사용하려면 `cb.createTupleQuery()` 또는 `cb.createQuery(Tuple.class)` 로 Criteria를 생성한다.

1. 튜플은 튜플의 검색 키로 사용할 튜플 전용 별칭을 **필수**로 할당해야 한다. 별칭은 `alias()` 메서드를 사용해서 지정할 수 있다.
2. 선언해둔 튜플 별칭으로 데이터를 조회할 수 있다.

튜플은 이름 기반이므로 순서 기반의 `Object[]` 보다 안전하다. 그리고 `tuple.getElements()` 같은 메서드를 사용해서 현재 튜플의 별칭과 자바 타입도 조회할 수 있다.

> **참고:** 튜플에 별칭을 준다고 해서 실제 SQL에 별칭이 달리는 것은 아니다. 튜플은 `Map`과 비슷한 구조여서 별칭을 키로 사용한다.

튜플은 물론 엔티티도 조회할 수 있다. 튜플을 사용할 때는 별칭을 필수로 주어야 하는 것에 주의하자.

====== 튜플과 엔티티 조회 ======

```
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<Member> m = cq.from(Member.class);
cq.select(cb.tuple(
        m.alias("m"), //회원 엔티티, 별칭 m
        m.get("username").alias("username") //단순 값 조회, 별칭 username
));

TypedQuery<Tuple> query = em.createQuery(cq);
List<Tuple> resultList = query.getResultList();
for (Tuple tuple : resultList) {
    Member member = tuple.get("m", Member.class);
    String username = tuple.get("username", String.class);
}
```

예제에서 `cq.multiselect(...)` 대신에 `cq.select(cb.tuple(...))`를 사용했는데 둘은 같은 기능을 한다.

집합

- GROUP BY

팀 이름별로 나이가 가장 많은 사람과 가장 적은 사람을 구해보자.

====== 집합 ======

```
/*
    JPQL:
    select m.team.name, max(m.age), min(m.age)
    from Member m
    group by m.team.name
 */

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);
```

```
Root<Member> m = cq.from(Member.class);

Expression maxAge = cb.max(m.<Integer>get("age"));
Expression minAge = cb.min(m.<Integer>get("age"));

cq.multiselect(m.get("team").get("name"), maxAge, minAge);
cq.groupBy(m.get("team").get("name")); //** //GROUP BY

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

`cq.groupBy(m.get("team").get("name"))` 은 JPQL에서 `group by m.team.name` 과 같다.

- HAVING

이 조건에 팀에 가장 나이 어린 사람이 10살을 초과하는 팀을 조회한다는 조건을 추가해보자.

```
cq.multiselect(m.get("team").get("name"), maxAge, minAge)
        .groupBy(m.get("team").get("name"))
        .having(cb.gt(minAge, 10)); //HAVING
```

`having(cb.gt(minAge, 10))` 은 JPQL에서 `having min(m.age) > 10` 과 같다.

정렬

정렬 조건도 Criteria 빌더를 통해서 생성한다.

`cb.desc(...)` 또는 `cb.asc(...)` 로 생성할 수 있다.

```
cq.select(m)
    .where(ageGt)
    .orderBy(cb.desc(m.get("age"))); //JPQL: order by m.age desc
```

정렬 API는 다음과 같이 정의되어 있다.

```
CriteriaQuery<T> orderBy(Order... o);
CriteriaQuery<T> orderBy(List<Order> o);
```

조인

조인은 `join()` 메서드와 `JoinType` 클래스를 사용한다.

```
public enum JoinType {

    INNER, //내부 조인
    LEFT, //왼쪽 외부 조인
    RIGHT //오른쪽 외부 조인, JPA 구현체나 데이터베이스에 따라 지원하지 않을 수도 있다.
}
```

예제를 보자.

====== JOIN ======

```
/* JPQL
    select m,t from Member m
    inner join m.team t
    where t.name = '팀A'
/*

Root<Member> m = cq.from(Member.class);
Join<Member, Team> t = m.join("team", JoinType.INNER); //** //내부 조인

cq.multiselect(m, t)
  .where(cb.equal(t.get("name"), "팀A"));
```

쿼리 루트(`m`)에서 바로 `m.join("team")` 메서드를 사용해서 회원과 팀을 조인했다. 그리고 조인한 `team` 에 `t` 라는 별칭을 주었다. 여기서는 `JoinType.INNER` 을 설정해서 내부 조인을 사용했다. 참고로 조인 타입을 생략하면 내부 조인을 사용한다. 외부 조인은 `JoinType.LEFT` 로 설정하면 된다.

