Java Persistence/JPQL

JPQL

The Java Persistence Query Language (JPQL) is the query language defined by JPA. JPQL is similar to SQL, but operates on objects, attributes and relationships instead of tables and columns. JPQL can be used for reading (`SELECT`), as well as bulk updates (`UPDATE`) and deletes (`DELETE`). JPQL can be used in a `NamedQuery` (through annotations or XML) or in dynamic queries using the `EntityManagercreateQuery()` API.

For the JPQL BNF seeBNF.

Select Queries

Select queries can be used to read objects from the database. Select queries can return a single object or data element, a list of objects or data elements, or an object array of multiple objects and data.

Select query examples

```
// Query for a List of objects.
Query query = em.createQuery("Select e FROM Employee e WHERE e.salary > 100000" );
List<Employee> result = query.getResultList();

// Query for a single object.
Query query = em.createQuery("Select e FROM Employee e WHERE e.id = :id" );
query.setParameter("id", id);
Employee result2 = (Employee)query.getSingleResult();

// Query for a single data element.
Query query = em.createQuery("Select MAX(e.salary) FROM Employee e" );
BigDecimal result3 = (BigDecimal)query.getSingleResult();

// Query for a List of data elements.
Query query = em.createQuery("Select e.firstName FROM Employee e" );
List<String> result4 = query.getResultList();

// Query for a List of element arrays.
Query query = em.createQuery("Select e.firstName, e.lastName FROM Employee e" );
List<Object[]> result5 = query.getResultList();
```

SELECT Clause

The SELECT clause can contain object expressions, attribute expressions, functions, sub-selects, constructors and aggregation functions.

Aggregation functions

Aggregation functions can include summary information on a set of objects. These include MIN, MAX, AVG, SUM, COUNT. These functions can be used to return a single result, or can be used with a GROUP BY to return multiple results.

```
SELECT COUNT(e) FROM Employee e
```

```
SELECT MAX(e.salary) FROM Employee e
```

Constructors

The NEW operator can be used with the fully qualified class name to return data objects from a JPQL query. These will not be managed objects, and the class must define a constructor that matches the arguments of the constructor and their types. Constructor queries can be used to select partial data or reporting data on objects, and get back a class instance instead of an object array

```
SELECT NEW com.acme.reports.EmpReport(e.firstName, e.lastName, e.salary) FROM Employee e
```

FROM Clause

The FROM clause defines what is being queried. A typical FROM clause will contain the entity name being queried and assign it an alias.

```
SELECT e FROM Employee e
```

JPQL allows for multiple root level objects to be queried. Caution should be used when doing this, as it can result in Cartesian products of the two table.The WHERE clause should ensure the two objects are joined in some way

```
SELECT e, a FROM Employee e, MailingAddress a WHERE e.address = a.address
```

The entity name used in JPQL comes from the name attribute of the @Entity annotation or XML. It defaults to the simple entity class name. Some JPA providers also allow for the full class name.

> EclipseLink : TopLink - allow for the fully qualified class name of the entity to be used.

JOIN

A JOIN clause can also be used in the FROM clause. The JOIN clause allows any of the object's relationships to be joined into the query so they can be used in the WHERE clause. JOIN does not mean the relationships will be fetched, unless the FETCH option is included.

```
SELECT e FROM Employee e JOIN e.address a WHERE a.city = :city
```

JOIN can be used with OneToOne, ManyToOne, OneToMany, ManyToMany and ElementCollection mappings. When used with a collection relationship you can join the same relationship multiple times to query multiple independent values.

```
SELECT e FROM Employee e JOIN e.projects p JOIN e.projects p2 WHERE p.name = :p1 and p2.name = :p2
```

JOIN FETCH

The FETCH option can be used on a JOIN to fetch the related objects in a single query. This avoids additional queries for each of the object's relationships, and ensures that the relationships have been fetched if they were LAZY

```
SELECT e FROM Employee e JOIN FETCH e.address
```

JOIN FETCH does not allow an alias in the JPA spec, but some JPA providers may allow it.

