Selection - The FROM clause

The FROM clause defines from which entities the data gets selected. Hibernate, or any other JPA implementation, maps the entities to the according database tables.

The syntax of a JPQL FROM clause is similar to SQL but uses the entity model instead of table or column names.

```
SELECT a FROM Author a
```

Joining multiple entities

Inner Joins

If you want to select data from multiple entities, like all authors and the books they've written, you have to join the entities in the FROM clause. The easiest way to do that is to use the defined associations of an entity like in the following code snippet.

```
SELECT a, b FROM Author a JOIN a.books b
```

JOINs of unrelated entities are not supported by the JPA specification, but you can use a theta join which creates a cartesian product and restricts it in the WHERE clause to the records with matching foreign and primary keys.

```
SELECT b, p FROM Book b, Publisher p WHERE
```

Left Outer Joins

If you want to include the authors without published books, you have to use a LEFT JOIN, like in the following code snippet.

```
SELECT a, b FROM Author a LEFT JOIN a.books b
```

Additional Join Conditions

Sometimes you only want to join the related entities which fulfill additional conditions. Since JPA 2.1, you can do this for INNER JOINs and LEFT JOINs with an additional ON statement

```
SELECT a, p FROM Author a JOIN a.publications p
                                ON p.publishingDate > ?1
```

Path expressions or implicit joins

Path expressions create implicit joins and are one of the benefits provided by the entity model. You can use the '.' operator to navigate to related entities like I do in the following code snippet.

```
SELECT b FROM Book b
            WHERE b.publisher.name LIKE '%es%
```

Polymorphism and Downcasting

Polymorphism

When you choose an inheritance strategy that supports polymorphic queries, your query selects all instances of the specified class and its subclasses.

```
SELECT p FROM Publication p
```

Or you can select a specific subtype of a Publication, like a BlogPost.

```
SELECT b FROM BlogPost b
```

Downcasting

Since JPA 2.1, you can also use the TREAT operator for downcasting in FROM and WHERE clauses. I use that in the following code snippet to select all Author entities with their related Book entities.

```
SELECT a, p FROM Author a
                JOIN treat (a.publications AS Book) p
```

Restriction - The WHERE clause

The next important part of a JPQL query is the WHERE clause which you can use to restrict the selected entities to the ones you need for your use case. The syntax is very similar to SQL, but JPQL supports only a small subset of the SQL features. If you need more sophisticated features for your query, you can use a native SQL query.

JPQL supports a set of basic operators to define comparison expressions. Most of them are identical to the comparison operators supported by SQL and you can combine them with the logical operators AND, OR and NOT into more complex expressions.

Operators for single-valued expressions

- Equal: author.id = 10
- Not equal: author.id <> 10
- Greater than: author.id > 10
- Greater or equal then: author.id => 10
- Smaller than: author.id < 10
- Smaller or equal then: author.id <= 10
- Between: author.id BETWEEN 5 and 10
- Like: author.firstName LIKE '%and%'
  The % character represents any character sequence. This example restricts the query result to all Authors with a firstName that contains the String 'and', like Alexander or Sandra. You can use an _ instead of % as a single character wildcard. You can also negate the operator with NOT to exclude all Authors with a matching firstName.
- Is null: author.firstName IS NULL
  You can negate the operator with NOT to restrict the query result to all Authors who's firstName IS NOT NULL.
- In: author.firstName IN ('John', 'Jane')
  Restricts the query result to all Authors with the first name John or Jane.

Operators for collection expressions

- Is empty: author.books IS EMPTY
  Restricts the query result to all Authors without associated Book entities. You can negate the operator (IS NOT EMPTY) to restrict the query result to all Authors with associated Book entities.
- Size: size(author.books) > 2
  Restricts the query result to all Authors who are associated with more than 2 Book entities.
- Member of: :myBook member of author.books
  Restricts the query result to all Authors who are associated with a specific Book entity.