```
m.join("team") //내부 조인
m.join("team", JoinType.INNER) //내부 조인
m.join("team", JoinType.LEFT) //외부 조인
```

FETCH JOIN

```
Root<Member> m = cq.from(Member.class);
m.fetch("team", JoinType.LEFT); //**
```

```
cq.select(m);
```

페치 조인은 `fetch(조인대상, JoinType)`을 사용한다. 페치 조인시 주의사항은 JPQL과 같다.

서브 쿼리

Criteria로 작성하는 서브 쿼리에 대해 알아보자.

간단한 서브 쿼리

우선 메인 쿼리와 서브 쿼리간에 관련이 없는 단순한 서브 쿼리부터 시작해보자. 다음은 평균 나이 이상의 회원을 구하는 서브 쿼리다.

====== 간단한 서브 쿼리 ======

```
/* JPQL:
    select m from Member m
    where m.age >=
        (select AVG(m2.age) from Member m2)
*/

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Member> mainQuery = cb.createQuery(Member.class);

//1. 서브 쿼리 생성
Subquery<Double> subQuery = mainQuery.subquery(Double.class);
Root<Member> m2 = subQuery.from(Member.class);
subQuery.select(cb.avg(m2.<Integer>get("age")));

//2. 메인 쿼리 생성
Root<Member> m = mainQuery.from(Member.class);
mainQuery.select(m)
  .where(cb.ge(m.<Integer>get("age"), subQuery));
```

1. 서브 쿼리 생성 부분을 보면 서브 쿼리는 `mainQuery.subquery(...)`로 생성한다.
2. 메인 쿼리 생성 부분을 보면 `where(..., subQuery)`에서 생성한 서브 쿼리를 사용한다.

상호 관련 서브 쿼리

이번에는 메인 쿼리와 서브 쿼리 간에 서로 관련이 있을 때 Criteria를 어떻게 작성하는지 알아보자.

서브 쿼리에서 메인 쿼리의 정보를 사용하려면 메인 쿼리에서 사용한 별칭을 얻어야 한다. 서브 쿼리는 메인 쿼리의 `Root` 나 `Join` 을 통해 생성된 별칭을 받아서 다음과 같이 사용할 수 있다.

```
.where(cb.equal(subM.get("username"), m.get("username")));
```

예제는 팀A에 소속된 회원을 찾도록 했다. 물론 이때는 서브 쿼리보다는 조인으로 해결하는 것이 더 효과적일 수 있다. 여기서는 상호 관련 서브 쿼리에 초점을 맞추자.

====== 상호 관련 서브 쿼리 ======

```
/* JPQL
    select m from Member m
    where exists
        (select t from m.team t where t.name='팀A')
*/
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Member> mainQuery = cb.createQuery(Member.class);

//서브 쿼리에서 사용되는 메인 쿼리의 m
Root<Member> m = mainQuery.from(Member.class);

//서브 쿼리 생성
Subquery<Team> subQuery = mainQuery.subquery(Team.class);
Root<Member> subM = subQuery.correlate(m); //** //메인 쿼리의 별칭을 가져옴
Join<Member, Team> t = subM.join("team");
subQuery.select(t)
        .where(cb.equal(t.get("name"), "팀A"));

//메인 쿼리 생성
mainQuery.select(m)
  .where(cb.exists(subQuery));

List<Member> resultList = em.createQuery(mainQuery).getResultList();
```

여기서 가장 핵심은 `subQuery.correlate(m)` 다. `correlate(...)` 메서드를 사용하면 메인 쿼리의 별칭을 서브 쿼리에서 사용할 수 있다.

IN 식

IN 식은 Criteria 빌더에서 `in(...)` 메서드를 사용한다. 예제로 알아보자.

====== IN 식 ======

```
/* JPQL
    select m from Member m
    where m.username in ("회원1", "회원2")
*/

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Member> cq = cb.createQuery(Member.class);
Root<Member> m = cq.from(Member.class);

cq.select(m)
  .where(cb.in(m.get("username")) //**
          .value("회원1")
          .value("회원2"));
```

CASE 식

CASE 식에는 `selectCase()` 메서드와 `when()`, `otherwise()` 메서드를 사용한다. 예제로 알아보자.

====== CASE 식 ======