> EclipseLink : TopLink - allow an alias.

LEFT JOIN

By default all joins in JPQL are INNER joins. This means that results that do not have the relationship will be filtered from the query results. To avoid this, a join can be defined as an OUTER join using the LEFT options.

```
SELECT e FROM Employee e LEFT JOIN e.address a ORDER BY a.city
```

ON (JPA 2.1)

The join condition used for a join comes from the mapping's join columns. This means that the JPQL user is normally free from having to know how every relationship is joined. In some cases it is desirable to append additional conditions to the join condition, normally in the case of outer joins.This can be done through the ON clause.The ON clause is defined in the JPA 2.1 specification, and may be supported by some JPA providers.

> EclipseLink : Hibernate : TopLink - support the ON clause.

```
SELECT e FROM Employee e LEFT JOIN e.address a ON a.city = :city
```

Sub-selects in FROM clause

JPA does not support sub-selects in the FROM clause. Some JPA providers may support this.

> EclipseLink : TopLink - support sub-selects in the FROM clause.

```
query.append(" select pg.regPatron, pso.nombreComercial as nombre from DitPatronGeneral pg ");
       query.append(" join pg.ditPatronSujetoObligado pso");
       query.append(" join pso.ditPersonaMoral pm ");
       query.append(" where pg.regPatron in :1   ");
       query.append(" union ");
       query.append(
               " select concat(:dp.nomNombre , :dp.nomPrimerApellido , :dp.nomSegundoApellido) as nombre
from DitPatronGeneral pg");
       query.append(" join pg.ditPatronSujetoObligado pso ");
       query.append(" join pso.ditPersonaFisica pf ");
       query.append(" join pf.ditPersona dp ");
       query.append(" where pg.regPatron in :2 ");
```

```
       List<String> map = new ArrayList<String>();

       Query queryRegistroPatronal = em.createQuery(query.toString());
       queryRegistroPatronal.setParameter("1", regPatronalLista);
       queryRegistroPatronal.setParameter("2", regPatronalLista);
```

ORDER BY clause

ORDER BY allows the ordering of the results to be specified. Multiple values can be ordered, either ascending (ASC) or descending (DESC). The JPA 1.0 and 2.0 BNFs restrict the usage of functions and sub-selects in the ORDER BY, but the JPA 2.1 draft removes most of these.

> EclipseLink : TopLink - allows functions, sub-selects, NULLS FIRST/LAST, object expressions, and other operations in the ORDER BY clause.

```
SELECT e FROM Employee e ORDER BY e.lastName ASC, e.firstName, ASC
```

```
SELECT e FROM Employee e ORDER BY UPPER(e.lastName)
```

GROUP BY Clause

GROUP BY allows for summary information to be computed on a set of objects. GROUP BY is normally used in conjunction with aggregation functions.

```
SELECT AVG(e.salary), e.address.city FROM Employee e GROUP BY e.address.city
```

```
SELECT AVG(e.salary), e.address.city FROM Employee e GROUP BY e.address.city ORDER BY AVG(e.salary)
```

```
SELECT e, COUNT(p) FROM Employee e LEFT JOIN e.projects p GROUP BY e
```

HAVING Clause

The HAVING clause allows for the results of a GROUP BY to be filtered.

```
SELECT AVG(e.salary), e.address.city FROM Employee e GROUP BY e.address.city HAVING AVG(e.salary) >
100000
```

UNION

JPA does not support the SQL UNION, INTERSECT and EXCEPT operations.Some JPA providers may support these.

> EclipseLink : TopLink - support UNION, INTERSECT, and EXCEPT.

