- **네이티브 SQL**
  - **네이티브 SQL 사용**
    - - 엔티티 조회
    - - 값 조회
    - - 결과 매핑 사용
    - - 결과 매핑 어노테이션
  - **Named 네이티브 SQL**
    - - @NamedNativeQuery
  - **네이티브 SQL XML 에 정의하기**
  - **네이티브 SQL 정리**
  - **스토어드 프로시저(JPA 2.1)**
    - - 스토어드 프로시저 사용하기
    - - Named 스토어드 프로시저 사용하기

네이티브 SQL

JPQL의 한계

JPQL은 표준 SQL이 지원하는 대부분의 문법과 SQL 함수들을 지원하지만, 특정 데이터베이스에 종속적인 기능은 지원하지 않는다. 예를 들어 다음과 같은 것들이다.

- 특정 데이터베이스만 지원하는 함수, 문법, SQL 쿼리 힌트
- 인라인 뷰(From 절에서 사용하는 서브쿼리), UNION, INTERSECT
- 스토어드 프로시저

때로는 특정 데이터베이스에 종속적인 기능이 필요하다. JPA는 특정 데이터베이스에 종속적인 기능을 사용할 수 있는 다양한 방법을 제공한다. 그리고 JPA 구현체들은 JPA 표준보다 더 다양한 방법을 지원한다.

특정 데이터베이스에 종속적인 기능을 지원하는 방법들

- 특정 데이터베이스만 사용하는 함수
  - JPQL에서 네이티브 SQL 함수를 호출할 수 있다. (JPA 2.1)
  - 하이버네이트는 데이터베이스 방언에 각 데이터베이스에 종속적인 함수들을 정의해두었다. 또한 직접 호출할 함수를 정의할 수도 있다.

- 특정 데이터베이스만 지원하는 문법
  - Oracle의 `CONNECT BY` 처럼 특정 데이터베이스에 너무 종속적인 SQL 문법은 지원하지는 않는다.

- 특정 데이터베이스만 지원하는 SQL 쿼리 힌트
  - 하이버네이트를 포함한 몇몇 JPA 구현체들이 지원한다.

- 인라인 뷰(From 절에서 사용하는 서브쿼리), UNION, INTERSECT
  - 하이버네이트는 지원하지 않지만 일부 JPA 구현체들이 지원한다.

- 스토어 프로지져
  - JPQL에서 스토어드 프로시저를 호출할 수 있다. (JPA 2.1)

다양한 이유로 JPQL을 사용할 수 없을 때 JPA는 SQL을 직접 사용할 수 있는 기능을 제공하는데 이것을 네이티브 SQL이라 한다.

JPQL을 사용하면 JPA가 SQL을 생성한다. 네이티브 SQL은 이 SQL을 개발자가 직접 정의하는 것이다. 쉽게 이야기해서 JPQL이 자동 모드라면 네이티브 SQL은 수동 모드다.

그럼 JPA가 지원하는 네이티브 SQL과 JDBC API를 직접 사용하는 것에는 어떤 차이가 있을까? **네이티브 SQL을 사용하면 엔티티를 조회할 수 있고 JPA가 지원하는 영속성 컨텍스트의 기능을 그대로 사용할 수 있다.** 반면에 JDBC API를 직접 사용하면 단순히 데이터의 나열을 조회할 뿐이다.

지금부터 네이티브 SQL을 알아보자.

네이티브 SQL 사용

네이티브 쿼리 API

네이티브 쿼리 API는 다음 3가지가 있다.

```
public Query createNativeQuery(String sqlString, Class resultClass); //결과 타입 정
public Query createNativeQuery(String sqlString); //결과 타입을 정의할 수 없을 때
public Query createNativeQuery(String sqlString, String resultSetMapping); //결과
```

우선 엔티티를 조회하는 것부터 보자.