Example

You can use one or more of the operators to restrict your query result. The following query returns all Author entities with a firstName attribute that contains the String “and” and an id attribute greater or equal 20 and who have written at least 5 books.

```
SELECT a FROM Author a
    WHERE a.firstName like ‘%and%’
        and a.id >= 20
        and size(author.books) >= 5
```

Projection - The SELECT clause

The projection of your query defines which information you want to retrieve from the database. This part of the query is very different to SQL. In SQL, you specify a set of database columns and functions as your projection. You can do the same in JPQL by selecting a set of entity attributes or functions as scalar values, but you can also define entities or constructor calls as your projection. Hibernate, or any other JPA implementation, maps this information to a set of database columns and function calls to define the projection of the generated SQL statement.

Entities

Entities are the most common projection in JPQL queries. Hibernate uses the mapping information of the selected entities to determine the database columns it has to retrieve from the database. It then maps each row of the result set to the selected entities.

```
SELECT a FROM Author a
```

Scalar values

Scalar value projections are very similar to the projections you know from SQL. Instead of database columns, you select one or more entity attributes or the return value of a function call with your query.

```
SELECT a.firstName, a.lastName FROM Author a
```

Constructor references

Constructor references are a good projection for read-only use cases. They're more comfortable to use than scalar value projections and avoid the overhead of managed entities.

JPQL allows you to define a constructor call in the SELECT clause.

```
SELECT new org.thoughts.on.java.model.AuthorValue
            (a.id, a.firstName, a.lastName)
      FROM Author a
```

Distinct query results

You probably know SQL's DISTINCT operator which removes duplicates from a projection. JPQL supports this operator as well.

```
SELECT DISTINCT a.lastName FROM Author a
```

Functions

Functions are another powerful feature of JPQL that you probably know from SQL. It allows you to perform basic operations in the WHERE and SELECT clause. You can use the following functions in your query:

- *upper(String s)*: transforms String s to upper case
- *lower(String s)*: transforms String s to lower case
- *current_date()*: returns the current date of the database
- *current_time()*: returns the current time of the database
- *current_timestamp()*: returns a timestamp of the current date and time of the database
- *substring(String s, int offset, int length)*: returns a substring of the given String s
- *trim(String s)*: removes leading and trailing whitespaces from the given String s
- *length(String s)*: returns the length of the given String s
- *locate(String search, String s, int offset)*: returns the position of the String search in s. The search starts at the position offset
- *abs(Numeric n)*: returns the absolute value of the given number
- *sqrt(Numeric n)*: returns the square root of the given number
- *mod(Numeric dividend, Numeric divisor)*: returns the remainder of a division
- *treat(x as Type)*: downcasts x to the given Type
- *size(c)*: returns the size of a given Collection c
- *index(orderdCollection)*: returns the index of the given value in an ordered Collection

Grouping - The GROUP BY and HAVING clause

When you use aggregate functions, like count(), in your SELECT clause, you need to reference all entity attributes that are not part of the function in the GROUP BY clause.

The following code snippet shows an example that uses the aggregate function count() to count how often each last name occurs in the Author table.

```
SELECT a.lastName, COUNT(a)
    FROM Author a
    GROUP BY a.lastName
```

The HAVING clause is similar to the WHERE clause and allows you to define additional restrictions for your query. The main difference is that the restrictions defined in a HAVING clause are applied to a group and not to a row.

I use it in the following example to select all last names that start with a 'B' and count how often each of them occurs in the Author table.

```
SELECT a.lastName, COUNT(a)
    FROM Author a
    GROUP BY a.lastName
    HAVING a.lastName LIKE 'B%'
```

Ordering - The ORDER BY clause

You can define the order in which the database shall return your query results with an ORDER BY clause. Its definition in JPQL is similar to SQL. You can provide one or more entity attributes to the ORDER BY clause and specify an ascending (ASC) or a descending (DESC) order.

```
SELECT a FROM Author a
    ORDER BY a.lastName ASC, a.firstName DESC
```

Subselects

A subselect is a query embedded into another query. It's a powerful feature you probably know from SQL. Unfortunately, JPQL supports it only in the WHERE clause and not in the SELECT or FROM clause.

Subqueries can return one or multiple records and can use the aliases defined in the outer query. The following example shows a query that uses a subquery to count all Books written by an Author and returns only the Authors who've written more than 1 Book.

```
SELECT a FROM Author a
    WHERE (SELECT count(b) FROM Book b WHERE a
        MEMBER OF b.authors ) > 1
```