WHERE Clause

The WHERE clause is normally the main part of the query as it defines the conditions that filter what is returned. The WHERE clause can use any comparison operation, logical operation, functions, attributes, objects, and sub-selects. The comparison operations include =, <, >, <=, >=, <>, LIKE, BETWEEN, IS NULL, and IN. NOT can also be used with any comparison operation (NOT LIKE, NOT BETWEEN, IS NOT NULL, NOT IN).The logical operations include AND, OR, and NOT

Comparison operations

Operation	Description	Example
=	equal	`e.firstName = 'Bob'`
<	less than	`e.salary < 100000`
>	greater than	`e.salary > :sal`
<=	less than or equal	`e.salary <= 100000`
>=	greater than or equal	`e.salary >= :sal`
LIKE	evaluates if the two string match, '%' and '_' are valid wildcards, and ESCAPE character is optional	`e.firstName LIKE 'A%' OR e.firstName NOT LIKE '%._%' ESCAPE '.'`

BETWEEN	evaluates if the value is between the two values	`e.firstName BETWEEN 'A' AND 'C'`
IS NULL	compares the value to null, databases may not allow or have unexpected results when using = with null	`e.endDate IS NULL`
IN	evaluates if the value is contained in the list	`e.firstName IN ('Bob', 'Fred', 'Joe')`

The IN operation allows for a list of values or parameters, a single list parameter, or a sub-select.

```
e.firstName IN (:name1, :name2, :name3)
e.firstName IN (:name1)
e.firstName IN :names
e.firstName IN (SELECT e2.firstName from Employee e2 WHERE e2.lastName = 'Smith')
```

A sub-select can be used with any operation provided it returns a single value, or if the ALL or ANY options are used. ALL indicates the operation must be true for all elements returned by the sub-select, ANY indicates the operation must be true for any of the elements returned by the sub-select.

```
e.firstName = (SELECT e2.firstName from Employee e2 WHERE e2.id = :id)
e.salary < (SELECT e2.salary from Employee e2 WHERE e2.id = :id)
e.firstName = ANY (SELECT e2.firstName from Employee e2 WHERE e.id <> e.id)
e.salary <= ALL (SELECT e2.salary from Employee e2)
```

Update Queries

You can perform bulk update of entities with the `UPDATE` statement. This statement operates on a single entity type and sets one or more single-valued properties of the entity subject to the condition in the `WHERE` clause. Update queries provide an equivalent to the `SQL UPDATE` statement, but with JPQL conditional expressions.

Update queries do not allow joins, but do support sub-selects. OneToOne and ManyToOne relationships can be traversed in the WHERE clause. Collection relationships can still be queried through using an EXISTS in the WHERE clause with a sub-select. Update queries can only update attributes of the object or its embeddables, its relationships cannot be updated. Complex update queries are dependent on the database's update support, and may make use of temp tables on some databases.

Update queries should only be used for bulk updates, regular updates to objects should be made by using the object's set methods within a transaction and committing the changes.

Update queries return the number of modified rows on the database (row count).

This example demonstrates how to use an update query to give employees a raise. The `WHERE` clause contains the conditional expression.

Update query example

```
Query query = em.createQuery("UPDATE Employee e SET e.salary = 60000 WHERE e.salary = 50000"  );
int rowCount = query.executeUpdate();
```

The persistence context is not updated to reflect results of update operations. If you use a transaction-scoped persistence context, you should either execute the bulk operation in a transaction all by itself, or be the first operation in the transaction. That is because any entity actively managed by the persistence context will remain unaware of the actual changes occurring at the database level.

The objects in the shared cache that match the update query will be invalidated to ensure subsequent persistence contexts see the updated data.

Delete Queries

You can perform bulk removal of entities with the `DELETE` statement. Delete queries provide an equivalent to the `SQL DELETE` statement, but with JPQL conditional expressions.

Delete queries do not allow joins, but do support sub-selects. OneToOne and ManyToOne relationships can be traversed in the WHERE clause. Collection relationships can still be queried through using an EXISTS in the WHERE clause with a sub-select. Complex delete queries are dependent on the database's delete support, and may make use of temp tables on some databases.

