JAVA PERSISTENCE QUERY LANGUAGE [JPQL]

JPQL
Object Oriented Version of SQL

Java Persistence Query Language (JPQL) is an object model focused query language similar in nature to SQL.

JPQL understands notions like inheritance, polymorphism and association.

JPQL is a heavily-inspired-by a subset of HQL. A JPQL query is always a valid HQL query, the reverse is not true however.

Anatomy of a Query

- THIS is Effectively what member.findAll() does:

```
Query query = entityManager.createQuery(select m    from Member”);
List<Member> members =  (List<Member>) query.getResultList();
```

Entity references are Case Sensitive

```
Query query = entityManager.createQuery("select m from Member m
                      where m.memberNumber =  :number");

Member member=
  (Member) query.setParameter("number", number).getSingleResult();
```

Parameterized Query

Named & Ordinal Parameters

Alternative to named parameters [:name]

```
Query query = entityManager.createQuery("select m from Member m
                      where m.memberNumber = ?1");

Member member=
  (Member) query.setParameter(1,number).getSingleResult();
```

The form of ordinal parameters is a question mark (?) followed by a positive int number.

Aside from syntax - named parameters and ordinal parameters are identical.

Named parameters provide added clarity - preferred over ordinal parameters.

They BOTH prevent SQL injection

dangerous characters are automatically escaped by the JDBC driver

JPQL Syntax

- **CLAUSES:**
  - SELECT, FROM, WHERE, GROUP BY, HAVING and ORDER BY
- **OPERATORS**
- **Navigation operator** (.)

- **Arithmetic operators:**
  * (multiplication), / (division), + (addition) and - (subtraction).

- **Comparison operators**:
  =, <>, <, <=,>, >=, IS [NOT] NULL, [NOT] BETWEEN,

- **Logical operators:** AND, OR, NOT.

Hibernate JPQL

Oracle JPQL

OpenJPA JPQL

ObjectDB JPQL

JPA Named Query

VALUE: Static, Precompiled

- **Declaration:**
- *@Entity*
- *@NamedQuery(name = “Member.findByNumber", query = "select m from Member m where m.memberNumber = :number")*
- **public** class Member{

- **Usage [in MemberDaoImpl]:**

```
Query query =
      entityManager.createNamedQuery(“Member.findByNumber”);
 Member member=
      (Member) query.setParameter("number", number).getSingleResult();
```

- **Multiple Declarations:**

@NamedQueries(value = {@NamedQuery(...) , @NamedQuery(…) })

Native Query

JPA supports native SQL. You can create these queries in a very similar way as JPQL queries and they can return managed entities

VALUE:

Takes advantage of database vendors specific features…

Handles very complex and DB-optimized SQL query

In JPQL

It is very difficult to achieve the performance of a DBA-optimized Query

Get an Individual class Object

The simplest way to map the result of a native query into a managed entity is to select all properties of the entity and provide its as a parameter to the createNativeQuery method.

Notice Ordinal Parameter binding..

-
  ```
  Query query = entityManager.createNativeQuery("SELECT m.* FROM
        Member m where m.memberNumber = ?", Member.class);
  ```

-
  ```
  Member member = (Member)
        query.setParameter(1,number).getSingleResult();
  ```

Native Query Select List

```
Query query = entityManager.createNativeQuery("SELECT
      m.member_id, m.age, m.title ,m.firstname, m.lastname,
      m.memberNumber FROM Member m",Member.class);

List<Object> memberList = (List<Object>) query.getResultList();
Member member = (Member)memberList.get(0);
```

Stored Procedure

DECLARE:

```
@NamedStoredProcedureQuery( name = "calculate", procedureName = "calculate",
 parameters = {
  @StoredProcedureParameter(mode=ParameterMode.IN,type=Double.class, name= "x"),
  @StoredProcedureParameter(mode = ParameterMode.IN, type = Double.class, name = "y"),
  @StoredProcedureParameter(mode = ParameterMode.OUT, type = Double.class, name = "sum")
 }
```

INVOKE:

```
StoredProcedureQuery query =
            this.entityManager.createNamedStoredProcedureQuery("calculate");
            query.setParameter("x", 1.23d);
            query.setParameter("y", 4.56d);
            query.execute();
Double sum = (Double) query.getOutputParameterValue("sum");
```

Criteria Query

- Criteria API is a programmatic approach to query instead of string based approach as in JPQL.
- **JPQL:**

```
Query query = entityManager.createQuery("select m from Member m
                      where m.memberNumber =:number");
```

