- **객체 지향 쿼리 - 소개**
  - **JPQL (Java Persistence Query Language)**
  - **Criteria 쿼리**
  - **QueryDSL**
  - **네이티브 SQL**
  - **JDBC 직접 사용, MyBatis 같은 SQL 매퍼 프레임워크 사용**

객체 지향 쿼리 - 소개

`EntityManager.find()` 메서드를 사용하면 식별자로 엔티티 하나를 조회할 수 있다. 이렇게 조회한 엔티티에 객체 그래프 탐색을 사용하면 연관된 엔티티들을 찾을 수 있다. 이 둘은 가장 단순한 검색 방법이다.

- 식별자로 조회 `EntityManager.find()`
- 객체 그래프 탐색 (ex: `a.getB().getC()`)

이 기능만으로 애플리케이션을 개발하기는 어렵다. 예를 들어 나이가 30살 이상인 회원을 모두 검색하고 싶다면 좀 더 현실적이고 복잡한 검색 방법이 필요하다. 그렇다고 모든 회원 엔티티를 메모리에 올려두고 애플리케이션에서 30살 이상인 회원을 검색하는 것은 현실성이 없다. 결국 데이터는 데이터베이스에 있으므로 SQL로 필요한 내용을 최대한 걸러서 조회해야 한다. 하지만 ORM을 사용하면 데이터베이스 테이블이 아닌 엔티티 객체를 대상으로 개발하므로 검색도 테이블이 아닌 엔티티 객체를 대상으로 하는 방법이 필요하다.

JPQL은 이런 문제를 해결하기 위해 만들어졌는데 다음과 같은 특징이 있다.

- 테이블이 아닌 객체를 대상으로 검색하는 객체 지향 쿼리다.
- SQL을 추상화해서 특정 데이터베이스 SQL에 의존하지 않는다.

SQL이 데이터베이스 테이블을 대상으로 하는 데이터 중심의 쿼리라면 JPQL은 엔티티 객체를 대상으로 하는 객체 지향 쿼리다. JPQL을 사용하면 JPA는 이 JPQL을 분석한 다음 적절한 SQL을 만들어 데이터베이스를 조회한다. 그리고 조회한 결과로 엔티티 객체를 생성해서 반환한다.

JPQL을 한마디로 정의하면 객체 지향 SQL이다. 처음 보면 SQL로 오해할 정도로 문법이 비슷하다. 따라서 SQL에 익숙한 개발자는 몇 가지 차이점만 이해하면 쉽게 적응할 수 있다.

JPA는 JPQL뿐만 아니라 다양한 검색 방법을 제공한다.

다음은 JPA가 공식 지원하는 기능이다.

- **JPQL (Java Persistence Query Language)**
- **Criteria 쿼리 (Criteria Query):** JPQL을 편하게 작성하도록 도와주는 API, 빌더 클래스 모음
- **네이티브 SQL (Native SQL):** JPA에서 JPQL 대신 직접 SQL을 사용할 수 있다.

다음은 JPA가 공식 지원하는 기능은 아니지만 알아둘 가치가 있다.

- **QueryDSL:** Criteria 쿼리처럼 JPQL을 편하게 작성하도록 도와주는 빌더 클래스 모음, 비표준 오픈소스 프레임워크다.
- **JDBC 직접 사용, MyBatis 같은 SQL 매퍼 프레임워크 사용:** 필요하면 JDBC를 직접 사용할 수 있다.

가장 중요한 건 JPQL이다. Criteria나 QueryDSL은 JPQL을 편하게 작성하도록 도와주는 빌더 클래스일 뿐이다. 따라서 JPQL을 이해해야 나머지도 이해할 수 있다.

이 장에서는 전체적인 감을 잡기 위해 하나하나 아주 간단히 살펴보자.

JPQL (Java Persistence Query Language)

JPQL은 엔티티 객체를 조회하는 객체지향 쿼리다. 문법은 SQL과 비슷하고 ANSI 표준 SQL이 제공하는 기능을 유사하게 지원한다.

JPQL은 SQL을 추상화해서 특정 데이터베이스에 의존하지 않는다. 그리고 데이터베이스 방언(Dialect)만 변경하면 JPQL을 수정하지 않아도 자연스럽게 데이터베이스를 변경할 수 있다. 예를 들어 같은 SQL 함수라도 데이터베이스마다 사용 문법이 다른 것이 있는데, JPQL이 제공하는 표준화된 함수를 사용하면 선택한 방언에 따라 해당 데이터베이스에 맞춘 적절한 SQL 함수가 실행된다.

JPQL은 SQL보다 간결하다. 엔티티 직접 조회, 묵시적 조인, 다형성 지원으로 SQL보다 코드가 간결하다.

JPQL을 사용하는 간단한 예제를 보자.