Delete queries should only be used for bulk deletes, regular deletes to objects should be performed through calling the `EntityManagerremove()` API.

Delete queries return the number of deleted rows on the database (row count).

This example demonstrates how to use a delete query to remove all employees who are not assigned to a department. The `WHERE` clause contains the conditional expression.

Delete query example

```
Query query = em.createQuery("DELETE FROM Employee e WHERE e.department IS NULL" );
int rowCount = query.executeUpdate();
```

Delete queries are polymorphic: any entity subclass instances that meet the criteria of the delete query will be deleted. However, delete queries do not honor cascade rules: no entities other than the type referenced in the query and its subclasses will be removed, even if the entity has relationships to other entities with cascade removes enabled. Delete queries will delete the rows from join and collection tables.

The persistence context may not be updated to reflect results of delete operations. If you use a transaction-scoped persistence context, you should either execute the bulk operation in a transaction all by itself, or be the first operation in the transaction. That is because any entity actively managed by the persistence context will remain unaware of the actual changes occurring at the database level.

Parameters

JPA defines named parameters, and positional parameters. Named parameters can be specified in JPQL using the syntax `:<name>`. Positional parameters can be specified in JPQL using the syntax `?` or `?<position>`. Positional parameters start at position `1` not `0`.

Named parameter query example

```
Query query = em.createQuery("SELECT e FROM Employee e WHERE e.firstName = :first and e.lastName =
:last");
query.setParameter("first", "Bob");
query.setParameter("last", "Smith");
List<Employee> list = query.getResultList();
```

Positional parameter query example

```
Query query = em.createQuery("SELECT e FROM Employee e WHERE e.firstName = ? and e.lastName = ?" );
query.setParameter(1, "Bob");
query.setParameter(2, "Smith");
List<Employee> list = query.getResultList();
```

Literals

Literal values can be in-lined in JPQL for standard Java types. In general it is normally better to use parameters instead of in-lining values. In-lined arguments will prevent the JPQL from benefiting from the EclipseLink's JPQL parser cache, and can potentially make the application vulnerable to JPQL injections attacks.

Each Java types defines its own in-lining syntax:

- `String - '<string>'`

```
SELECT e FROM Employee e WHERE e.name = 'Bob'
```

  - To define a `'` (quote) character in a string, the quote is double quoted, i.e. `'Baie-D''Urfé'`.
- `Integer - +|-<digits>`

```
SELECT e FROM Employee e WHERE e.id = 1234
```

- `Long - +|-<digits>L`

```
SELECT e FROM Employee e WHERE e.id = 1234L
```

- `Float - +|-<digits>.<decimale><exponent>F`

```
SELECT s FROM Stat s WHERE s.ratio > 3.14F
```

- `Double - +|-<digits>.<decimale><exponent>D`

```
SELECT s FROM Stat s WHERE s.ratio > 3.14e32D
```

- `Boolean - TRUE | FALSE`

```
SELECT e FROM Employee e WHERE e.active = TRUE
```

- `Date - {d'yyyy-mm-dd'}`

```
SELECT e FROM Employee e WHERE e.startDate = {d'2012-01-03'}
```

- `Time - {t'hh:mm:ss'}`

```
SELECT e FROM Employee e WHERE e.startTime = {t'09:00:00'}
```

- `Timestamp - {ts'yyyy-mm-dd hh:mm:ss.nnnnnnnnn'}`

```
SELECT e FROM Employee e WHERE e.version = {ts'2012-01-03 09:00:00.000000001' }
```

- `Enum - package.class.enum`

```
SELECT e FROM Employee e WHERE e.gender = org.acme.Gender.MALE
```

- `null - NULL`

```
UPDATE Employee e SET e.manager = NULL WHERE e.manager = :manager
```

Functions

JPQL supports several database functions. These functions are database independent in name and syntax, but require database support. If the database supports an equivalent function or different syntax the standard JPQL function is supported, if the database does not provide any way to perform the function, then it is not supported. For mathematical functions (+, -, /, *) BEDMAS rules apply. Some JPA provider may support additional functions.