Criteria API Version: Pretty verbose and complex for simple queries

```
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
CriteriaQuery<Member> query=  criteriaBuilder.createQuery(Member.class);
Root<Member> memberRoot = query.from(Member.class);
query.select(memberRoot);
query.where(criteriaBuilder.equal(memberRoot.get("memberNumber"),  number) );

Member member=
       (Member)  entityManager.createQuery(query).getSingleResult();
```

- **Good for Dynamic queries..*See CriteriaApi demo***

Named Entity Graph

- **RAISON D'ETRE:**
- FetchType.LAZY - is the recommendation for performance & scaling
- Hydrating the graph becomes an issue
- Manually "walking" the graph is complex & "tedious" –requires:

  Maintaining custom queries for variations of the graph

  OR

  Iteratively query parts of graph as needed

- **Entity Graph Alternative**
- Declaratively Identify attributes to fetch from the database.
- Independent of specific query
- Can be used either as a Fetch or a Load Graph.

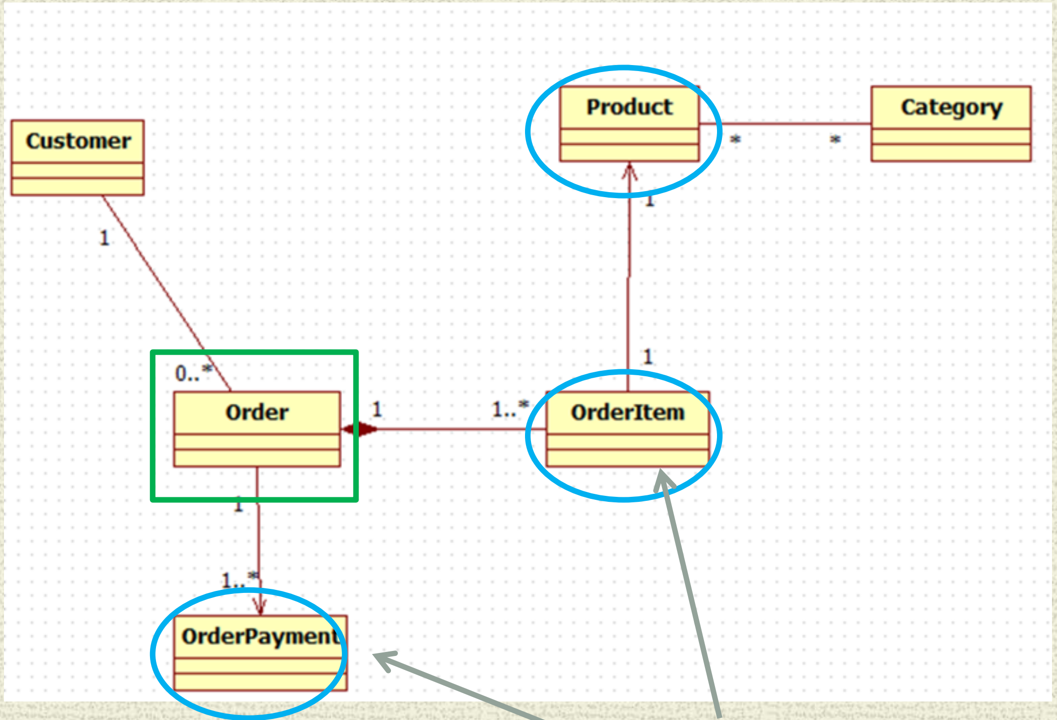
  Fetch graph - only the attributes specified by the entity graph will fetched

  Load graph - attributes not specified keep their default fetch type

  i.e., EAGER will occur

Example

```
• @NamedEntityGraph(name = "graph.Order.items",
•     attributeNodes= { @NamedAttributeNode(value = "items", subgraph= "items"),
•                   @NamedAttributeNode(value = "payments")} ,
•         subgraphs = @NamedSubgraph(name = "items",
•                   attributeNodes = @NamedAttributeNode("product")))
```



This Graph will “hydrate” the OrderPayments, as well as the OrderItems AND the Product on each OrderItem

See NamedEntityGraph Demo

```
private Set<OrderItem> items = new HashSet<OrderItem>();
private Set<OrderPayment> payments = new HashSet<OrderPayment>();
```

Query Hints

- JPA and Hibernate support a set of hints to provide additional information to the ORM to influence the execution of a query.