====== 회원 엔티티 ======

```
@Entity(name="Member") //name 속성의 기본값은 클래스명
public class Member {

    @Column(name = "name")
    private String username;
    //...
}
```

====== JPQL 사용 ======

```
//쿼리 생성
String jpql = "select m from Member as m where m.username = 'kim'";
List<Member> resultList = em.createQuery(jpql, Member.class).getResultList();
```

이 코드는 회원이름이 `kim`인 엔티티를 조회한다. JPQL에서 `Member`는 엔티티 이름이다. 그리고 `m.username`은 테이블 컬럼명이 아니라 엔티티 객체의 필드명이다.

`em.createQuery()` 메서드에 실행할 JPQL과 반환할 엔티티의 클래스 타입인 `Member.class`를 넘겨주고 `getResultList()` 메서드를 실행하면 JPA는 JPQL을 SQL로 변환해서 데이터베이스를 조회한다. 그리고 조회한 결과로 `Member` 엔티티를 생성해서 반환한다.

====== 실행한 JPQL ======

```
select m
from Member as m
where m.username = 'kim'
```

====== 실제 실행된 SQL ======

```
select
    member.id as id,
    member.age as age,
    member.team_id as team,
    member.name as name
from
    Member member
where
    member.name='kim'
```

참고로 하이버네이트 구현체가 생성한 SQL은 별칭이 너무 복잡해서 알아보기 쉽게 수정했다.

Criteria 쿼리

Criteria는 JPQL을 생성하는 빌더 클래스다. **Criteria의 장점은 문자가 아닌 `query.select(m).where(...)` 처럼 프로그래밍 코드로 JPQL을 작성할 수 있다는 점이다.**

예를 들어 JPQL에서 `select m from Membeeee m`처럼 오타가 있다고 가정해보자. 그래도 컴파일은 성공하고 애플리케이션을 서버에 배포할 수 있다. 문제는 해당 쿼리가 실행되는 런타임 시점에 오류가 발생하다는 점이다. 이것이 문자기반 쿼리의 단점이다. 반면에 Criteria는 문자가 아닌 코드로 JPQL을 작성한다. 따라서 컴파일 시점에 오류를 발견할 수 있다.

문자로 작성한 JPQL보다 코드로 작성한 Criteria의 장점

- 컴파일 시점에 오류를 발견할 수 있다.
- IDE를 사용하면 코드 자동완성을 지원한다.
- 동적 쿼리를 작성하기 편하다.

하이버네이트를 포함한 몇몇 ORM 프레임워크들은 이미 오래전부터 자신만의 Criteria를 지원했다. JPA는 2.0부터 Criteria를 지원한다.

간단한 Criteria 사용 코드를 보자. 방금 보았던 JPQL을 Criteria로 작성해보자.

```
select m from Member as m where m.username = 'kim'"
```

====== Criteria 쿼리 ======

```
//Criteria 사용 준비
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Member> query = cb.createQuery(Member.class);
Root<Member> m = query.from(Member.class); //루트 클래스 (조회를 시작할 클래스)

//쿼리 생성
CriteriaQuery<Member> cq = query.select(m).where(cb.equal(m.get("username"), "ki
List<Member> resultList = em.createQuery(cq).getResultList();
```

쿼리를 문자가 아닌 코드로 작성한 것을 확인할 수 있다. 아쉬운 점은 `m.get("username")`를 보면 필드명을 문자로 작성했다. 만약 이 부분도 문자가 아닌 코드로 작성하고 싶으면 메타 모델(MetaModel)을 사용하면 된다.

[메타 모델 API]

자바가 제공하는 어노테이션 프로세서(Annotation Processor) 기능을 사용하면 어노테이션을 분석해서 클래스를 생성할 수 있다. JPA는 이 기능을 사용해서 `Member` 엔티티 클래스로부터 `Member_`라는 Criteria 전용 클래스를 생성하는데 이것을 메타 모델이라 한다. 어노테이션 프로세서 기능을 사용해서 메타 모델을 생성하는 자세한 방법은 Criteria 장에서 알아보겠다.

메타 모델을 사용하면 온전히 코드만 사용해서 쿼리를 작성할 수 있다.

메타 모델 사용 전 -> 사용 후

```
m.get("username") -> m.get(Member_.username)
```

이 코드를 보면 “username”이라는 문자에서 `Member_.username`이라는 코드로 변경된 것을 확인할 수 있다.

참고로 Criteria는 코드로 쿼리를 작성할 수 있어서 동적 쿼리를 작성할 때 유용하다.

Criteria가 가진 장점이 많지만 모든 장점을 상쇄할 정도로 복잡하고 장황하다. 따라서 사용하기 불편한 건 물론이고 Criteria로 작성한 코드도 한눈에 들어오지 않는다는 단점이 있다.

QueryDSL

QueryDSL도 Criteria처럼 JPQL 빌더 역할을 한다. QueryDSL의 장점은 코드 기반이면서 단순하고 사용하기 쉽다. 그리고 작성한 코드도 JPQL과 비슷해서 한눈에 들어온다. QueryDSL과 Criteria를 비교하면 Criteria는 너무 복잡하다.