```
/* JPQL
    select m.username,
        case when m.age>=60 then 600
             when m.age<=15 then 500
             else 1000
        end
    from Member m
*/

Root<Member> m = cq.from(Member.class);

cq.multiselect(
        m.get("username"),
        cb.selectCase()
                .when(cb.ge(m.<Integer>get("age"), 60), 600)
                .when(cb.le(m.<Integer>get("age"), 15), 500)
                .otherwise(1000)
);
```

파라미터 정의

JPQL에서 `:PARAM1` 처럼 파라미터를 정의했듯이 Criteria도 파라미터를 정의할 수 있다.

====== 파라미터 정의 ======

```
/* JPQL
    select m from Member m
    where m.username = :usernameParam
*/

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Member> cq = cb.createQuery(Member.class);

Root<Member> m = cq.from(Member.class);

//** 1. 정의
cq.select(m)
    .where(cb.equal(m.get("username"), cb.parameter(String.class, "usernameParam

List<Member> resultList = em.createQuery(cq)
        .setParameter("usernameParam", "회원1") //** 2. 바인딩
        .getResultList();
```

순서대로 알아보자.

1. `cb.parameter(타입, 파라미터 이름)` 메서드를 사용해서 파라미터를 정의했다.
2. `setParameter("usernameParam", "회원1")` 를 사용해서 해당 파라미터에 사용할 값을 바인딩 했다.

> **참고:** 하이버네이트는 다음처럼 Criteria에서 파라미터를 정의하지 않고 직접 값을 입력해도 실제 SQL에서는 PreparedStatement에 파라미터 바인딩을 사용한다.
>
> ```
> cq.select(m)
>     .where(cb.equal(m.get("username"), "회원1"));
> ```
>
> 다음 실행된 SQL을 보면 ? 로 파리미터 바인딩을 받고 있다.
>
> ```
> select * from Member m where m.name=?
> ```

네이티브 함수 호출

네이티브 SQL 함수를 호출하려면 `cb.function(...)` 메서드를 사용하면 된다.

```
Root<Member> m = cq.from(Member.class);
Expression<Long> function = cb.function("SUM", Long.class, m.get("age"));
cq.select(function);
```

여기서는 전체 회원의 나이 합을 구했다. “SUM” 대신에 원하는 네이티브 SQL 함수를 입력하면 된다.

> **참고:** JPQL에서 설명했듯이 하이버네이트 구현체는 방언에 사용자정의 SQL 함수를 등록해야 호출할 수 있다.

동적 쿼리

다양한 검색 조건에 따라 실행 시점에 쿼리를 생성하는 것을 동적 쿼리라 한다. 동적 쿼리는 문자 기반인 JPQL보다는 코드 기반인 Criteria로 작성하는 것이 더 편리하다. 우선 JPQL로 만든 동적 쿼리부터 보자. 다음 예제는 나이, 이름, 팀명을 검색 조건으로 사용해서 동적으로 쿼리를 생성한다.

====== JPQL 동적 쿼리 ======

```
//검색 조건
Integer age = 10;
String username = null;
String teamName = "팀A";

//JPQL 동적 쿼리 생성
StringBuilder jpql = new StringBuilder("select m from Member m join m.team t ");
List<String> criteria = new ArrayList<String>();

if (age != null) criteria.add(" m.age = :age ");
if (username != null) criteria.add(" m.username = :username ");
if (teamName != null) criteria.add(" t.name = :teamName ");

if (criteria.size() > 0) jpql.append(" where ");

for (int i = 0; i < criteria.size(); i++) {
```

```
        if (i > 0) jpql.append(" and ");
        jpql.append(criteria.get(i));
    }

    TypedQuery<Member> query = em.createQuery(jpql.toString(), Member.class);
    if (age != null) query.setParameter("age", age);
    if (username != null) query.setParameter("username", username);
    if (teamName != null) query.setParameter("teamName", teamName);

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

JPQL로 동적 쿼리를 구성하는 것은 아슬아슬한 줄타기 같다. 이처럼 단순한 동적 쿼리 코드를 개발해도 문자 더하기로 인해, 여러 번 버그를 만날 것이다. 특히 문자 사이에 공백을 입력하지 않아서 `age=:ageandusername=:username` 처럼 되는 것은 다들 한 번씩은 겪는 문제고 `where` 와 `and` 의 위치를 구성하는 것도 신경을 써야 한다.