- Uses:

I. **Set a timeout for query**
II. ***Use an entity graph***
III. **Define the caching of a query result.**

Access NamedEntityGraph

A Hint is
Vendor specific Query Property
We SET it:
- on a query via setHint method
- In the find() and refresh() – by passing in a Map

- `OrderDaoImpl.java`

```
    EntityGraph graph = entityManager.getEntityGraph("graph.Order.items");
```

Query version
query.setHint(("javax.persistence.fetchgraph", graph) .getResultList();

```
    Order order = this.findOne(id, hints);
```

We are telling the find() to use the graph…

- `GenericDaoImpl.java`

```
public T findOne( Long id, Map<String,Object> hints ){
    return (T) entityManager.find( daoType, id, hints );
 }
```

NOTE: See NamedEntityGraphCartesian for query.setHint() example

JPA Query - Joins

- Join combines data from multiple tables as follows:

  Construct product of 2 tables

  Filter the rows using join condition

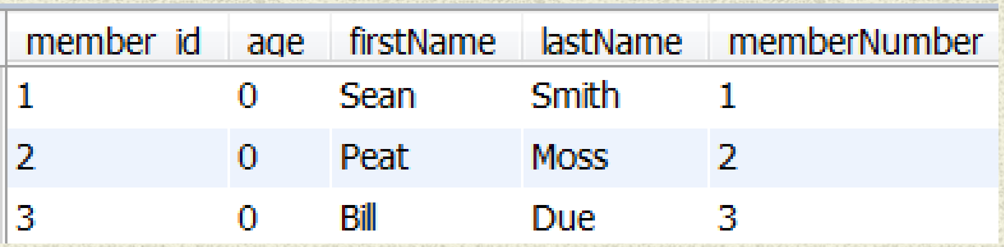
  WHERE

  join condition[boolean] determines if row is in result set
- JPA supports:

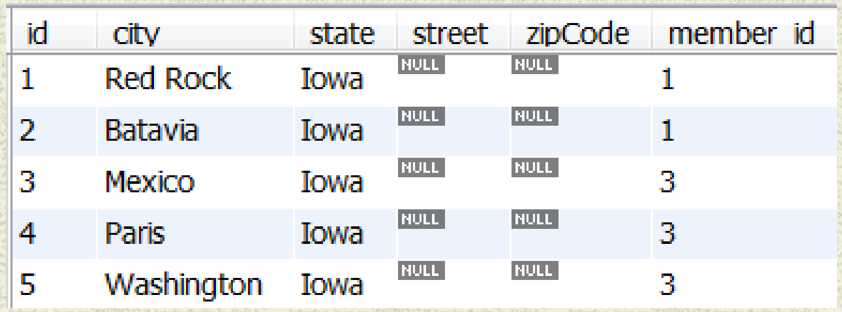
  Inner Join

  Left Outer Join

Join Examples



member_id	age	firstName	lastName	memberNumber
1	0	Sean	Smith	1
2	0	Peat	Moss	2
3	0	Bill	Due	3



id	city	state	street	zipCode	member_id
1	Red Rock	Iowa	NULL	NULL	1
2	Batavia	Iowa	NULL	NULL	1
3	Mexico	Iowa	NULL	NULL	3
4	Paris	Iowa	NULL	NULL	3
5	Washington	Iowa	NULL	NULL	3

Inner Join

Use: JOIN FETCH

Sean	Red Rock
	Batavia
Bill	Mexico
	Paris
	Washington

Only where there are matches between the 2 tables

Left Outer Join

Use: LEFT JOIN FETCH

Sean	Red Rock
	Batavia
Peat	
Bill	Mexico
	Paris
	Washington

All rows from “Left Side”

See JPQLJoinTypes **Demo**

JPA Right Outer Join NOT Supported

- Right outer joins are rarely used; developers always think from left to right and put the driving table first.

Hibernate specific Version

```
Query query = session.createQuery("from Member m right outer join m.address a ");
List<Object[]> memberAddress = query.list();
```

Sean	Red Rock
	Batavia
Bill	Mexico
	Paris
	Washington
	Russia

All rows from "Right Side"

JPA - Do LEFT outer FROM Address

See JPQLJoinTypes Demo

Main Point

- The persistence query language operates over the defined entity mappings, allowing us to transverse the complex associations with the same object oriented paradigm of our programs.

- ***Science of Consciousness***: *Enlivening Transcendental Consciousness empowers our mind with the ability to recognize complex interactions.*