- 엔티티 조회

====== 엔티티 조회 코드 ======

```
//SQL 정의
String sql =
        "SELECT ID, AGE, NAME, TEAM_ID " +
        "FROM MEMBER WHERE AGE > ?";

Query nativeQuery = em.createNativeQuery(sql, Member.class)
        .setParameter(1, 20);

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

네이티브 SQL은 `em.createNativeQuery(SQL, 결과 클래스)`를 사용한다. 첫 번째 파라미터는 네이티브 SQL을 입력하고 두 번째 파라미터는 조회할 엔티티 클래스의 타입을 입력한다. JPQL를 사용할 때와 거의 비슷하지만 실제 데이터베이스 SQL을 사용한다는 것과 위치기반 파라미터만 지원한다는 차이가 있다.

여기서 가장 중요한 점은 네이티브 SQL로 SQL만 직접 사용할 뿐이지 나머지는 JPQL을 사용할 때와 같다. 조회한 엔티티도 영속성 컨텍스트에서 관리된다.

> **참고:** JPA는 공식적으로 네이티브 SQL에서 이름 기반 파라미터를 지원하지 않고 위치 기반 파라미터만 지원한다. 하지만 하이버네이트는 네이티브 SQL에 이름 기반 파라미터를 사용할 수 있다. 따라서 하이버네이트 구현체를 사용한다면 예제를 이름 기반 파라미터로 변경해도 동작한다.

> **참고:** `em.createNativeQuery()`를 호출하면서 타입 정보를 주었는데도 `TypeQuery`가 아닌 `Query`가 리턴되는데 이것은 JPA1.0에서 API 규약이 정의되어 버려서 그렇다. 너무 신경쓰지 않아도 된다.

- 값 조회

====== 값 조회 ======

```
//SQL 정의
String sql =
        "SELECT ID, AGE, NAME, TEAM_ID " +
        "FROM MEMBER WHERE AGE > ?";

Query nativeQuery = em.createNativeQuery(sql) //**
        .setParameter(1, 10);

List<Object[]> resultList = nativeQuery.getResultList();
for (Object[] row : resultList) {
    System.out.println("id = " + row[0]);
    System.out.println("age = " + row[1]);
    System.out.println("name = " + row[2]);
    System.out.println("team_id = " + row[3]);
}
```

이번에는 엔티티로 조회하지 않고 단순히 값으로 조회했다. 이렇게 여러 값으로 조회하려면 `em.createNativeQuery(SQL)`의 두번째 파라미터를 사용하지 않으면 된다. JPA는 조회한 값들을 `Object[]`에 담아서 반환한다. 여기서는 스칼라 값들을 조회했을 뿐이므로 결과를 영속성 컨텍스트가 관리하지 않는다. 마치 JDBC로 데이터를 조회한 것과 비슷하다.

- 결과 매핑 사용

지금까지 특정 엔티티를 조회하거나 스칼라 값들을 나열해서 조회하는 단순한 조회 방법을 설명했다.

엔티티와 스칼라 값을 함께 조회하는 것처럼 매핑이 복잡해지면 `@SqlResultSetMapping`를 정의해서 결과 매핑을 사용해야 한다.

이번에는 회원 엔티티와 회원이 주문한 상품 수를 조회해보자.

====== 결과 매핑 사용 ======

```
//SQL 정의
String sql =
        "SELECT M.ID, AGE, NAME, TEAM_ID, I.ORDER_COUNT " +
        "FROM MEMBER M " +
        "LEFT JOIN " +
        "    (SELECT IM.ID, COUNT(*) AS ORDER_COUNT " +
        "     FROM ORDERS O, MEMBER IM " +
        "     WHERE O.MEMBER_ID = IM.ID) I " +
        "ON M.ID = I.ID";

Query nativeQuery = em.createNativeQuery(sql, "memberWithOrderCount"); //**

List<Object[]> resultList = nativeQuery.getResultList();
for (Object[] row : resultList) {
    Member member = (Member) row[0];
    BigInteger orderCount = (BigInteger)row[1];

    System.out.println("member = " + member);
    System.out.println("orderCount = " + orderCount);
}
```

`em.createNativeQuery(sql, "memberWithOrderCount")`의 두번째 파라미터에 결과 매핑 정보의 이름이 사용되었다.

결과 매핑을 정의하는 코드를 보자.

====== 결과 매핑을 정의 ======

```
@Entity
@SqlResultSetMapping(name = "memberWithOrderCount",
        entities = {@EntityResult(entityClass = Member.class)},
        columns = {@ColumnResult(name = "ORDER_COUNT")}
)
public class Member {
```

`memberWithOrderCount`의 결과 매핑을 잘 보면 회원 엔티티와 `ORDER_COUNT` 컬럼을 매핑했다. 위에서 사용한 쿼리 결과중 `ID`, `AGE`, `NAME`, `TEAM_ID`는 `Member` 엔티티와 매핑하고 `ORDER_COUNT`는 단순히 값으로 매핑한다. 그리고 `entities`, `columns`라는 이름에서 알 수 있듯이 여러 엔티티와 여러 컬럼을 매핑할 수 있다.

이번에는 JPA 표준 명세에 있는 예제 코드로 결과 매핑을 어떻게 하는지 좀 더 자세히 알아보자.

====== 표준 명세 예제 - SQL ======