이번에는 Criteria로 구성한 동적 쿼리를 보자.

====== Criteria 동적 쿼리 ======

```
    //검색 조건
    Integer age = 10;
    String username = null;
    String teamName = "팀A";

    //Criteria 동적 쿼리 생성
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Member> cq = cb.createQuery(Member.class);

    Root<Member> m = cq.from(Member.class);
    Join<Member, Team> t = m.join("team");

    List<Predicate> criteria = new ArrayList<Predicate>();

    if (age != null) criteria.add(cb.equal(m.<Integer>get("age"), cb.parameter(Integ
    if (username != null) criteria.add(cb.equal(m.get("username"), cb.parameter(Stri
    if (teamName != null) criteria.add(cb.equal(t.get("name"), cb.parameter(String.c

    cq.where(cb.and(criteria.toArray(new Predicate[0])));

    TypedQuery<Member> query = em.createQuery(cq);
    if (age != null) query.setParameter("age", age);
    if (username != null) query.setParameter("username", username);
    if (teamName != null) query.setParameter("teamName", teamName);

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

Criteria로 동적 쿼리를 구성하면 최소한 공백이나 `where`, `and`의 위치로 인해 에러가 발생하지는 않는다. 이런 장점에도 불구하고 Criteria의 장황하고 복잡함으로 인해, 코드가 읽기 힘들다는 단점은 여전히 남아있다.

함수 정리

Criteria는 JPQL 빌더 역할을 하므로 JPQL 함수를 코드로 지원한다.

먼저 살펴볼 함수는 `Expression`에 정의되어 있다. (예: `m.get("username")`의 반환 타입)

Expression 메서드

함수명	JPQL
isNull()	IS NULL
isNotNull()	IS NOT NULL
in()	IN

예) `m.get("username").isNull()`

JPQL에서 사용하는 함수는 대부분 `CriteriaBuilder`에 정의되어 있다. 다음 분류로 나누어 알아보자.

- 조건 함수
- 스칼라와 기타 함수
- 집합 함수
- 분기 함수

조건 함수

함수명	JPQL
and()	and
or()	or
not()	not
equal(), notEqual()	=, <>
lt(), lessThan()	<
le(), LessThanOrEqualTo()	<=
gt(), greaterThan()	>
ge(), greaterThanOrEqualTo()	>=
between()	between
like(), notLike()	like, not like
isTrue(), isFalse	is true, is false
in(), not(in())	in, not(in())
exists(), not(exists())	exists, not exists
isNull(), isNotNull()	is null, is not null
isEmpty(), isNotEmpty()	is empty, is not empty
isMember(), isNotMember()	member of, not member of

스칼라와 기타 함수

함수명	JPQL
sum()	+
neg(), diff()	-
prod()	*

quot()	/
all()	all
any()	any
some()	some
abs()	abs
sqrt()	sqrt
mod()	mod
size()	size
length()	length
locate()	locate
concat()	concat
upper()	upper
lower()	lower
substring()	substring
trim()	trim
currentDate()	current_date
currentTime()	current_time
currentTimestamp()	current_timestamp

집합 함수

함수명	JPQL
avg()	avg
max(), greatest()	max
min(), least()	min
sum(), sumAsLong(), sumAsDouble()	sum
count()	count
countDistinct()	count distinct

분기 함수

함수명	JPQL
nullif()	nullif
coalesce()	coalesce
selectCase()	case

Criteria 메타 모델 API

Criteria는 코드 기반이므로 컴파일 시점에 오류를 발견할 수 있다. 하지만 `m.get("age")`에서 `age`는 문자다. 'age' 대신에 실수로 'ageaaa' 이렇게 잘못 적어도 컴파일 시점에 에러를 발견하지 못한다. 따라서 완전한 코드 기반이라 할 수 없다. 이런 부분까지 코드로 작성하려면 메타 모델 API를 사용하면 된다. 메타 모델 API를 사용하려면 먼저 메타 모델 클래스를 만들어야 한다.

메타 모델 API를 적용한 간단한 예를 보자.

====== 메타 모델 API 적용 전 ======

