### JPQL (Java Persistence Query Language)

\*\*Java Persistence Query Language (JPQL)\*\* is a query language used in Java applications to interact with relational databases through the \*\*Java Persistence API (JPA)\*\*. It is object-oriented and operates on the entity model, allowing developers to query entities and their relationships, rather than directly querying the database tables.

JPQL is similar to SQL but designed to work with Java objects rather than database tables and records. It uses the entity classes, attributes, and relationships to form queries.

---

### Key Features of JPQL:

1. \*\*Object-oriented\*\*: Queries are written using entity names and fields, not table names and columns.

2. \*\*Portable\*\*: JPQL is independent of the underlying database, meaning it can work across different databases without changes.

3. \*\*Powerful\*\*: Supports advanced querying, including joins, filtering, ordering, and aggregation.

---

### Basic Structure of a JPQL Query

JPQL follows a structure similar to SQL. The basic syntax includes the \*\*`SELECT`\*\*, \*\*`FROM`\*\*, and \*\*`WHERE`\*\* clauses, but these operate on entities and their fields rather than database tables and columns.

#### Example:

```java

SELECT u FROM User u WHERE u.age > 18

```

- \*\*`User`\*\*: Entity class.

- \*\*`u`\*\*: Alias for the entity.

- \*\*`u.age`\*\*: Refers to the `age` field of the `User` entity.

---

### Common JPQL Clauses

1. \*\*SELECT Clause\*\*: Retrieves data from entities.

```java

SELECT u FROM User u

```

2. \*\*WHERE Clause\*\*: Adds conditions to filter the data.

```java

SELECT u FROM User u WHERE u.age > 18

```

3. \*\*ORDER BY Clause\*\*: Sorts the results based on one or more fields.

```java

SELECT u FROM User u ORDER BY u.age DESC

```

4. \*\*JOIN Clause\*\*: Fetches data from related entities.

```java

SELECT o FROM Order o JOIN o.customer c WHERE c.name = 'John'

```

5. \*\*GROUP BY Clause\*\*: Groups results by one or more fields.

```java

SELECT u.age, COUNT(u) FROM User u GROUP BY u.age

```

6. \*\*HAVING Clause\*\*: Applies conditions to grouped results.

```java

SELECT u.age, COUNT(u) FROM User u GROUP BY u.age HAVING COUNT(u) > 1

```

---

### JPQL Query Types

#### 1. \*\*Select Queries\*\*

These are used to retrieve data from entities.

\*\*Example\*\*:

```java

SELECT u FROM User u WHERE u.name = 'John'

```

This query will retrieve all `User` entities where the `name` field is "John".

#### 2. \*\*Update Queries\*\*

These are used to update entities in the database.

\*\*Example\*\*:

```java

UPDATE User u SET u.active = true WHERE u.age > 18

```

This query updates all users older than 18 to set their `active` field to `true`.

#### 3. \*\*Delete Queries\*\*

These are used to remove entities from the database.

\*\*Example\*\*:

```java

DELETE FROM User u WHERE u.active = false

```

This query deletes all users whose `active` field is `false`.

---

### Example Entity for JPQL Queries

```java

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

private int age;

private boolean active;

// Getters and setters

}

```

---

### JPQL Query Examples

#### 1. \*\*Simple Select Query\*\*

```java

SELECT u FROM User u WHERE u.name = 'Alice'

```

This query retrieves the `User` entity where `name` is "Alice".

#### 2. \*\*Selecting Specific Fields\*\*

```java

SELECT u.name, u.age FROM User u

```

This query retrieves only the `name` and `age` fields of the `User` entity.

#### 3. \*\*Filtering with the WHERE Clause\*\*

```java

SELECT u FROM User u WHERE u.age > 25 AND u.active = true

```

This query retrieves all users who are older than 25 and are active.

#### 4. \*\*Using Aggregate Functions\*\*

```java

SELECT COUNT(u) FROM User u WHERE u.active = true

```

This query counts the number of active users.

#### 5. \*\*Ordering Results\*\*

```java

SELECT u FROM User u ORDER BY u.age ASC

```

This query retrieves all users, sorted by their age in ascending order.

#### 6. \*\*Joins in JPQL\*\*

Assuming `User` and `Order` entities have a relationship where one `User` can have many `Orders`.

```java

SELECT o FROM Order o JOIN o.user u WHERE u.name = 'John'

```

This query retrieves all orders placed by the user named "John".

---

### JPQL Named Queries

Named queries are pre-defined JPQL queries that can be reused throughout the application. They are defined using the `@NamedQuery` annotation in the entity class.

\*\*Example\*\*:

```java

@Entity

@NamedQuery(name = "User.findByName", query = "SELECT u FROM User u WHERE u.name = :name")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

// Getters and setters

}

```

\*\*Using a Named Query\*\*:

```java

Query query = entityManager.createNamedQuery("User.findByName");

query.setParameter("name", "Alice");

List<User> users = query.getResultList();

```

---

### JPQL with the EntityManager

In Spring Boot, JPQL queries are often executed using the `EntityManager` interface.

\*\*Example of executing a JPQL query using `EntityManager`:\*\*

```java

@Autowired

private EntityManager entityManager;

public List<User> getUsersOlderThan(int age) {

String jpql = "SELECT u FROM User u WHERE u.age > :age";

return entityManager.createQuery(jpql, User.class)

.setParameter("age", age)

.getResultList();

}

```

---

### JPQL Query Parameters

You can use named or positional parameters in JPQL queries to avoid SQL injection and make queries dynamic.

#### Named Parameters:

```java

SELECT u FROM User u WHERE u.name = :name

```

Usage:

```java

Query query = entityManager.createQuery("SELECT u FROM User u WHERE u.name = :name");

query.setParameter("name", "Alice");

```

#### Positional Parameters:

```java

SELECT u FROM User u WHERE u.name = ?1

```

Usage:

```java

Query query = entityManager.createQuery("SELECT u FROM User u WHERE u.name = ?1");

query.setParameter(1, "Alice");

```

---

### Advantages of JPQL

- \*\*Database-agnostic\*\*: JPQL queries are portable across different databases, unlike native SQL queries.

- \*\*Entity-oriented\*\*: Queries are written in terms of the object model (entities) rather than tables, making it easier for developers to work with.

- \*\*Type-safe\*\*: Since JPQL operates on entities and their fields, the queries are type-safe and checked at compile-time.

---

### Conclusion

JPQL is a powerful query language that integrates seamlessly with JPA, providing an object-oriented approach to database querying. It abstracts away the complexity of writing database-specific SQL queries, making it easier for developers to work with entity objects and relationships in Java-based applications. By leveraging JPQL, you can create portable, efficient, and maintainable database queries.