```
Query q = em.createNativeQuery(
        "SELECT o.id AS order_id, " +
            "o.quantity AS order_quantity, " +
            "o.item AS order_item, " +
            "i.name AS item_name, " +
        "FROM Order o, Item i " +
        "WHERE (order_quantity > 25) AND (order_item = i.id)","OrderResults");
```

====== 표준 명세 예제 - 매핑 정보 ======

```
@SqlResultSetMapping(name="OrderResults",
        entities={
          @EntityResult(entityClass=com.acme.Order.class, fields={
              @FieldResult(name="id", column="order_id"),
              @FieldResult(name="quantity", column="order_quantity"),
              @FieldResult(name="item", column="order_item")})},
        columns={
          @ColumnResult(name="item_name")}
)
```

보면 `@FieldResult`를 사용해서 컬럼명과 필드명을 직접 매핑한다. 이 설정은 엔티티의 필드에 정의한 `@Column`보다 앞선다. 조금 불편한 것은 `@FieldResult`를 한 번이라도 사용하면 전체 필드를 `@FieldResult`로 매핑해야 한다.

다음처럼 두 엔티티를 조회하는데 컬럼명이 중복될 때도 `@FieldResult`를 사용해야 한다.

```
SELECT A.ID, B.ID FROM A, B
```

A, B 둘다 ID라는 필드를 가지고 있으므로 컬럼명이 출동한다. 따라서 다음과 같이 별칭을 적절히 사용하고 `@FieldResult`로 매핑하면 된다.

```
SELECT
    A.ID AS A_ID,
    B.ID AS B_ID
FROM A, B
```

- 결과 매핑 어노테이션

결과 매핑에 사용하는 어노테이션을 알아보자.

표 - `@SqlResultSetMapping` 속성

속성	기능
name	결과 매핑 이름
entities	`@EntityResult`를 사용해서 엔티티를 결과로 매핑한다.
columns	`@ColumnResult`를 사용해서 컬럼을 결과로 매핑한다.

표 - `@EntityResult` 속성

속성	기능
entityClass	결과로 사용할 엔티티 클래스를 지정한다.
fields	`@FieldResult`을 사용해서 결과 컬럼을 필드와 매핑한다.
discriminatorColumn	엔티티의 인스턴스 타입을 구분하는 필드 (상속에서 사용됨)

표 - `@FieldResult` 속성

속성	기능
name	결과를 받을 필드명
column	결과 컬럼명

표 - `@ColumnResult` 속성

속성	기능
name	결과 컬럼명

Named 네이티브 SQL

JPQL처럼 네이티브 SQL도 Named 네이티브 SQL을 사용해서 정적 SQL을 작성할 수 있다. 이것을 사용해서 엔티티를 조회해보자.

엔티티 조회

```
@Entity
@NamedNativeQuery(
        name = "Member.memberSQL",
        query = "SELECT ID, AGE, NAME, TEAM_ID " +
                "FROM MEMBER WHERE AGE > ?",
        resultClass = Member.class
)
public class Member {...}
```

`@NamedNativeQuery`로 Named 네이티브 SQL을 등록했다. 다음으로 사용하는 예제를 보자.

```
TypedQuery<Member> nativeQuery = em.createNamedQuery("Member.memberSQL", Member.
        .setParameter(1, 20);
```

흥미로운 점은 JPQL Named 쿼리와 같은 `createNamedQuery` 메서드를 사용한다는 것이다. 따라서 `TypeQuery`를 사용할 수 있다.

다음으로 Named 네이티브 SQL에서 결과 매핑을 사용해보자.

결과 매핑 사용

```
@Entity
@SqlResultSetMapping( name = "memberWithOrderCount",
            entities = {@EntityResult(entityClass = Member.class)},
            columns = {@ColumnResult(name = "ORDER_COUNT")}
)
@NamedNativeQuery(
    name = "Member.memberWithOrderCount",
    query = "SELECT M.ID, AGE , NAME , TEAM_ID, I.ORDER_COUNT " +
            "FROM MEMBER M " +
            "LEFT JOIN " +
            "    (SELECT IM.ID, COUNT(*) AS ORDER_COUNT " +
            "     FROM ORDERS O, MEMBER IM " +
            "     WHERE O.MEMBER_ID = IM.ID) I " +
            "ON M.ID = I.ID",
    resultSetMapping = "memberWithOrderCount" //**
)
public class Member {
```

Named 네이티브 쿼리에서 `resultSetMapping = "memberWithOrderCount"`로 조회 결과를 매핑할 대상까지 지정했다. 다음은 위에서 정의한 Named 네이티브 쿼리를 사용하는 코드다.