> **참고:** QueryDSL은 JPA 표준은 아니고 오픈소스 프로젝트다. 이것은 JPA뿐만 아니라 JDO, MONGODB, Java Collection, Lucene, Hibernate Search도 거의 같은 문법으로 지원한다. 현재 스프링 데이터 프로젝트가 지원할 정도로 많이 기대되는 프로젝트다. 나는 Criteria보다 QueryDSL을 선호한다.

QueryDSL로 작성한 코드를 보자.

====== QueryDSL ======

```
//준비
JPAQuery query = new JPAQuery(em);
QMember member = QMember.member;

//쿼리, 결과조회
List<Member> members =
    query.from(member)
    .where(member.username.eq("kim"))
    .list(member);
```

특별한 설명을 하지 않아도 코드만으로 대부분 이해가 될 것이다.

QueryDSL도 어노테이션 프로세서를 사용해서 쿼리 전용 클래스를 만들어야 한다. `QMember`는 `Member` 엔티티 클래스를 기반으로 생성한 QueryDSL 쿼리 전용 클래스다.

네이티브 SQL

JPA는 SQL을 직접 사용할 수 있는 기능을 지원하는데 이것을 네이티브 SQL이라 한다.

JPQL을 사용해도 가끔은 특정 데이터베이스에 의존하는 기능을 사용해야 할 때가 있다. 예를 들어 Oracle 데이터베이스만 사용하는 `CONNECT BY` 기능이나 특정 데이터베이스에서만 동작하는 SQL 힌트를 같은 것이다. 이런 기능들은 전혀 표준화되어 있지 않으므로 JPQL에서 사용할 수 없다. 그리고 SQL은 지원하지만 JPQL이 지원하지 않는 기능도 있다. 이때는 네이티브 SQL을 사용하면 된다.

네이티브 SQL의 단점은 특정 데이터베이스에 의존하는 SQL을 작성해야 한다는 것이다. 따라서 데이터베이스를 변경하면 네이티브 SQL도 수정해야 한다.

==== 네이티브 SQL ====

```
String sql = "SELECT ID, AGE, TEAM_ID, NAME FROM MEMBER WHERE NAME = 'kim'";
List<Member> resultList = em.createNativeQuery(sql, Member.class).getResultList(
```

네이티브 SQL은 `em.createNativeQuery()`를 사용하면 된다. 나머지는 API는 JPQL과 같다. 실행하면 직접 작성한 SQL을 데이터베이스에 전달한다.

JDBC 직접 사용, MyBatis 같은 SQL 매퍼 프레임워크 사용

이럴 일은 드물겠지만, JDBC 커넥션에 직접 접근하고 싶으면 JPA는 JDBC 커넥션을 획득하는 API를 제공하지 않으므로 JPA 구현체가 제공하는 방법을 사용해야 한다. 하이버네이트에서 직접 JDBC Connection을 획득하는 방법은 다음과 같다.

먼저 JPA `EntityManager`에서 하이버네이트 `Session`을 구한다. 그리고 `Seesion`의 `doWork` 메서드를 호출하면 된다.

====== 하이버네이트 JDBC 획득 ======

```
Session session = entityManager.unwrap(Session.class);
session.doWork(new Work() {

    @Override
    public void execute(Connection connection) throws SQLException {
        //work...
    }
});
```

JDBC나 MyBatis를 JPA를 함께 사용할 때 주의할 점

JDBC를 직접 사용하든 MyBatis 같은 SQL Mapper와 사용하든 모두 JPA를 우회해서 데이터베이스에 접근한다. 문제는 JPA를 우회하는 SQL에 대해서는 JPA가 전혀 인식하지 못한다는 점이다. 최악의 시나리오는 영속성 컨텍스트와 데이터베이스를 불일치 상태로 만들어 데이터 무결성을 훼손할 수 있다. 예를 들어 같은 트랜잭션에서 영속성 컨텍스트에 있는 10,000원 하는 상품 A의 가격을 9,000원으로 변경하고 아직 플러시를 하지 않았는데 JPA를 우회해서 데이터베이스에 직접 상품 A를 조회하면 상품 가격이 얼마겠는가? 데이터베이스에 상품 가격은 아직 10,000원이므로 10,000원이 조회된다.

이런 이슈를 해결하는 방법은 JPA를 우회해서 SQL을 실행하기 직전에 영속성 컨텍스트를 수동으로 플러시해서 데이터베이스와 영속성 컨텍스트를 동기화하면 된다.

참고로 스프링 프레임워크를 사용하면 JPA와 MyBatis를 손쉽게 통합할 수 있다. 또한 스프링 프레임워크의 AOP를 적절히 활용해서 JPA를 우회하여 데이터베이스에 접근하는 메서드를 호출할 때 마다 영속성 컨텍스트를 플러시 하면 위에서 언급한 문제도 깔끔하게 해결할 수 있다.