EclipseLink : TopLink - also support CAST, EXTRACT, and REGEXP.

JPQL supported functions

Function	Description	Example
-	subtraction	`e.salary - 1000`
+	addition	`e.salary + 1000`
*	multiplication	`e.salary * 2`
/	division	`e.salary / 2`
ABS	absolute value	`ABS(e.salary - e.manager.salary)`
CASE	defines a case statement	`CASE e.status WHEN 0 THEN 'active' WHEN 1 THEN 'consultant' ELSE 'unknown' END`
COALESCE	evaluates to the first non null argument value	`COALESCE(e.salary, 0)`
CONCAT	concatenates two or more string values	`CONCAT(e.firstName, ' ', e.lastName)`
CURRENT_DATE	the current date on the database	`CURRENT_DATE`
CURRENT_TIME	the current time on the database	`CURRENT_TIME`
CURRENT_TIMESTAMP	the current date-time on the database	`CURRENT_TIMESTAMP`
LENGTH	the character/byte length of the character or binary value	`LENGTH(e.lastName)`
LOCATE	the index of the string within the string, optionally starting at a start index	`LOCATE('-', e.lastName)`

LOWER	convert the string value to lower case	`LOWER(e.lastName)`
MOD	computes the remainder of dividing the first integer by the second	`MOD(e.hoursWorked / 8)`
NULLIF	returns null if the first argument to equal to the second argument, otherwise returns the first argument	`NULLIF(e.salary, 0)`
SQRT	computes the square root of the number	`SQRT(o.result)`
SUBSTRING	the substring from the string, starting at the index, optionally with the substring size	`SUBSTRING(e.lastName, 0, 2)`
TRIM	trims leading, trailing, or both spaces or optional trim character from the string	`TRIM(TRAILING FROM e.lastName), TRIM(e.lastName), TRIM(LEADING '-' FROM e.lastName)`
UPPER	convert the string value to upper case	`UPPER(e.lastName)`

Special Operators

JPQL defines several special operators that are not database functions, but have special meaning in JPQL. These include INDEX, KEY, SIZE, IS EMPTY, TYPE, FUNCTION and TREAT.

> EclipseLink : TopLink - also support FUNCTION, TREAT, FUNC, OPERATOR, SQL, COLUMN, and TABLE.

JPQL special operators

Function	Description	Example
INDEX	the index of the ordered List element, only supported with @OrderColumn	`SELECT toDo FROM Employee e JOIN e.toDoList toDo WHERE INDEX(toDo) = 1`
KEY	the key of the Map element	`SELECT p FROM Employee e JOIN e.priorities p WHERE KEY(p) = 'high'`
SIZE	the size of the collection relationships, this evaluates to a sub-select	`SELECT e FROM Employee e WHERE SIZE(e.managedEmployees) < 2`
IS EMPTY, IS NOT EMPTY	evaluates to true if the collection relationship is empty, this evaluates to a sub-select	`SELECT e FROM Employee e WHERE e.managedEmployees IS EMPTY`
MEMBER OF, NOT MEMBER OF	evaluates to true if the collection relationship contains the value, this evaluates to a sub-select	`SELECT e FROM Employee e WHERE 'write code' MEMBER OF e.responsibilities`
TYPE	the inheritance discriminator value	

		`SELECT p FROM Project p WHERE TYPE(p) = LargeProject`
TREAT	treat (cast) the object as its subclass value (JPA 2.1 draft)	`SELECT e FROM Employee JOIN TREAT(e.projects as LargeProject) p WHERE p.budget > 1000000`
FUNCTION	call a database function (JPA 2.1 draft)	`SELECT p FROM Phone p WHERE FUNCTION('TO_NUMBER', p.areaCode) > 613`