```
cq.select(m)
        .where(cb.gt(m.<Integer>get("username"), 20))
        .orderBy(cb.desc(m.get("age")));
```

====== 메타 모델 API 적용 후 ======

```
cq.select(m)
        .where(cb.gt(m.get(Member_.age), 20))
        .orderBy(cb.desc(m.get(Member_.age)));
```

메타 모델 적용 전 후를 비교해보자. `m.<Integer>get("age")`를 보면 문자 기반에서 `m.get(Member_.age)` 처럼 정적인 코드 기반으로 변경된 것을 확인할 수 있다. 이렇게 하려면 `Member_` 클래스가 필요한데 이것을 메타 모델 클래스라 한다.

`Member_` 클래스를 살펴보자.

====== `Member_` 클래스 ======

```
@Generated(value = "org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
@StaticMetamodel(Member.class)
public abstract class Member_ {

    public static volatile SingularAttribute<Member, Long> id;
    public static volatile SingularAttribute<Member, String> username;
    public static volatile SingularAttribute<Member, Integer> age;
    public static volatile ListAttribute<Member, Order> orders;
    public static volatile SingularAttribute<Member, Team> team;

}
```

이런 클래스를 표준(CANONICAL) 메타 모델 클래스라 하는데 줄여서 메타 모델이라 한다.

`Member_` 메타 모델 클래스는 `Member` 엔티티를 기반으로 만들어야 한다. 물론 이렇게 복잡한 코드를 개발자가 직접 작성하지는 않는다. 대신에 코드 자동 생성기가 엔티티 클래스를 기반으로 메타 모델 클래스들을 만들어 준다.

하이버네이트 구현체를 사용하면 코드 생성기는 `org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor`를 사용하면 된다.

코드 생성기는 모든 엔티티 클래스를 찾아서 “엔티티명_(언더라인).java” 모양의 메타 모델 클래스를 생성해준다.

```
엔티티 -> 코드 자동 생성기 -> 메타 모델 클래스
```

```
src/jpabook/domain/Member.java //원본 코드
target/generated-sources/annotations/jpabook/domain/Member_.java //자동 생성된 메타
```

코드 생성기 설정

코드 생성기는 보통 메이븐이나 엔트, 그래들(Gradle)을 같은 빌드 도구를 사용해서 실행한다. 여기서는 메이븐을 기준으로 설명하겠다.

메이븐에 다음 두 설정을 추가한다.

====== 코드 생성기 MAVEN 설정 ======

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-jpamodelgen</artifactId>
        <version>1.3.0.Final</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.6</source>
                <target>1.6</target>
                <compilerArguments>
                    <processor>org.hibernate.jpamodelgen.JPAMetaModelEntityProce
                </compilerArguments>
            </configuration>
        </plugin>
    </plugins>
</build>
```

그리고 `mvn compile` 명령어를 사용하면 `target/generated-sources/annotations/` 하위에 메타 모델 클래스들이 생성된다. 이클립스나 인텔리J 같은 IDE를 사용하면 조금 더 편리하게 메타 모델이 생성되도록 할 수 있다.

코드를 자동으로 생성하는 것이 약간 어색할 수 있고 설정도 생각보다 쉽지는 않다. 하지만 코드 기반이므로 IDE의 도움을 받을 수 있고 컴파일 시점에 에러를 잡을 수 있다는 장점이 있으므로 Criteria를 자주 사용한다면 적용하는 것을 권장한다.

> **참고**: 하이버네이트가 제공하는 JPA 메타모델 생성기는 다음 URL을 참고하자. 메이븐, 엔트, 이클립스, 인텔리J 각각에 대해 메타 모델 API 생성기를 설정하는 방법이 상세히 나와 있다. 참고로 메타 모델 API를 사용하기 위해선 JAVA 1.6 이상을 사용해야 한다.
>
> http://docs.jboss.org/hibernate/jpamodelgen/1.3/reference/en-US/html_single

---

> **참고:** JPA Criteria는 너무 장황하고 복잡하다. 반면에 비슷한 역할을 하는 QueryDSL은 코드 기반이라는 장점을 가지고 있으면서 단순하다. 또한 결과 코드가 JPQL과 비슷해서 직관적으로 이해할 수 있다. 이런 이유로 나는 Criteria보다 QueryDSL을 선호한다. QueryDSL은 다음 장에서 다룬다.