```
List<Object[]> resultList = em.createNamedQuery("Member.memberWithOrderCount")
        .getResultList();
```

- @NamedNativeQuery

`@NamedNativeQuery`의 속성을 알아보자.

표 - `@NamedNativeQuery` 속성

속성	기능	기본값
name	네임드 쿼리 이름 (필수)	
query	SQL 쿼리 (필수)	
hints	벤더 종속적인 힌트	
resultClass	결과 클래스	
resultSetMapping	결과 매핑 사용	

각 기능은 이미 설명했으므로 표로 이해가 될 것이다. 여기서 `hints` 속성이 있는데 이것은 SQL 힌트가 아니라 하이버네이트 같은 JPA 구현체에 제공하는 힌트다.

여러 Named 네이티브 쿼리를 선언하려면 다음처럼 사용하면 된다.

```
@NamedNativeQueries({
    @NamedNativeQuery(...),
    @NamedNativeQuery(...)
})
```

네이티브 SQL XML 에 정의하기

Named 네이티브 쿼리를 XML에 정의해보자.

====== ormMember.xml ======

```
<entity-mappings ...>
```

```
    <named-native-query name="Member.memberWithOrderCountXml"
                        result-set-mapping="memberWithOrderCountResultMap" >
        <query><![CDATA[
            SELECT M.ID, AGE, NAME, TEAM_ID, I.ORDER_COUNT
            FROM MEMBER M
            LEFT JOIN
                (SELECT IM.ID, COUNT(*) AS ORDER_COUNT
                 FROM ORDERS O, MEMBER IM
                 WHERE O.MEMBER_ID = IM.ID) I
            ON M.ID = I.ID
        ]]></query>
    </named-native-query>

    <sql-result-set-mapping name="memberWithOrderCountResultMap">
        <entity-result entity-class="jpabook.domain.Member" />
        <column-result name="ORDER_COUNT" />
    </sql-result-set-mapping>

</entity-mappings>
```

XML에 정의할 때는 순서를 지켜야 하는데 `<named-native-query>`를 먼저 정의하고 `<sql-result-set-mapping>`를 정의해야 한다.

XML과 어노테이션 둘 다 사용하는 코드는 같다.

```
List<Object[]> resultList = em.createNamedQuery("Member.memberWithOrderCount")
        .getResultList();
```

> **참고:** 네이티브 SQL은 보통 JPQL로 작성하기 어려운 복잡한 SQL 쿼리를 작성하거나 SQL을 최적화해서 데이터베이스 성능을 향상할 때 사용한다. 이런 쿼리들은 대체로 복잡하고 라인수가 많다. 따라서 어노테이션보다는 XML 사용하는 것이 여러모로 편리하다. 자바는 멀티 라인 문자열을 지원하지 않으므로 라인을 변경할 때마다 문자열을 더해야 하는 큰 불편함이 있다. 반면에 XML을 사용하면 SQL 개발 도구에서 완성한 SQL을 바로 붙여 넣을 수 있어 편리하다.

네이티브 SQL 정리

네이티브 SQL도 JPQL을 사용할 때와 마찬가지로 `Query` , `TypeQuery` (Named 네이티브 쿼리의 경우에만)를 반환한다. 따라서 JPQL API를 그대로 사용할 수 있다. 예를 들어 네이티브 SQL을 사용해도 페이징 처리 API를 적용할 수 있다.

====== 네이티브 SQL과 페이징 처리 ======

```
String sql = "SELECT ID, AGE, NAME, TEAM_ID FROM MEMBER";
Query nativeQuery = em.createNativeQuery(sql, Member.class)
        .setFirstResult(10)
        .setMaxResults(20)
```

데이터베이스 방언에 따라 결과는 다르겠지만, 다음처럼 페이징 정보를 추가한 SQL을 실행한다.

```
SELECT ID, AGE, NAME, TEAM_ID
FROM MEMBER
limit ? offset ? # 페이징 정보 추가
```

네이티브 SQL은 JPQL이 자동 생성하는 SQL을 수동으로 직접 정의하는 것이다. 따라서 JPA가 제공하는 기능 대부분을 그대로 사용할 수 있다.

네이티브 SQL은 관리하기 쉽지 않고 자주 사용하면 특정 데이터베이스에 종속적인 쿼리가 증가해서 이식성이 떨어진다. 그렇다고 현실적으로 네이티브 SQL을 사용하지 않을 수는 없다. 될 수 있으면 표준 JPQL을 사용하고 기능이 부족하면 차선책으로 하이버네이트 같은 JPA 구현체가 제공하는 기능을 사용하자. 그래도 안되면 마지막 방법으로 네이티브 SQL을 사용하자.

그리고 네이티브 SQL로도 부족함을 느낀다면 MyBatis나 스프링 프레임워크가 제공하는 JdbcTemplate 같은 SQLMapper 프레임워크와 JPA를 함께 사용하는 것도 고려할만하다.

스토어드 프로시저(JPA 2.1)

JPA는 2.1부터 스토어드 프로시저를 지원한다.

- 스토어드 프로시저 사용하기

단순히 입력 값을 두 배로 증가시켜 주는 `proc_multiply` 라는 스토어드 프로시저가 있다. 이 프로시져는 첫 번째 파라미터로 값을 입력받고 두 번째 파라미터로 결과를 반환한다.

====== `proc_multiply` MySQL 프로시져 ======

```
DELIMITER //

CREATE PROCEDURE proc_multiply (INOUT inParam INT, INOUT outParam INT)
BEGIN
    SET outParam = inParam * 2;
END //
```

JPA로 이 스토어드 프로시저를 호출해보자.

순서 기반 파라미터 호출

```
StoredProcedureQuery spq = em.createStoredProcedureQuery("proc_multiply");
spq.registerStoredProcedureParameter(1, Integer.class, ParameterMode.IN);
spq.registerStoredProcedureParameter(2, Integer.class, ParameterMode.OUT);

spq.setParameter(1,100);
spq.execute();

Integer out = (Integer)spq.getOutputParameterValue(2);
System.out.println("out = " + out); //결과 = 200
```

스토어드 프로시저를 사용하려면 `em.createStoredProcedureQuery()` 메서드에 사용할 스토어드 프로시저 이름을 입력하면 된다. 그리고 `registerStoredProcedureParameter()` 메서드를 사용해서 프로시저에서 사용할 파라미터를 순서, 타입, 파라미터 모드 순으로 정의하면 된다. 사용할 수 있는 `ParameterMode`는 다음과 같다.

```
public enum ParameterMode {
    IN,         //INPUT 파라미터
    INOUT,      //INPUT, OUTPUT 파라미터
    OUT,        //OUTPUT 파라미터
    REF_CURSOR //CURSOR 파라미터
}
```

다음처럼 파라미터에 순서 대신에 이름을 사용할 수 있다.

```
StoredProcedureQuery spq = em.createStoredProcedureQuery("proc_multiply");
spq.registerStoredProcedureParameter("inParam", Integer.class, ParameterMode.IN)
spq.registerStoredProcedureParameter("outParam", Integer.class, ParameterMode.OU
```

```
spq.setParameter("inParam",100);
spq.execute();

Integer out = (Integer)spq.getOutputParameterValue("outParam");
System.out.println("out = " + out); //결과 = 200
```

- Named 스토어드 프로시저 사용하기

스토어드 프로시저 쿼리에 이름을 부여해서 사용하는 것을 Named 스토어드 프로시저라 한다.

Named 스토어드 프로시저 어노테이션에 정의하기

```
@NamedStoredProcedureQuery(
        name = "multiply",
        procedureName = "proc_multiply",
        parameters = {
            @StoredProcedureParameter(name = "inParam", mode = ParameterMode.IN,
            @StoredProcedureParameter(name = "outParam", mode = ParameterMode.OU
        }
)
@Entity
public class Member { ... }
```

`@NamedStoredProcedureQuery` 로 정의하고 `name` 속성으로 이름을 부여하면 된다. `procedureName` 속성에 실제 호출할 프로시저 이름을 적어주고 `@StoredProcedureParameter`를 사용해서 파라미터 정보를 정의하면 된다. 참고로 둘 이상을 정의하려면 `@NamedStoredProcedureQueries`를 사용하면 된다.

Named 스토어드 프로시저 XML에 정의하기

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
        version="2.1">

    <named-stored-procedure-query name="multiply" procedure-name="proc_multiply"
        <parameter name="inParam"  mode="IN" class="java.lang.Integer" />
        <parameter name="outParam"  mode="OUT" class="java.lang.Integer" />
    </named-stored-procedure-query>

</entity-mappings>
```

이렇게 정의한 Named 스토어드 프로시저는 `em.createNamedStoredProcedureQuery()` 메서드에 등록한 Named 스토어드 프로시저 이름을 파라미터로 사용해서 찾아올 수 있다.

사용 코드는 다음과 같다.

```
StoredProcedureQuery spq = em.createNamedStoredProcedureQuery("multiply");

spq.setParameter("inParam", 100);
spq.execute();

Integer out = (Integer) spq.getOutputParameterValue("outParam");
System.out.println("out = " + out);
```